SciViews

R Tcl/Tk Recipes

GUI for R using tcltk2

James Wettenhall & Philippe Grosjean

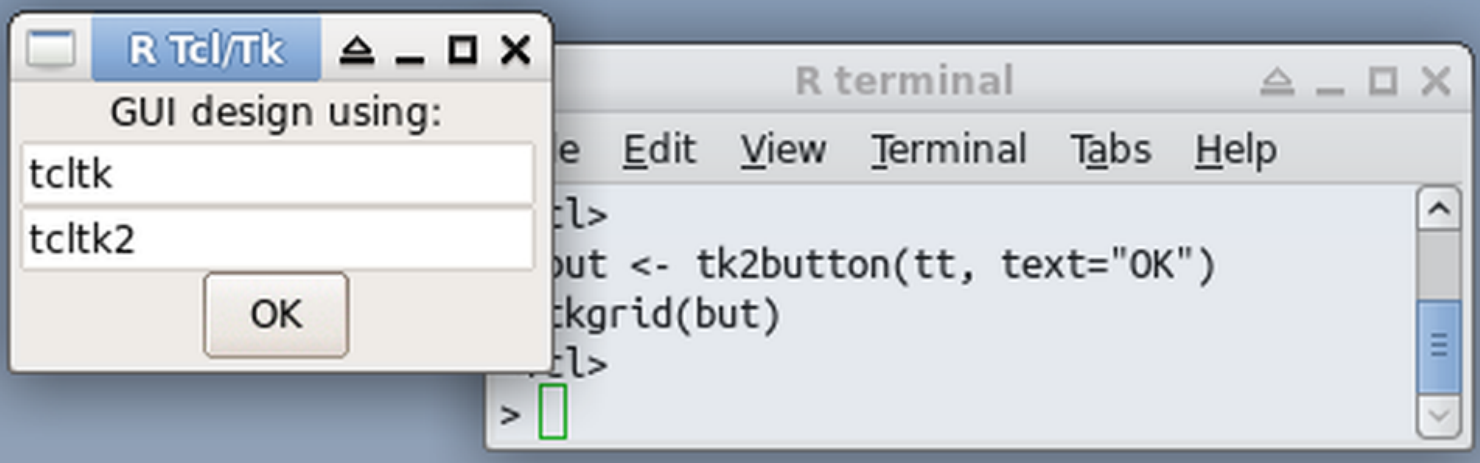
.

PHGROSJEAN@SCIVIEWS.ORG

HTTP://WWW.SCIVIEWS.ORG/

Version 0.1-0, December 2015

Contents



1. Getting started with Tcl/Tk in R

1.1 Prerequisites

Before starting the exploration of R Tcl/Tk recipes, you should install R (R Core Team (2015)). You also need the CRAN version of the **tcltk2** and **tkrplot** R packages. You can install them with this commands:

```
install.packages(c("tcltk2", "tkrplot"))
```

Make also sure you understand the bases of the **tcltk** package. You should read both R News articles by Peter Dalgaard (Dalgaard (2001), Dalgaard (2002)).

1.2 Introduction

Tcl (Tool Command Language) is a dynamic scripting language that is easily embedded in other applications. **Tk** is a cross-platform graphical user interface (GUI) toolkit. Both are useable from within R thanks to the tcltk package. The Tk toolkit is a decent one, but not the most feature-rich. However, the big advantage of Tcl/Tk is its wide availability in all platforms supported by R: the package is maintained by the R Core Team, and it is shipped with R itself. To check if Tcl/Tk is available, use the command `capabilities("tcltk")`.

The tcltk2 package is also available from CRAN. It offers additional possibilities and more widgets. These recipes show how to use both the tcltk and the tcltk2 packages with R to build a GUI, or to use other potentials of Tcl.

1.2.1 Other sources of R tcltk help/examples

1. Run `help.start()` in R to get HTML help, then click on `Packages`, then click on `tcltk` or `tcltk2`.
2. Read the ActiveTcl help and learn how to convert Tcl options to R arguments, e.g.
   `-background white`
   becomes in R
   `background = "white"`
3. Read Peter Dalgaard's articles in Rnews : Rnews 2001, Vol. 3 and Rnews 2002, Vol. 3.
4. Study the demos in the R `tcltk` package, e.g. `tkdensity` and `tkttest`.
5. Read and participate in R-Help and other R mailing lists.
6. Search the web for Tcl/Tk examples, and don't ignore them completely if they use a language other than R, e.g. Perl or Python. A lot can be learned from these examples.

1.3 Basic techniques

1.3.1 A simple Tk toplevel window with an OK button

The primary goal of the **tcltk** R package is to use the Tk graphical user interface (GUI) toolkit with R. Here is a Tk window with an `OK` button that just destroys the window when it is clicked:

```
# Import the tcltk package
library(tcltk)
# Create a new Tk toplevel window assigned to win1
win1 <- tktoplevel()
# Create a Tk button whose function (command) is to destroy the window win1
butOK <- tkbutton(win1, text = "OK",
  command = function() tkdestroy(win1))
# Place the button on the window, using the grid geometry manager
tkgrid(butOK)
```

You should get the following window:

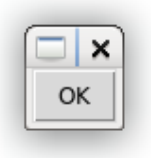


Figure 1.1: A Tk window

Click `OK` to close the window

Note that a Tk widget is not placed automatically inside its container[1]. You have to use one of the three Tk geometry managers (*grid*, *pack* or *place*, using respectively the

[1] A container is a widget that can contain other widgets. A toplevel window like `win1` is a container, while `butOK` is not.

tcltk functions `tkgrid()`, `tkpack()` and `tkplace()` in R)[2]. The *grid* manager is the most powerful and the most used one. It devides the container into a grid of rows and columns, arranges nicely the widgets in the grid, and then automatically resizes the container to best match its content (resulting here in a shrinked small window around the `OK` button).

Our Tk window and the way we manage it is indeed far from optimal. It can be ameliorated in four ways:

1. It would be nice to give a title to our Tk window. This can be done using `tktitle()`.
2. We could use **ttk** instead of **tk** widgets by replacing `tkbutton()` by `ttkbutton()`. The **ttk** widgets are styled according to a theme that makes your GUI look better, more modern, and sometimes more native (on Windows, for instance).
3. We should think about the size of the widgets and margins around them for a better layout. Our tiny `OK` button in the middle of a small window is not that nice. So, let's improve this.
4. It is nice to keep track of our Tk windows and widgets by assigning variables, like `win1` or `butOK` here. However, these variables clutter our workspace. They also do not reflect the hierarchy. `butOK` is embedded in `win1` at the Tk level. It makes clean up more difficult once the window is destroyed: you must get rid of both `win1` **and** `butOK` to free memory from items that are not needed any more. Finally, if you have two windows, each with an `OK` button, you should of course not call them both `butOK`. Also, reassigning `win1` before the first window is destroyed leads to problems[3]. With a more complex GUI, you easily end up with dozens of variables to keep track of your Tk widgets, and you may be at risk for clashes and hard-to-debug behaviour!

1.3.1.1 A better approach

Here is an improved version that implements all four points raised here above:

```
library(tcltk)
win2 <- tktoplevel()
# Give a title to the window
tktitle(win2) <- "Tk window"
# Create a Ttk button with a minimum size (note negative value) of six characters
# The command is a lot more complicated to make it survive a reassignation to win2
# (explanation is beyond the scope of this introductory tutorial)
# Assign inside win2 to avoid the inflation of variables in the global environment
# We assign to win2$env, instead of win2$, so that butOK is available to all
# shared versions of win1 (need further explanation!)
win2$env$butOK <- ttkbutton(win2, text = "OK", width = -6,
  command = (function(win) { force(win); function() tkdestroy(win)})(win2))
# Place the button on the window, with large margins around it
tkgrid(win2$env$butOK, padx = 70, pady = 30)
```

The button has now much more space around it. On Windows, it looks native, but on Linux it is still looking old-fashioned...

[2] Never mix Tk managers inside the same container!

[3] Rerun the previous code to recreate the window and the button. Do not close that window, but rerun `win1 <- tktoplevel()`. This will create a new window, as `win1`. Now, when you click the `OK` button on the *first* window, it is the *second* window that is destroyed!

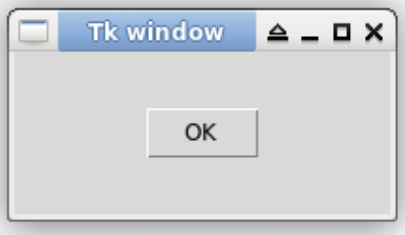


Figure 1.2: A Tk window

Our code is now becoming quite complicated. However, the **tcltk2** package would be helpful here.

1.3.1.2 The tcltk2 version

The **tcltk2** R package[4] provides more advanced Tk widgets, additional R-Tcl commands, more modern themes for Linux and Mac OS X and it simplifies the creation of GUI items. Here is how you could get the same window using **tcltk2**:

```
# NOTE THAT THIS DOES NOT WORK YET WITH THE VERSION OF TCLTK2 AVAILABLE ON CRAN
library(tcltk2)
# You can configure the window at creation. If you specify a manager, it will be
# automatically used for each child widget created, unless specified otherwise
win3 <- tk2toplevel(title = "Tk2 window", manage = "grid", padx = 70, pady = 30)
# Create and place the same button (note the simpler syntax)
win3$butOK <- tk2button(text = "OK", width = -6, command = TkCmd_destroy(parent))
```

The default theme on Linux is `clearlooks`, which gives the next visual:

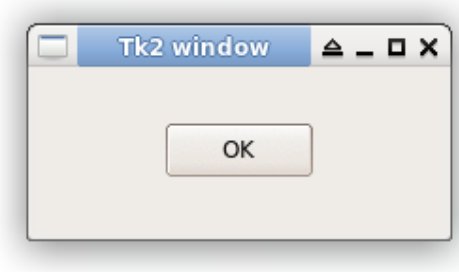


Figure 1.3: A Tk window

1.3.2 Message boxes in R tcltk

The following code demonstrates a simple "Hello World" message box.

```
library(tcltk2) # For themed message boxes; library(tcltk) is fine too here
res <- tkmessageBox(title = "Greetings from R TclTk",
```

[4]Install the package from CRAN with the instruction `install.packages("tcltk2")`.

```
  message = "Hello, world!", icon = "info", type = "ok")
```

Figure 1.4: A Tk window

After pressing the `OK` button, we can check the return value of the message box function.

```
res               # This is a Tcl variable
## <Tcl> ok
tclvalue(res)     # Get the value from a Tcl variable
## [1] "ok"
as.character(res) # It works also that way
## [1] "ok"
```

We notice that the window size for the message box is too small to display the full title in the title bar, and unfortunately message boxes are not resizable by default (whereas tktoplevel windows are resizable by default). A simple way to fix this (which is admittedly not very elegant), is to add spaces on the end of the message to make it at least as long as the title.

```
res <- tkmessageBox(title = "Greetings from R TclTk",
  message = "Hello, world!                ", icon = "info", type = "ok")
```

Figure 1.5: A Tk window

Of course, sometimes it is desirable to have other buttons and/or other icons in a message box. The following examples illustrate some typical choices of buttons and icons.

```
tkmessageBox(message = "An error has occurred!", icon = "error", type = "ok")
```

```
tkmessageBox(message = "This is a warning!", icon = "warning", type = "ok")
```

```
tkmessageBox(message = "Do you want to save before quitting?",
    icon = "question", type = "yesnocancel", default = "yes")
```

Figure 1.6: A Tk window

Figure 1.7: A Tk window

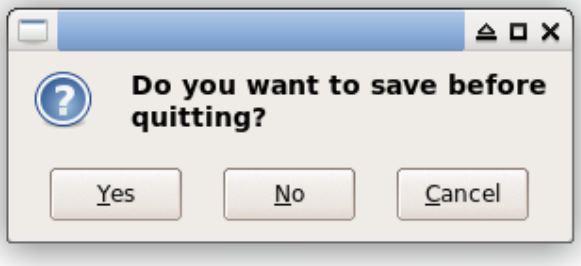


Figure 1.8: A Tk window

1.3.3 File Open/Save dialogs in R tcltk

1.3.3.1 The Open file dialog

```
library(tcltk2)
filename <- tclvalue(tkgetOpenFile()) # Very simple, isn't it?
if (!nchar(filename)) {
  tkmessageBox(message = "No file was selected!")
} else {
  tkmessageBox(message = paste("The file selected was", filename))
}
```

The code above produces the following window:

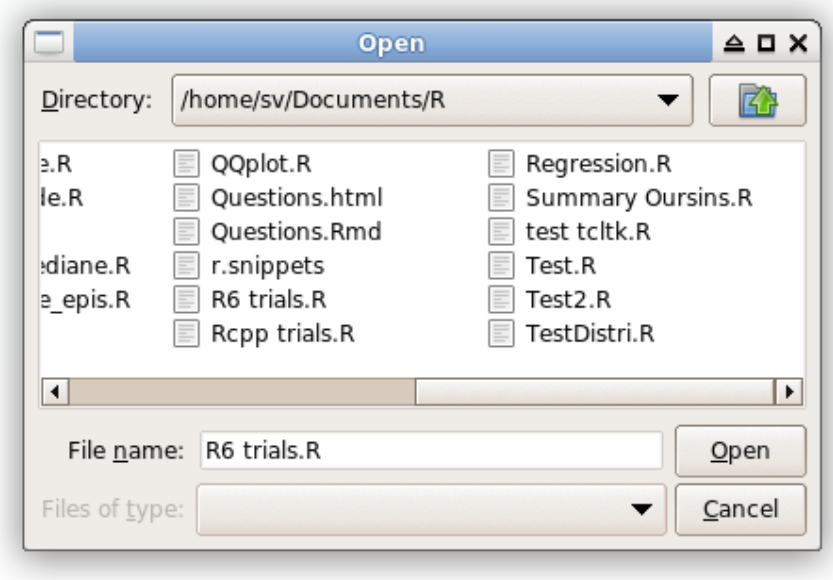


Figure 1.9: An Open file box

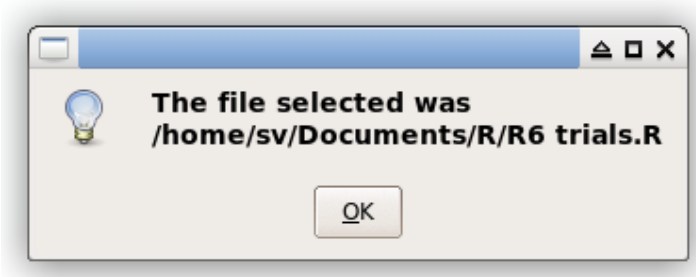


Figure 1.10: Messagebox

1.3.3.2 The Save file dialog

```
filename <- tclvalue(tkgetSaveFile())
if (!nchar(filename)) {
  tkmessageBox(message = "No file was selected!")
```

```
} else {
  tkmessageBox(message = paste("The file selected was", filename))
}
```

With this code, you get the following dialog box:

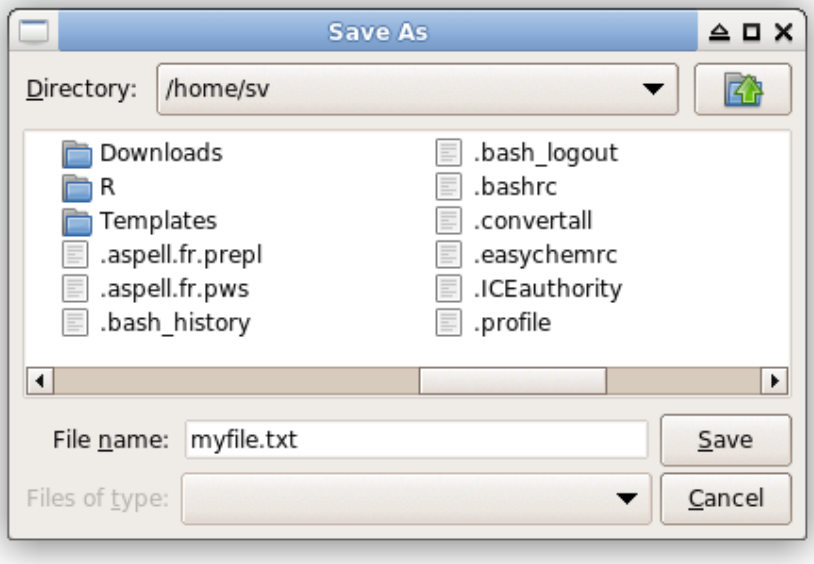


Figure 1.11: A Save file box

Now we will assume that the user pressed `Cancel`:

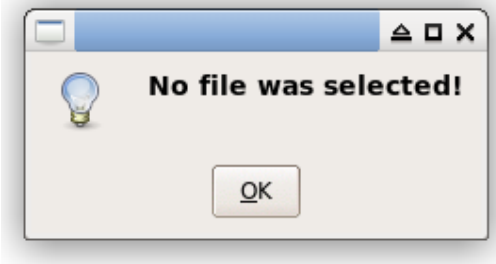


Figure 1.12: Messagebox

1.3.3.3 Opening CSV files with the open file dialog

Here is how you can specify to the **OpenFile** dialog the type of files to look for:

```
getcsv <- function() {
  name <- tclvalue(tkgetOpenFile(
    filetypes = "{ {CSV Files} {.csv} } { {All Files} * }"))
  if (name == "")
    return(data.frame()) # Return an empty data frame if no file was selected
  data <- read.csv(name)
  assign("csv_data", data, envir = .GlobalEnv)
```

```
  cat("The imported data are in csv_data\n")
}

win1 <- tktoplevel()
win1$env$butSelect <- tk2button(win1, text = "Select CSV File", command = getcsv)
tkpack(win1$env$butSelect)
# The content of the CSV file is placed in the variable 'csv_data' in the global environment
```

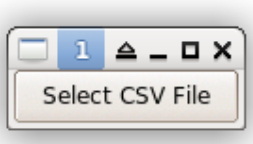


Figure 1.13: Button to open a file

Pressing the button gives the following **OpenFile** dialog, which knows which file extension to look for. In this case, only files with the extension `.csv` are displayed.

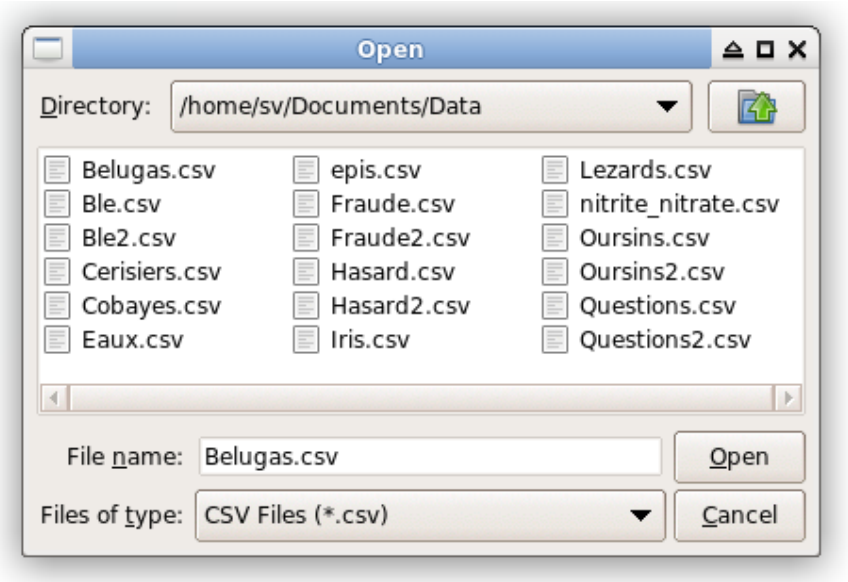


Figure 1.14: Open a csv file box

1.3.3.4 Saving (or opening) files with more than one possible extension

Multiple possibilities for file extensions (e.g., `.jpg` and `.jpeg`) can be separated by a space as follows:

```
jpeg_filename <- tclvalue(tkgetSaveFile(initialfile = "foo.jpg",
  filetypes = "{ {JPEG Files} {.jpg .jpeg} } { {All Files} * }"))
```

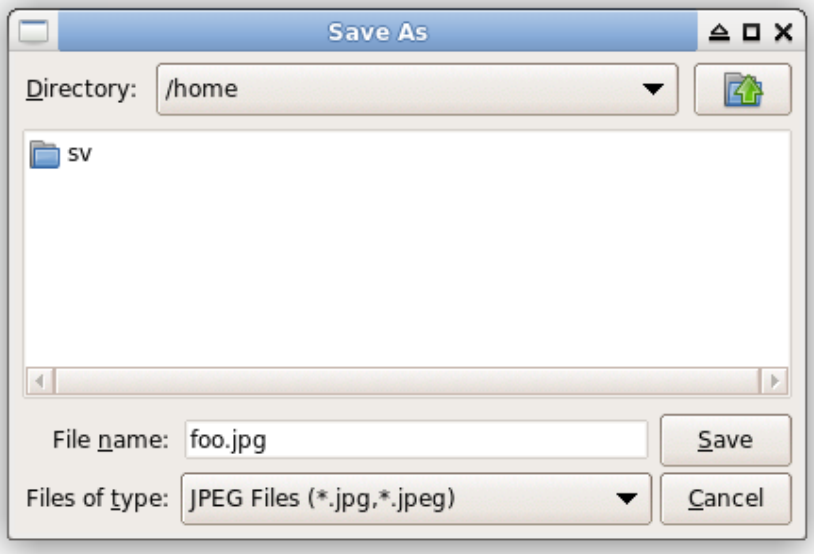


Figure 1.15: Open Jpeg file box

1.3.4 Menus in TclTk

1.3.4.1 A Simple file menu

The example below illustrates how to add a simple menu to a Tk toplevel window.

```
library(tcltk2)
win1 <- tktoplevel()
win1$env$menu <- tk2menu(win1)           # Create a menu
tkconfigure(win1, menu = win1$env$menu)  # Add it to the 'win1' window
win1$env$menuFile <- tk2menu(win1$env$menu, tearoff = FALSE)
tkadd(win1$env$menuFile, "command", label = "Quit",
  command = function() tkdestroy(win1))
tkadd(win1$env$menu, "cascade", label = "File", menu = win1$env$menuFile)
```

Running the code above gives the following window:

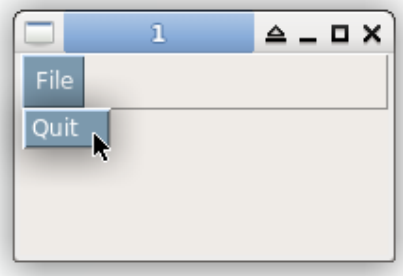


Figure 1.16: Simple menu

1.3.4.2 Cascading menus within other menus

The example below illustrates how to cascade menu within another menu.

```
win2 <- tktoplevel()
win2$env$menu <- tk2menu(win2)
tkconfigure(win2, menu = win2$env$menu)
win2$env$menuFile <- tk2menu(win2$env$menu, tearoff = FALSE)
# Our cascaded menu
win2$env$menuOpenRecent <- tk2menu(win2$env$menuFile, tearoff = FALSE)
tkadd(win2$env$menuOpenRecent, "command", label = "Recent File 1",
  command = function() tkmessageBox(
    message = "I don't know how to open Recent File 1", icon = "error"))
tkadd(win2$env$menuOpenRecent, "command", label = "Recent File 2",
  command = function() tkmessageBox(
    message = "I don't know how to open Recent File 2", icon = "error"))
tkadd(win2$env$menuFile, "cascade", label = "Open recent file",
  menu = win2$env$menuOpenRecent)
tkadd(win2$env$menuFile, "command", label = "Quit",
  command = function() tkdestroy(win2))
tkadd(win2$env$menu, "cascade", label = "File", menu = win2$env$menuFile)
```

Running the code above gives the following window:

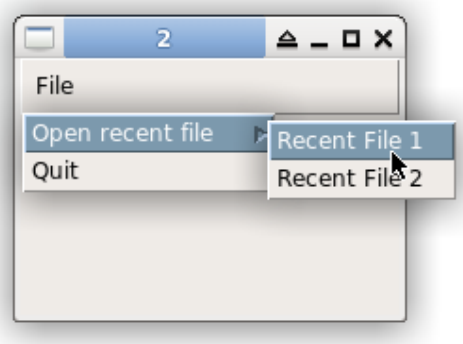


Figure 1.17: Cascaded menu

1.3.4.3 Adding a pop-up menu to a text window

The example below demonstrates how to add a simple pop-up menu to a text window. The hard part is determining the mouse coordinates in order to ensure that the menu appears where the mouse is right-clicked. Note that the keyboard shortcuts for copying and pasting (<Ctrl-C> and <Ctrl-V>) are mapped automatically for a Tk text widget.

```
win3 <- tktoplevel()
win3$env$txt <- tk2text(win3)         # Create a text widget
tkpack(win3$env$txt, fill = "both")   # And place it on 'win3'

# Create the popup menu, and its associated R function
copyText <- function()
  .Tcl(paste("event", "generate", .Tcl.args(.Tk.ID(win3$env$txt), "<<Copy>>")))
```

```
win3$env$txtPopup <- tk2menu(win3$env$txt, tearoff = FALSE)
tkadd(win3$env$txtPopup, "command", label = "Copy", command = copyText)

# The function that displays the popup menu at the right place
rightClick <- function(x, y) { # x and y are the mouse coordinates
  # tkwinfo() return several infos
  rootx <- as.integer(tkwinfo("rootx", win3$env$txt))
  rooty <- as.integer(tkwinfo("rooty", win3$env$txt))
  xTxt <- as.integer(x) + rootx
  yTxt <- as.integer(y) + rooty
  # Create a Tcl command in a character string and run it
  .Tcl(paste("tk_popup", .Tcl.args(win3$env$txtPopup, xTxt, yTxt)))
}
tkbind(win3$env$txt, "<Button-3>", rightClick) # Tcl recognizes three mouse buttons
# For mouses having two buttons, the right one is still labelled 'Button-3'!
```

Here is what you get when you run this code:

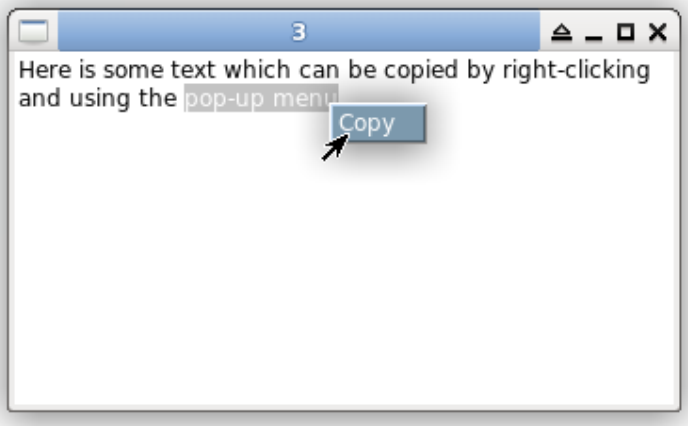


Figure 1.18: Popup menu

1.3.5 Modal dialog boxes

A modal dialog box requires the user to respond to it before changing the focus to other windows within the application. The Tk command `tk_dialog` is designed for this purpose, and can be called in R tcltk, using the `tkdialog()` function. However, the method illustrated below will use the `tktoplevel()` function and call `tkgrab.set()` and `tkgrab.release()` explicitly, rather than relying on `tkdialog()` to call them automatically.

```
library(tcltk2)

modalDialog <- function(parent, title, question, entryInit, entryWidth = 20,
returnValOnCancel = "ID_CANCEL") {
  dlg <- tktoplevel()
  tkwm.deiconify(dlg)
  tkgrab.set(dlg)
```

```
  tkfocus(dlg)
  tkwm.title(dlg, title)
  textEntryVarTcl <- tclVar(paste(entryInit))
  textEntryWidget <- tk2entry(dlg, width = paste(entryWidth),
    textvariable = textEntryVarTcl)
  tkgrid(tklabel(dlg, text = question), textEntryWidget, padx = 10, pady = 15)
  returnVal <- returnValOnCancel

  onOK <- function() {
    returnVal <<- tclvalue(textEntryVarTcl)
    tkgrab.release(dlg)
    tkdestroy(dlg)
    tkfocus(parent)
  }

  onCancel <- function() {
    returnVal <<- returnValOnCancel
    tkgrab.release(dlg)
    tkdestroy(dlg)
    tkfocus(parent)
  }

  butOK <- tk2button(dlg, text = "OK", width = -6, command = onOK)
  butCancel <- tk2button(dlg, text = "Cancel", width = -6, command = onCancel)
  tkgrid(butCancel, butOK, padx = 10, pady = c(0, 15))

  tkfocus(dlg)
  tkbind(dlg, "<Destroy>", function() {tkgrab.release(dlg); tkfocus(parent)})
  tkbind(textEntryWidget, "<Return>", onOK)
  tkwait.window(dlg)

  returnVal
}

# Create a "main" window with a button which activates our dialog
win1 <- tktoplevel()
tktitle(win1) <- "Main window"
win1$env$launchDialog <- function() {
  returnVal <- modalDialog(win1, "First Name Entry", "Enter Your First Name:", "")
  if (returnVal == "ID_CANCEL") return()
  tkmessageBox(title = "Greeting",
    message = paste0("Hello, ", returnVal, "."))
}
win1$env$butDlg <- tk2button(win1, text = "Launch Dialog",
  command = win1$env$launchDialog)
tkpack(win1$env$butDlg, padx = 60, pady = 50)
```

Clicking on the `Launch Dialog` opens our modal dialog box, i.e., you must respond to it before you can change the focus back to the main window in the Tk application.

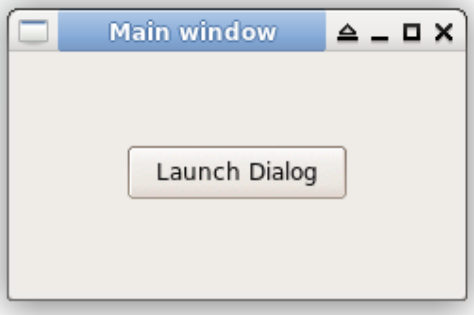


Figure 1.19: Modal button

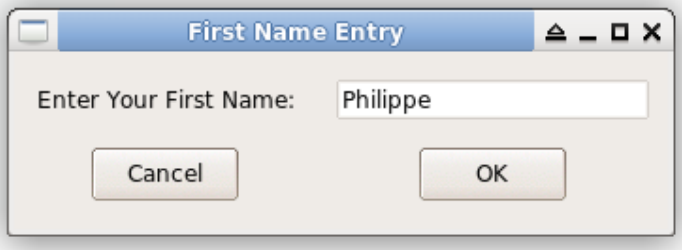


Figure 1.20: Modal dialog

Clicking `OK` gives the following message box:

Figure 1.21: Modal OK

When you have finished with this example, you can close `win1` with:

```
tkdestroy(win1)
```

1.3.6 Simple non-modal dialog box

The R code below illustrates some of the basic R TclTk functions required to create a non-modal dialog box. Non-modal means that the user is not forced to respond to this dialog box immediately. Instead the user can change the focus to another window and do something else before responding to this dialog box.

A Tcl variable `done` is used to keep track of the state of the dialog box (active, closed

with OK, or closed with Cancel/Destroyed). The `tkgrid()` function is used to layout the buttons on the window. The `tkbind()` function is used to capture the event of the window being destroyed, e.g. with the cross in the upper right-hand corner or with `Alt-F4` (in Windows) and to bind this event to a function which sets the state variable (`done`) appropriately. In order to demonstrate how to determine whether `OK` or `Cancel` was pressed, a message box is used in each case to announce the result of the dialog box.

```
library(tcltk2)

win1 <- tktoplevel()  # Create a new toplevel window
tktitle(win1) <- "Simple Dialog"  # Give the window a title

# Create a variable to keep track of the state of the dialog window:
#  If the window is active,                                          done = 0
#  If the window has been closed using the OK button,                done = 1
#  If the window has been closed using the Cancel button or destroyed, done = 2
done <- tclVar(0)   # tclVar() creates a Tcl variable

# Create two buttons and for each one, set the value of the done variable
# to an appropriate value
win1$env$butOK <- tk2button(win1, text = "OK", width = -6,
  command = function() tclvalue(done) <- 1)
win1$env$butCancel <- tk2button(win1, text = "Cancel", width = -6,
  command = function() tclvalue(done) <- 2)

# Place the two buttons on the same row in their assigned window (win1)
tkgrid(win1$env$butCancel, win1$env$butOK, padx = 20, pady = 15)

# Capture the event "Destroy" (e.g. Alt-F4 in Windows) and when this happens,
# assign 2 to done
tkbind(win1, "<Destroy>", function() tclvalue(done) <- 2)
tkfocus(win1)         # Place the focus to our Tk window

# Do not proceed with the following code until the variable done is non-zero.
# (but other processes can still run, i.e., the system is not frozen)
tkwait.variable(done)

# The variable done is now non-zero, so we would like to record its value before
# destroying the window win1.  If we destroy it first, then done will be set to 2
# because of our earlier binding, but we want to determine whether the user
# pressed OK (i.e., see whether done is equal to 1)

doneVal <- tclvalue(done)   # Get content of a Tcl variable
tkdestroy(win1)

# Test the result
switch(doneVal,
  "1" = tkmessageBox(message = "You pressed OK!"),
  "2" = tkmessageBox(message = "You either pressed Cancel or destroyed the dialog!")
)
```

Running the R code above results in the following window:

Figure 1.22: Non modal

The dialog is resizable by default, so you can easily make it big enough to see the title by dragging any of the edges or corners with the mouse. If you want the buttons to lay out nicely when de dialog is resized, you will need a little bit more work, or you should use `tkpack()` instead.

The result of pressing `OK` is shown below:

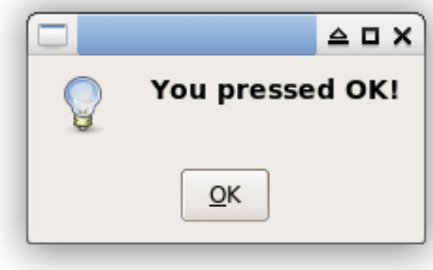


Figure 1.23: Non modal OK

The result of pressing `Cancel` is shown below:

Figure 1.24: Non modal Cancel

1.4 Basic widgets

1.4.1 A button that triggers a function call

The following R code maps the `OK` button of a Tk toplevel window to a R function which displays a message box. We give a minimum size of six characters for the button (by using a negative value for its `width`).

```
library(tcltk2)

pressedOK <- function()
  tkmessageBox(message = "You pressed OK!")

win1 <- tktoplevel()    # Create a new Tk window
win1$env$butOK <- tk2button(win1, text = "OK", width = -6, command = pressedOK)
tkgrid(win1$env$butOK, padx = 20, pady = 15)        # Place the button on the window
```

The toplevel window is shown below. We have not given it a title (using `tktitle()` or `tkwm.title()`), so the title bar displays the Tcl ID for this window, which is `1` in this case.

Figure 1.25: OK button

The result of pressing OK is shown below:

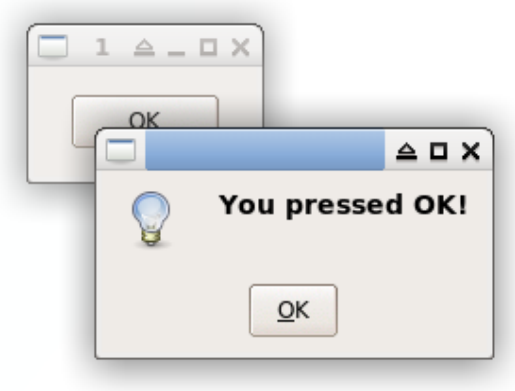


Figure 1.26: OK button pressed

To close the tk window from within R, use:

```
tkdestroy(win1)       # Kill the 'win1' Tk window
```

1.4.2 Text labels in Tk windows

Text labels can easily be added to a toplevel window using the `tklabel()` or `ttklable()` functions in **tcltk**, or the `tk2label()` function in **tcltk2**. It is not necessary to assign the result of `tklabel()` to a variable unless you want to change the text later on.

```
library(tcltk2)
win1 <- tktoplevel()
tkgrid(tk2label(win1, text = "This is a text label"))
```

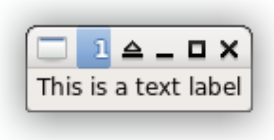


Figure 1.27: label

The following code illustrates how the text in a label can be linked to a variable:

```
win2 <- tktoplevel()
labelText <- tclVar("This is a text label")
win2$env$label <- tk2label(win2, textvariable = labelText)
tkgrid(win2$env$label)

changeText <- function()
  tclvalue(labelText) <- "This text label has changed!"
win2$env$butChange <- tk2button(win2, text = "Change text label", command = changeText)
tkgrid(win2$env$butChange)
```

Running the R code above gives the following window:

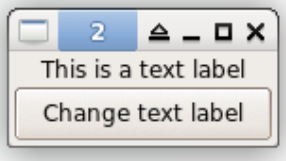


Figure 1.28: label changeable

Pressing the "Change text label" button, gives the following window:

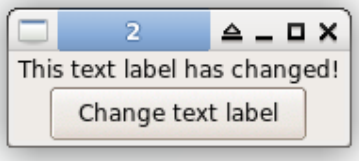


Figure 1.29: label changed

1.4.3 Checkboxes in R TclTk

The following example illustrates the use of a checkbox in a Tk toplevel window. The value of the checkbox is mapped to a Tcl variable called `cbValue`, which is initialized to zero (i.e. the checkbox will be initially unchecked). The `onOK()` function triggered by the `OK` button captures the value of the Tcl variable mapped to the checkbox (`cbValue`) before destroying the window. Then it displays an appropriate message box depending on the value of the checkbox.

```
library(tcltk2)

win1 <- tktoplevel()
win1$env$cb <- tk2checkbutton(win1, text = "I like R Tcltk")
cbValue <- tclVar("0")
tkconfigure(win1$env$cb, variable = cbValue)
tkgrid(win1$env$cb, padx = 20, pady = 15)

onOK <- function() {
  cbVal <- as.character(tclvalue(cbValue))
  tkdestroy(win1)
  switch(cbVal,
    "1" = tkmessageBox(message = "So do I!"),
    "0" = tkmessageBox(
      message = "You forgot to check the box to say that you like R TclTk!",
      icon = "warning")
  )
}
win1$env$butOK <- tk2button(win1, text = "OK", width = -6, command = onOK)
tkgrid(win1$env$butOK, padx = 10, pady = c(0, 15))

tkfocus(win1)
```

You should get a window similar to this one:

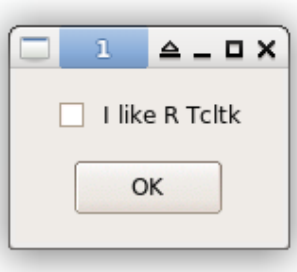


Figure 1.30: unchecked box

Click `OK` without checking the box...

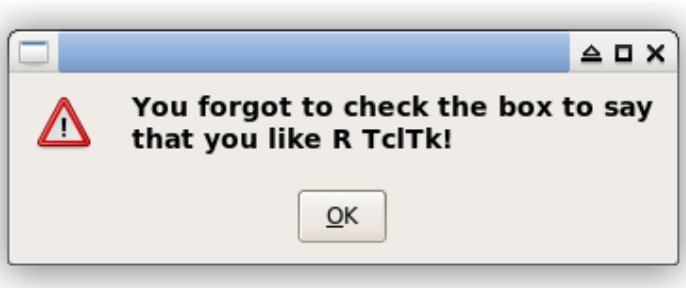


Figure 1.31: unchecked box not OK

Now, rerun the code and check the box:

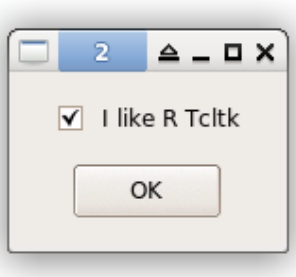


Figure 1.32: checked box

Click `OK`...

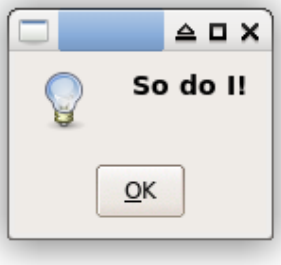


Figure 1.33: checked box OK

1.4.4 Radiobuttons in R TclTk

The following example illustrates the use of radiobuttons in a Tk toplevel window. The choice of radiobutton is mapped to a Tcl variable called `rbValue`, which is initialized to `"oranges"`, which is the value of the second radio button (i.e. initially, the second radio button will be selected). The `onOK()` function triggered by the `OK` button captures the value of the Tcl variable mapped to the radiobuttons (`rbValue`) before destroying the window. Then it displays an appropriate message box depending on the choice.

```
library(tcltk2)

win1 <- tktoplevel()
win1$env$rb1 <- tk2radiobutton(win1)
win1$env$rb2 <- tk2radiobutton(win1)
rbValue <- tclVar("oranges")
tkconfigure(win1$env$rb1, variable = rbValue, value = "apples")
tkconfigure(win1$env$rb2, variable = rbValue, value = "oranges")
tkgrid(tk2label(win1, text = "Which fruits do you prefer?"),
  columnspan = 2, padx = 10, pady = c(15, 5))
tkgrid(tk2label(win1, text = "Apples"), win1$env$rb1,
  padx = 10, pady = c(0, 5))
tkgrid(tk2label(win1, ,text = "Oranges"), win1$env$rb2,
  padx = 10, pady = c(0, 15))

onOK <- function() {
```

```
  rbVal <- as.character(tclvalue(rbValue))
  tkdestroy(win1)
  switch(rbVal,
    "apples" = tkmessageBox(
      message = "Good choice! An apple a day keeps the doctor away!"),
    "oranges" = tkmessageBox(
      message = "Good choice! Oranges are full of vitamin C!")
  )
}
win1$env$butOK <- tk2button(win1, text = "OK", width = -6, command = onOK)
tkgrid(win1$env$butOK, columnspan = 2, padx = 10, pady = c(5, 15))
tkfocus(win1)
```

You should get a window similar to this one:

Figure 1.34: radiobutton oranges

Click `OK` without changing the selection...

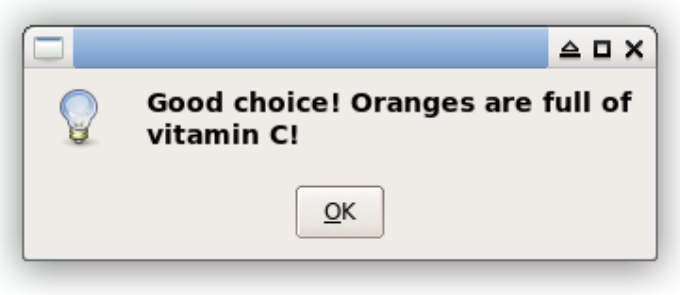


Figure 1.35: radiobutton oranges OK

Now, rerun the code and select `Apples`:

Click `OK`...

1.4.5 Edit boxes in R TclTk

The following example illustrates how to use an edit box in a Tk window. Note that the `Enter/Return` key is mapped to have the same effect as clicking the `OK` button with the

Figure 1.36: radiobutton apples

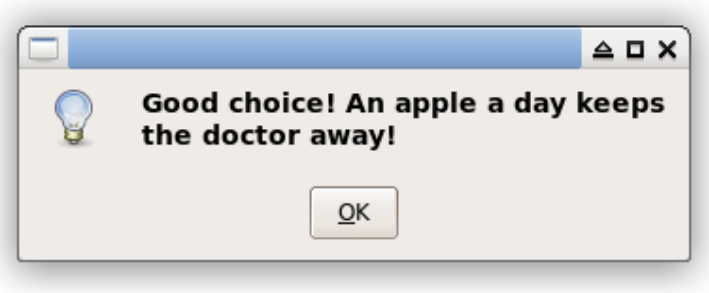


Figure 1.37: radiobutton apples OK

mouse. A common mistake is to assume that the `<Enter>` event corresponds to the `Enter` key being pressed, but this would actually mean that the user is entering data into the Tk widget (in this case the edit box). So for this example, `<Return>` is the correct event to capture.

```
library(tcltk2)

win1 <- tktoplevel()
name <- tclVar("Anonymous")
win1$env$entName <-tk2entry(win1, width = "25", textvariable = name)
tkgrid(tk2label(win1, text = "Please enter your first name:", justify = "left"),
  padx = 10, pady = c(15, 5), sticky = "w")
tkgrid(win1$env$entName, padx = 10, pady = c(0, 15))

onOK <- function() {
  nameVal <- tclvalue(name)
  tkdestroy(win1)
  msg <- paste("You have a nice name,", nameVal)
  tkmessageBox(message = msg)
}
win1$env$butOK <-tk2button(win1, text = "OK", width = -6, command = onOK)
tkgrid(win1$env$butOK, padx = 10, pady = c(5, 15))
tkbind(win1$env$entName, "<Return>", onOK)
```

```
tkfocus(win1)
```

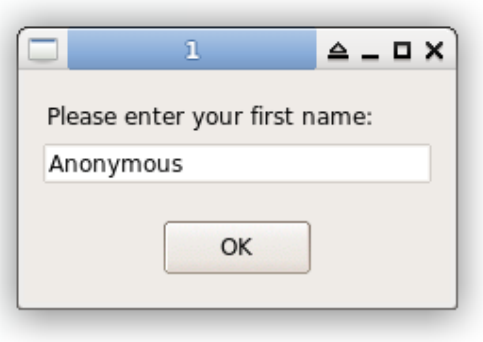


Figure 1.38: editbox

Change name and click `OK`...

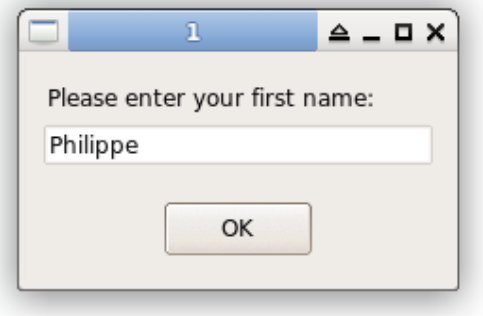


Figure 1.39: editbox edited

Figure 1.40: editbox OK

1.4.6 List boxes in R TclTk

The following examples illustrate how to use a list box in a Tk window. The first example does not have a scrollbar, so it is simpler.

1.4.6.1 List box with tk2listbox()

```
library(tcltk2)

win1 <- tktoplevel()
win1$env$lst <- tk2listbox(win1, height = 4, selectmode = "single")
tkgrid(tk2label(win1, text = "What's your favorite fruit?", justify = "left"),
  padx = 10, pady =c(15, 5), sticky = "w")
tkgrid(win1$env$lst, padx = 10, pady = c(5, 10))
fruits <- c("Apple", "Orange", "Banana", "Pear", "Apricot")
for (fruit in fruits)
  tkinsert(win1$env$lst, "end", fruit)
# Default fruit is Banana.  Indexing starts at zero.
tkselection.set(win1$env$lst, 2)

onOK <- function() {
  fruitChoice <- fruits[as.numeric(tkcurselection(win1$env$lst)) + 1]
  tkdestroy(win1)
  msg <- paste0("Good choice! ", fruitChoice, "s are delicious!")
  tkmessageBox(message = msg)
}
win1$env$butOK <-tk2button(win1, text = "OK", width = -6, command = onOK)
tkgrid(win1$env$butOK, padx = 10, pady = c(5, 15))
```

The code above produces the following window:

Figure 1.41: listbox simple

The user can then select their favorite fruit with the mouse:

The `tk2listbox()` function automatically adds a scrollbar. With the **tcltk** version (`tklistbox()`), you have to add it manually yourself and it requires much more code for the same result. The `tk2listbox()` function also eases the initial filling of the list and the preselection of items, as it will be done in the second example here under.

1.4.6.2 Prefilling of a list and deletion of items from a list

Here is a multiple selection list that is prefilled:

Figure 1.42: listbox simple selected

Figure 1.43: listbox simple OK

```
win2 <- tktoplevel()
tkgrid(tk2label(win2, text = "Please delete the fruit(s) which you don't like.",
  wraplength = 200, justify = "left"),
  padx = 10, pady = c(15, 5), sticky = "w", columnspan = 2)
# Note that 'selection' uses indices starting at 1, like R and not Tcl/Tk!
win2$env$lst <- tk2listbox(win2,
  values <- c("Apple", "Orange", "Banana", "Pear"),
  selection = 3, height = 4, selectmode = "extended")
tkgrid(win2$env$lst, padx = 10, pady = c(5, 10), sticky = "ew", columnspan = 2)

onDelete <- function() {
  fruitsSel <- as.integer(tkcurselection(win2$env$lst))
  # Warning! We have to delete elements from bottom to top, otherwise
  # as soon as we delete an element in the front of the list, the indices
  # of the remaining items are shifted!
  for (i in rev(fruitsSel))
    tkdelete(win2$env$lst, i)
}
win2$env$butDel <- tk2button(win2, text = "Delete", width = -6,
  command = onDelete)

onOK <- function() {
  fruitsRemaining <- as.character(tkget(win2$env$lst, 0, "end"))
  tkdestroy(win2)
  if (!length(fruitsRemaining)) {
    msg <- "Oh no! You don't like fruits at all, isn't it?"
  } else {
    msg <- paste0("So, you do like these fruits: ",
      paste(fruitsRemaining, collapse = ", "))
  }
  tkmessageBox(message = msg)
}
win2$env$butOK <-tk2button(win2, text = "OK ", width = -6, command = onOK)

tkgrid(win2$env$butDel, win2$env$butOK, padx = 10, pady = c(10, 15))
```

Running the code above gives the following window:

Select “Orange” from the list, then press `<Ctrl>` key and select “Pear”. You got a multiple selection:

Click `Delete` to remove these fruits from the list.

Click `OK` to get this message:

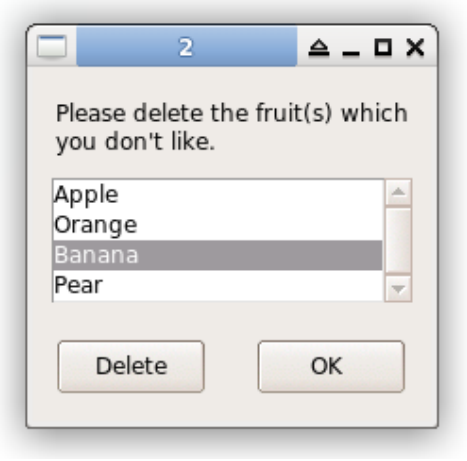


Figure 1.44: listbox delete

Figure 1.45: listbox delete sel

Figure 1.46: listbox deleted item

Figure 1.47: listbox remaining items

1.5 Additional widgets

1.5.1 Text areas (editable and non editable)

1.5.1.1 An editable text window

Here is a text area that completelly fills a Tk window.

```
library(tcltk2)

win1  <- tktoplevel()
tktitle(win1) <- "My first text widget!"
# Note that width and height are in number of characters and lines
win1$env$txt <- tk2text(win1, width = 60, height = 10)
tkpack(win1$env$txt, fill = "both", expand = TRUE)
tkfocus(win1$env$txt)

# A couple of commands to interact with the text widget:
# Add some text at the beginning of first line
tkinsert(win1$env$txt,
  "1.0", "Here is the text area...\nThis is a second line")
# Add text at the end of current one
tkinsert(win1$env$txt, "end", "\nFurther text added")
# Get the whole text
tclvalue(tkget(win1$env$txt, "0.0", "end"))
# Change the selection (select whole second line)
tktag.add(win1$env$txt, "sel", "2.0", "3.0")
# Place the cursor after the beginning of third line
# (cursor do not follow selection when it is set programmatically)
tkmark.set(win1$env$txt, "insert", "3.0")
# Get first position of the selection
tkindex(win1$env$txt, "sel.first")
# Get last poistion of the selection
tkindex(win1$env$txt, "sel.last")
# Get the range of the selection
tktag.ranges(win1$env$txt, "sel")
```

You can freely edit, cut, copy and paste in the text area.

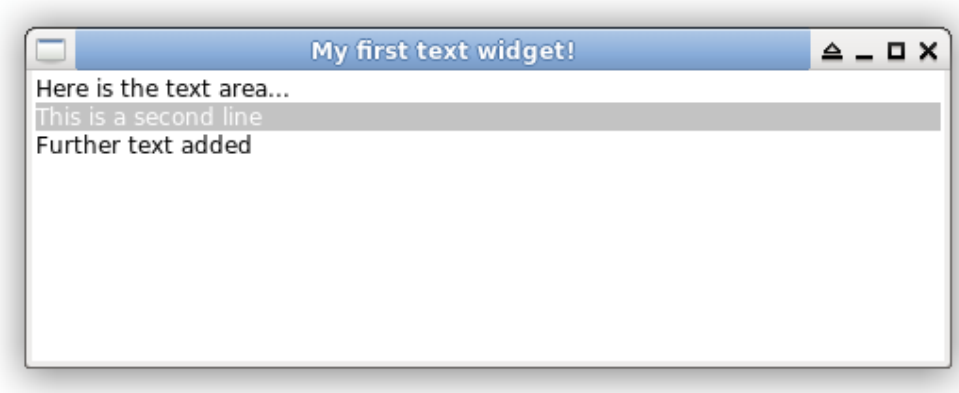


Figure 1.48: text area

1.5.1.2 A non-editable text window

```
win2  <- tktoplevel()
tktitle(win2) <- "A read-only text"
win2$env$txt <- tk2text(win2, width = 60, height = 10)
tkpack(win2$env$txt, fill = "both", expand = TRUE)
# You must insert text before to disable edition!
tkinsert(win2$env$txt, "end", "Hello, world!\n(from a read-only text widget)")
tkconfigure(win2$env$txt, state = "disabled")
tkfocus(win2$env$txt)
```

Here is what you get. Try editing the text.

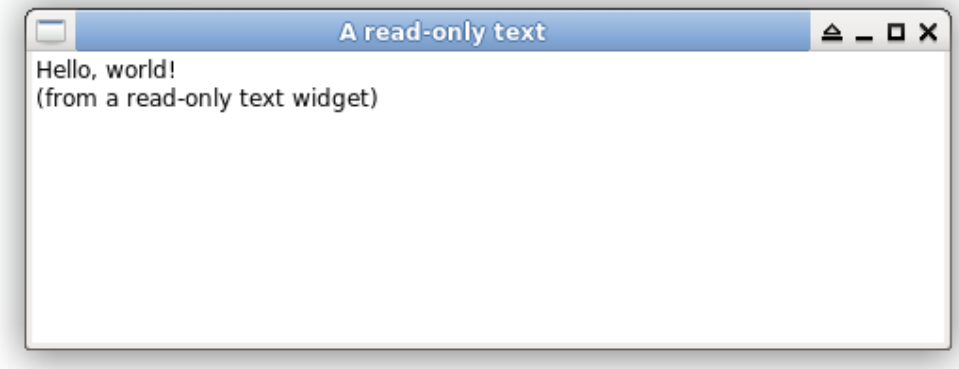


Figure 1.49: text read-only

1.5.1.3 A text window with a vertical scrollbar

```
win3  <- tktoplevel()
tktitle(win3) <- "Text area with one scrollbar"
# Scrollbar must be defined first
win3$env$scr <- tk2scrollbar(win3, orient = "vertical",
  command = function(...) tkyview(win3$env$txt, ...))
```

```
win3$env$txt <- tk2text(win3, bg = "white",
  font = "courier", width = 60, height = 10,
  yscrollcommand = function(...) tkset(win3$env$scr, ...))
# Use grid manager, telling to occupy the whole area
tkgrid(win3$env$txt, win3$env$scr, sticky = "nsew")
# Indicate that win3$env$txt must spread in x and y on window resize
tkgrid.rowconfigure(win3, win3$env$txt, weight = 1)
tkgrid.columnconfigure(win3, win3$env$txt, weight = 1)
# Populate the text area with many lines
for (i in (1:100))
  tkinsert(win3$env$txt, "end", paste0(i, "^2 = ", i*i, "\n"))
tkconfigure(win3$env$txt, state = "disabled")
tkfocus(win3$env$txt)
```

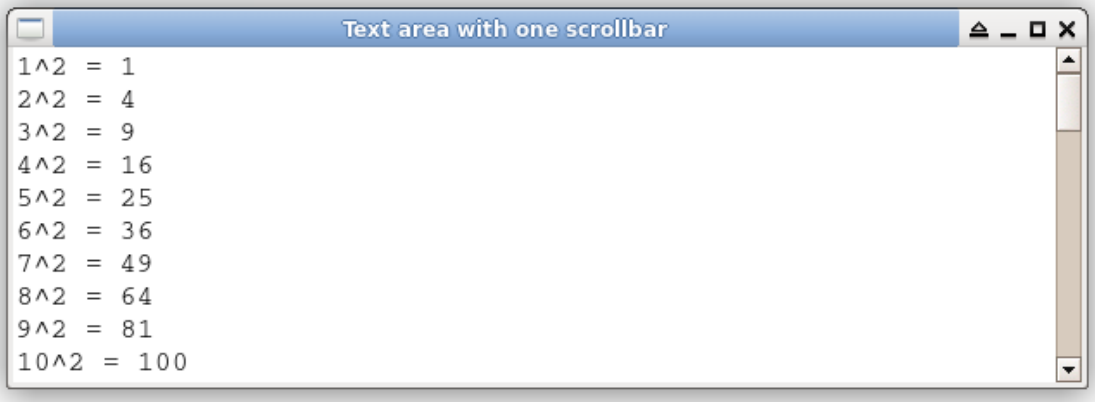


Figure 1.50: text scroll

Scrolling down reveals the remaining contents of the text widget:

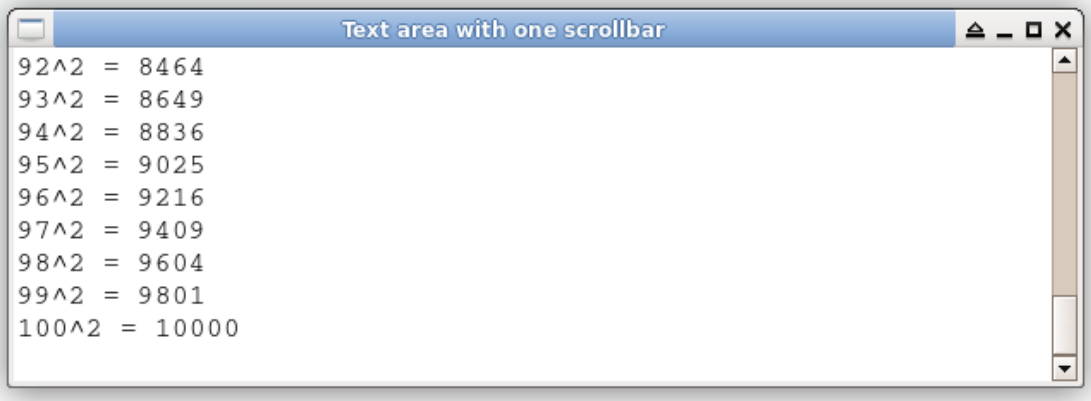


Figure 1.51: text scrolled

1.5.1.4 A text window with horizontal and vertical scrollbars (and no word wrap)

```
win4  <- tktoplevel()
tktitle(win4) <- "Text area with two scrollbars"
# Scrollbars must be defined first
win4$env$scrx <- tk2scrollbar(win4, orient = "horizontal",
  command = function(...) tkxview(win4$env$txt, ...))
win4$env$scry <- tk2scrollbar(win4, orient = "vertical",
  command = function(...) tkyview(win4$env$txt, ...))
win4$env$txt <- tk2text(win4, width = 60, height = 10, wrap = "none",
  xscrollcommand = function(...) tkset(win4$env$scrx, ...),
  yscrollcommand = function(...) tkset(win4$env$scry, ...))
tkgrid(win4$env$txt, win4$env$scry, sticky = "nsew")
tkgrid.rowconfigure(win4, win4$env$txt, weight = 1)
tkgrid.columnconfigure(win4, win4$env$txt, weight = 1)
tkgrid(win4$env$scrx, sticky = "ew")
# Populate the text area with many lines
for (i in (1:100))
  tkinsert(win4$env$txt, "end", paste0(i, "^2 = ", i*i, ", "))
tkconfigure(win4$env$txt, state = "disabled")
tkfocus(win4$env$txt)
```

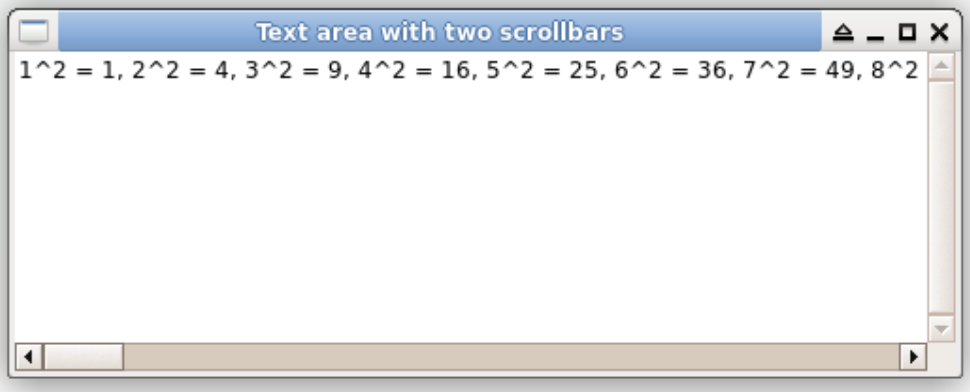


Figure 1.52: text double-scroll

Scrolling across reveals the remaining contents of the text widget:

1.5.2 Drop-down combobox

There is a `ttkcombobox()` drop-down combo box widgets in **tcltk**, and a very similar `tk2combobox()` in **tcltk2**. The following examples illustrate how to use it in a Tk window.

```
library(tcltk2)

win1 <- tktoplevel()
win1$env$combo <- tk2combobox(win1)
tkgrid(win1$env$combo, padx = 10, pady = 15)
```

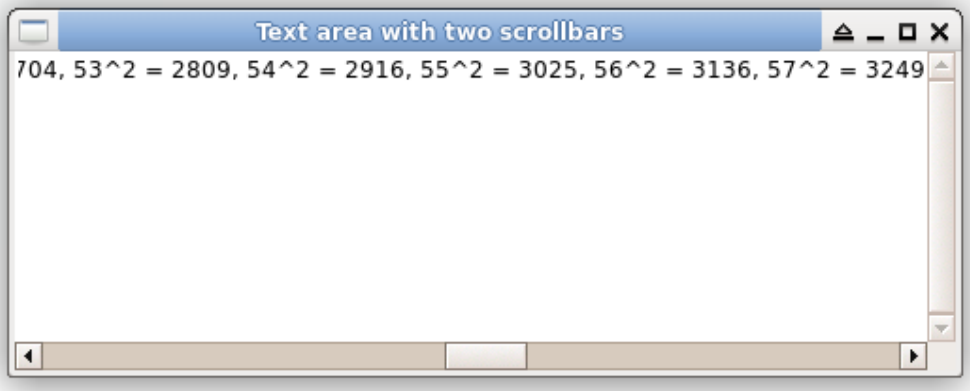


Figure 1.53: text double-scroll scrolled

```
# A couple of functions to interact with the combobox:
# Fill the combobox list
fruits <- c("Apple", "Orange", "Banana")
tk2list.set(win1$env$combo, fruits)
# Add one or more elements to the list
tk2list.insert(win1$env$combo, "end", "Apricot", "Pear")
# Delete, query, get the list
tk2list.delete(win1$env$combo, 3)   # 0-based index!
tk2list.size(win1$env$combo)
tk2list.get(win1$env$combo)   # All items
# Link current selection to a variable
fruit <- tclVar("Orange")
tkconfigure(win1$env$combo, textvariable = fruit)

# Create a button to get the content of the combobox
onOK <- function() {
  tkdestroy(win1)
  msg <- paste0("Good choice! ", tclvalue(fruit), "s are delicious!")
  tkmessageBox(title = "Fruit Choice", message = msg)
}
win1$env$butOK <- tk2button(win1, text = "OK", width = -6, command = onOK)
tkgrid(win1$env$butOK, padx = 10, pady = c(0, 15))
```

The code above produces the following window:

Change the selection to “Pear”...

... and click `OK`.

1.5.3 Frames in R TclTk

The following example illustrates how to use frames in a Tk window. Possible relief effects are `raised`, `sunken`, `flat`, `ridge`, `solid`, and `groove`. The `raised` and `sunken` effects would make the frame look like a button which is not currently being pressed (`raised`)

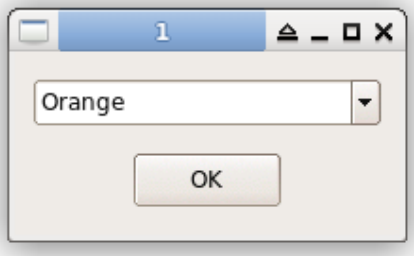


Figure 1.54: combobox

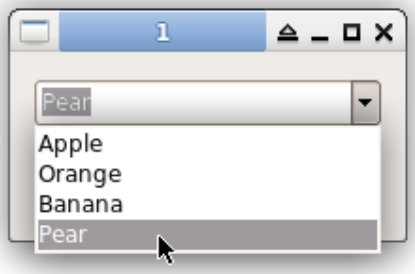


Figure 1.55: combobox selection

Figure 1.56: combobox OK

or like a button which is currently being pressed (`sunken`).

The frame creates a container inside another container. It is useful for complex widget layout, or to combine two different managers in the same Tk window. The following code first uses `tkpack()` to place areas at the to and the left of the window. Then, it uses `tkgrid()` to layout a series of widgets inside a frame.

```
library(tcltk2)

win1 <- tktoplevel()
tktitle(win1) <- "Use frames!"

# Define a frame inside 'win1'
win1$env$frm <- tk2frame(win1, borderwidth = 3, relief = "sunken",
  padding = 10)

# Pack a label at the top of the window
# Could be something like a message at the top...
# or an area for a toolbar
tkpack(tk2label(win1,
  text = "A label that is packed at the top of the window",
  width = 40, justify = "left", background = "#ffffff"),
  side = "top", expand = FALSE, ipadx = 5, ipady = 5,
  fill = "x")
# Pack a label at the bottom of the window
# Could be an area reserved for a status bar for instance
tkpack(tk2label(win1,
  text = "An area reserved at the bottom of the window",
  width = 40, justify = "left", background = "#ffffff"),
  side = "bottom", expand = FALSE, ipadx = 5, ipady = 5,
  fill = "x")
# Pack a label at the left (display a general text or image)
tkpack(tk2label(win1, text = "A label at the left of the window",
  wraplength = 50, relief = "sunken", background = "#999999"),
  side = "left", expand = FALSE,
  ipadx = 5, ipady = 5, fill = "both")
# Pack our frame in the remaining area, allowing it to expand
# (try resizing the window to see its effect)
tkpack(win1$env$frm, expand = TRUE, fill = "both")

# Now, you can populate your frame as if it was a separate
# container
# For instance, we could switch to the grid manager...
tkgrid(tk2label(win1$env$frm, text = "What is you name?"),
  columnspan = 2, padx = 10, pady = c(15, 5))
tkgrid(tk2entry(win1$env$frm),
  columnspan = 2, padx = 10, pady = c(5, 5))
tkgrid(
  # A Cancel button
  tk2button(win1$env$frm, text = "Cancel", width = -6,
```

```
    command = function() tkdestroy(win1)),
  tk2button(win1$env$frm, text = "OK", width = -6,
    command = function() tkdestroy(win1)),
  padx = 10, pady = c(5, 15))
```

The code above produces the following window:

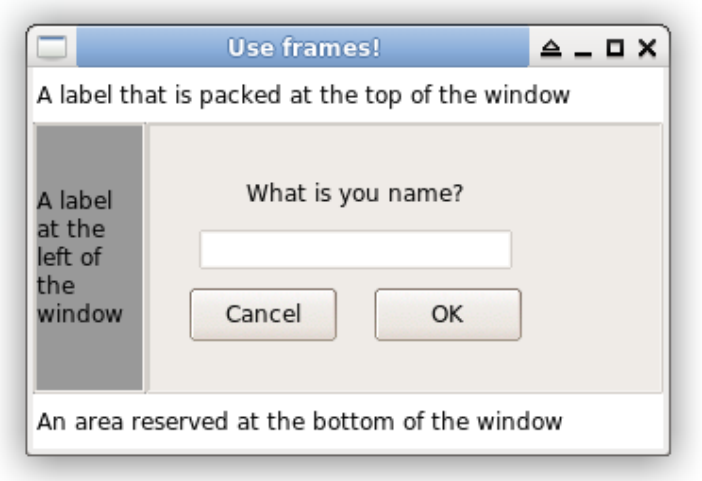


Figure 1.57: frame

Try resizing the window to see that the labels at top, bottom and left are nicely resized too. The content of the frame is **not** resized here.

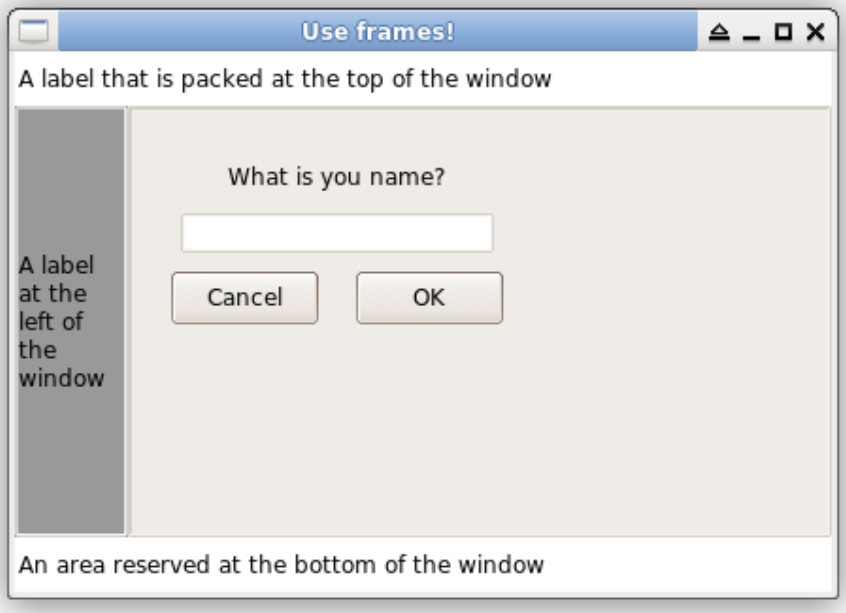


Figure 1.58: frame resized

1.5.4 Sliders in R TclTk

The following example illustrates how to use a slider in a Tk window.

```
library(tcltk2)

win1 <- tktoplevel()
tktitle(win1) <- "Slider"

# Use a linked Tcl variable to catch the value
sliderValue <- tclVar("50")

# Add a label with the current value of the slider
win1$env$label <- tk2label(win1,
  text = "Slider value: 50%")
tkgrid(win1$env$label, padx = 10, pady = c(15, 5))

# A function that changes the label
onChange <- function(...) {
  value <- as.integer(tclvalue(sliderValue))
  label <- sprintf("Slider value: %s%%", value)
  tkconfigure(win1$env$label, text = label)
}

# Add the slider
win1$env$slider <- tk2scale(win1, from = 0, to = 100,
  variable = sliderValue, orient = "horizontal", length = 200,
  command = onChange)
tkgrid(win1$env$slider, padx = 10, pady = c(5, 15))
```

The code above produces the following window:

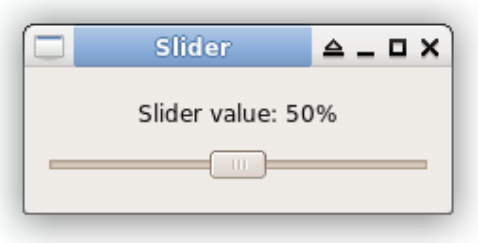


Figure 1.59: slider

Dragging the slider up to 95% gives the following:

1.5.5 Using the color-selection widget

The following code creates a toplevel widget with a Tk canvas showing the currently selected color as its background, and a button which can be used to change the color.

```
library(tcltk2)
win1 <- tktoplevel()
```

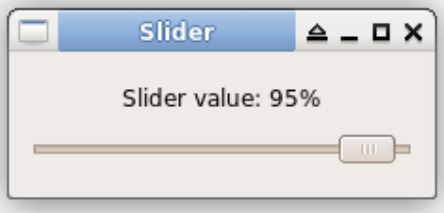


Figure 1.60: slider dragged

```
tkwm.title(win1,"Color Selection")

# Store the color name or code (#rrggbb) in a variable
color <- "blue"
win1$env$canvas <- tk2canvas(win1, width = 80, height = 25, bg = color)

# The button to call the color selector and change the color
changeColor <- function() {
  color <- tclvalue(.Tcl(paste("tk_chooseColor",
    .Tcl.args(initialcolor = color, title = "Choose a color"))))
  if (nchar(color) > 0)
    tkconfigure(win1$env$canvas, bg = color)
}
win1$env$butChange <- tk2button(win1, text = "Change Color",
  command = changeColor)

# Place both widgets sid-by-side
tkgrid(win1$env$canvas, win1$env$butChange, padx = 10, pady = 15)
```

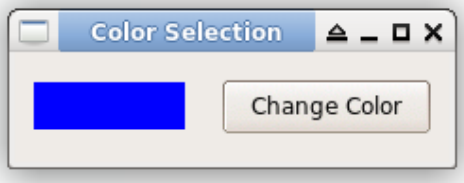


Figure 1.61: color test window

Clicking on the `Change Color` button gives the color-selection widget:

Change the color from blue to red:

Click `Ok`. Now the color is updated on our canvas:

1.5.6 Displaying an image in a Tk widget

The following code displays a GIF image in a Tk window (actually in a Tk label widget within a Tk window).

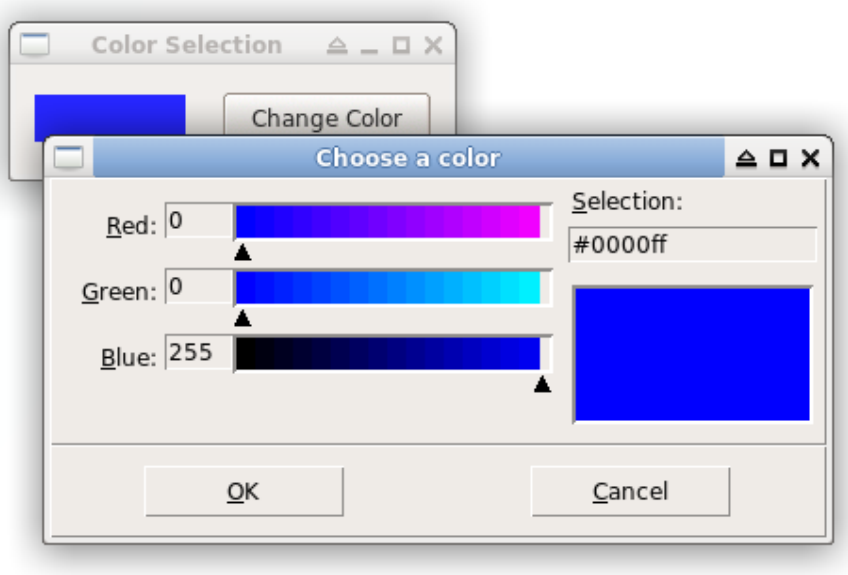


Figure 1.62: color selector

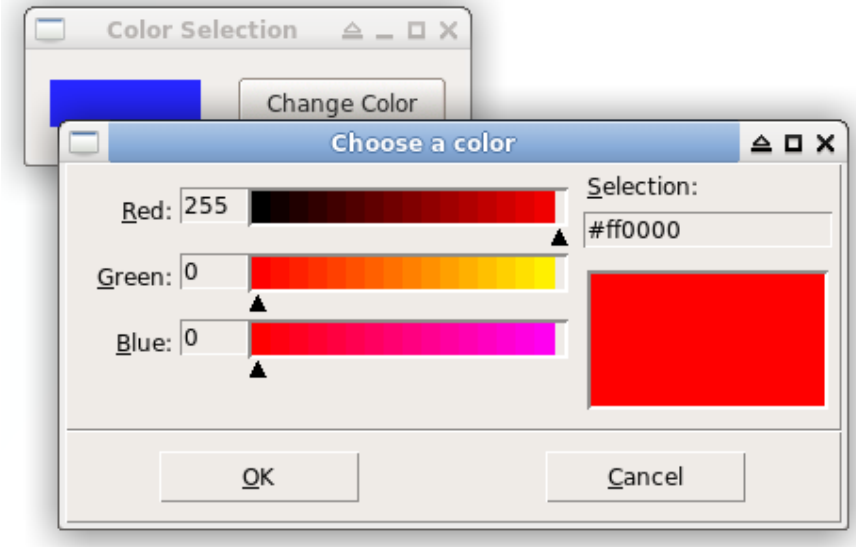


Figure 1.63: color selector after change

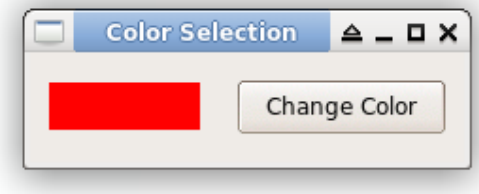


Figure 1.64: color test window updated

```
library(tcltk2)

# Tk supports natively gif images, like this one:
imgfile <- system.file("gui/SciViews.gif", package = "tcltk2")
image1 <- tclVar()
tkimage.create("photo", image1, file = imgfile)

# Create a Tk window with a label displaying this image
win1 <- tktoplevel()
win1$env$label <- tk2label(win1, image = image1)
tkpack(win1$env$label)
```

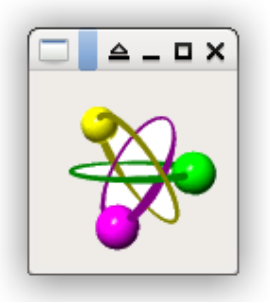


Figure 1.65: image in a Tk label

Tk supports natively PNG, GIF, PPM and PGM images. For other formats, you can install the Tk **Img** package and use `tclRequire("Img")` in R to make it available to R tcltk. From this moment on, you have also access to TIFF, JPEG, BMP, XBM, . . . , images.

1.5.7 Using the Tk table widget in R Tcltk

1.5.7.1 A short example

The TkTable widget is a very sophisticated spreadsheet-like widget which can display tables or allow the user to enter data in a tabular format. To use it, you must make sure to have the `Tktable` package installed in Tcl. Firstly, a short example using a `tclArray()`.

```
library(tcltk2)

# A simple matrix in R
mat1 <- matrix(c("Name", "James Wettenhall", "R-Help",
  "Email", "wettenhall@wehi.edu.au", "R-Help@stat.math.ethz.ch"),
  ncol = 2)

# Data must be transferred one item at a time to the tclArray object
# Also note that Tcl indexes start from 0, while they start from 1 in R
# and that without the strsplit() hack, strings with spaces are displayed
# as {string wuth spaces} in Tk Table
tclTable <- tclArray()
for (i in 1:nrow(mat1))
  for (j in 1:ncol(mat1))
```

```
    tclTable[[i-1, j-1]] <- strsplit(mat1[i, j], " ", fixed = TRUE)[[1]]

# Create a window to display this table
win1 <- tktoplevel()
win1$env$table1 <- tk2table(win1, variable = tclTable, rows = 3, cols = 2,
  titlerows = 1, selectmode = "extended", colwidth = 25, background = "white")
tkpack(win1$env$table1, fill = "both", expand = TRUE)
```

Running the R code above gives the following window:

Figure 1.66: a Tk table

1.5.7.2 A more sophisticated example

The next example demonstrates the use of another S3 object that we build to interface Tcl arrays. A `tclArrayVar` object is created using a function based on Peter Dalgaard's `tclVar()` function. An `edit()` method is defined as well as some subscripting operators. Before showing the code for the `tclArrayVar` object and methods, we will give an example of their use.

```
# Define a matrix
mat2 <- matrix(1:2000, nrow = 50, ncol = 40,
  dimnames = list(paste("Row", 1:50), paste("Col", 1:40)))

# Define a tclArrayVar and initialize it to that matrix
tclArr2 <- tclArrayVar(mat2)

# Display the Tcl array in a Tk table widget (using edit method).
# The Tcl name of the array variable is displayed in the title bar.
edit(tclArr2)

# Display the Tcl array, showing only 10 rows and 10 columns
edit(tclArr2, height = 10, width = 5)
```

```
# Change the value of one of the elements in tclArrayVar
tclArr2[2, 2] <- 999999
```

```
# Check the value of one of the elements in tclArrayVar
tclArr2[2, 2]
## [1] "999999"

tclArr2[5]
## Error in "[.tclArrayVar"(tclArr2, 5) :
##     Object is not a one-dimensional tclArrayVar
```

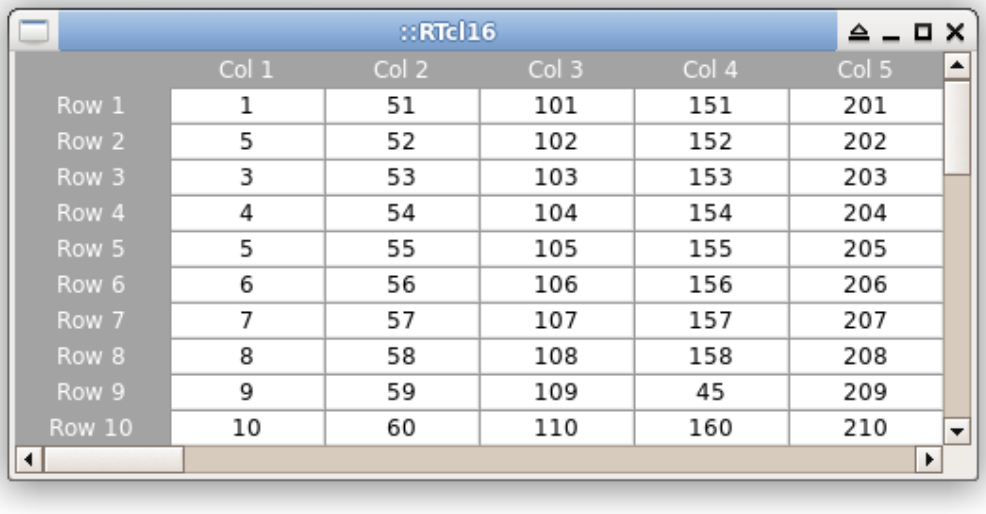


Figure 1.67: edited matrix

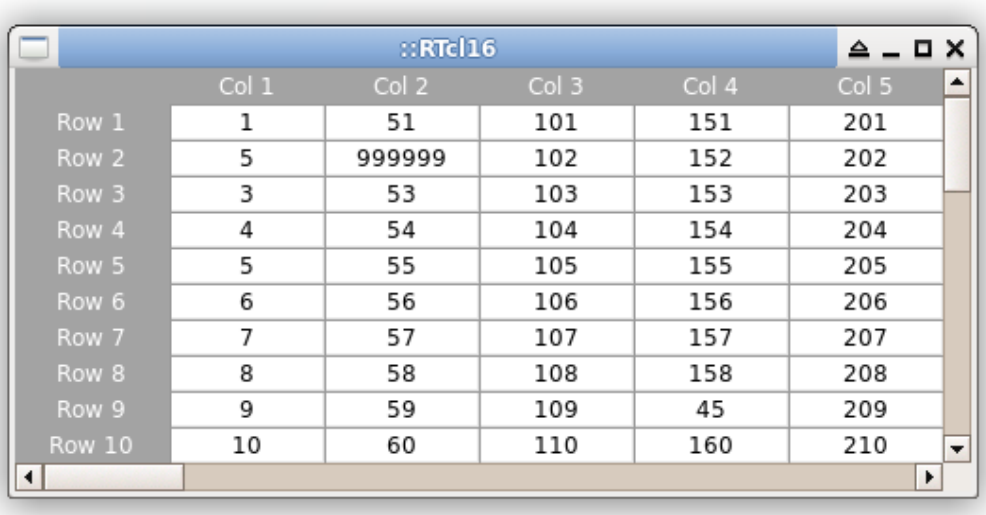


Figure 1.68: edited matrix with change value

For one-dimensional arrays (vectors):

```
# Define a vector
vec1 <- 1:100

# Define a tclArrayVar object and initialize it to that vector
tclArr3 <- tclArrayVar(vec1)

# Display the tclArrayVar object, showing only 10 rows
edit(tclArr3, height = 10)
```

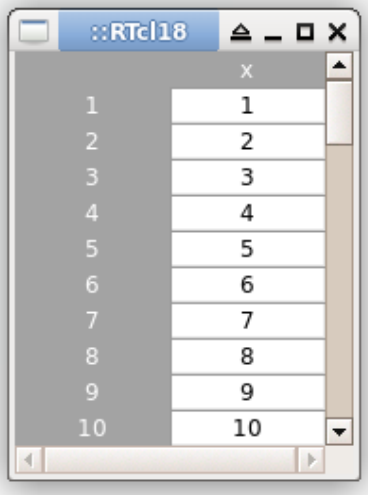


Figure 1.69: edited vector

```
# Check the value of one of the elements in the tclArrayVar object
tclArr3[5]
## [1] "5"
tclArr3[2, 3]
##Error in "[.tclArrayVar"(tclArr3, 2, 3) :
##        Object is not a two-dimensional tclArrayVar
```

Using a `tclArrayVar` object with data frames:

```
# Define a data frame
df1 <- data.frame(names = c("foo", "bar"), ages = c(20, 30))
tclArr4  <- tclArrayVar(df1)
edit(tclArr4)
```

1.5.7.2.1 Code for the tclArrayVar object

```
tclArrayVar <- function(x = NULL) {
  # Check argument
  if (!is.null(x) && !is.vector(x) && length(dim(x))!= 2)
    stop("Array must be one-dimensional or two-dimensional, or NULL.")

  library(tcltk2)
```

Figure 1.70: edited data frame

```
# Create the Tcl variable and the R Tcl object
n <- .TkRoot$env$TclVarCount <- .TkRoot$env$TclVarCount + 1
name <- paste0("::RTcl", n)
l <- list(env = new.env(), nrow = 0, ncol = 0, ndim = 0)
assign(name, NULL, envir = l$env)
reg.finalizer(l$env, function(env) tkcmd("unset", ls(env)))
class(l) <- "tclArrayVar"

# A NULL array
if (is.null(x)) {
  .Tcl(paste0("set ", name, "(0,0) \"\""))
  l$nrow <- 0
  l$ncol <- 0
  l$ndim <- 2
  return(l)
}

# A vector, matrix, or data frame
if (is.vector(x)) {
  ndim <- 1
  x <- as.data.frame(x)
} else ndim <- 2

# Populate the Tcl array
for (i in (1:nrow(x)))
  for (j in (1:ncol(x)))
    .Tcl(paste0("set ", name, "(", i, ",", j,") \"", x[i, j], "\""))

# Process dim names
if (nrow(x)) {
  if (is.null(rownames(x)))
    rownames(x) <- rep("", nrow(x))
  for (i in 1:nrow(x))
    .Tcl(paste0("set ", name, "(", i, ",", 0, ") \"",
      rownames(x)[i], "\""))
}

if (ncol(x)) {
```

```
    if (is.null(colnames(x)))
      colnames(x) <- rep("", ncol(x))
    for (j in 1:ncol(x))
      .Tcl(paste0("set ", name, "(", 0, ",", j, ") \"",
        colnames(x)[j], "\""))
  }

  l$nrow <- nrow(x)
  l$ncol <- ncol(x)
  l$ndim <- ndim
  l
}

# edit() generic function is defined in the utils package
edit.tclArrayVar <- function(name, height = 20, width = 10) {
  library(tcltk2)

  win <- tktoplevel()

  tclArrayName <- ls(name$env)
  tkwm.title(win, tclArrayName)

  table <- tk2table(win,
    rows = name$nrow + 1, cols = name$ncol + 1,
    titlerows = 1, titlecols = 1,
    maxwidth = 1000, maxheight = 1000,
    drawmode = "fast",
    height = height + 1, width = width + 1,
    xscrollcommand = function(...) tkset(xscr, ...),
    yscrollcommand = function(...) tkset(yscr,...))
  xscr <-tk2scrollbar(win, orient = "horizontal",
    command = function(...) tkxview(table, ...))
  yscr <- tk2scrollbar(win, orient = "vertical",
    command = function(...) tkyview(table, ...))

  tkgrid(table, yscr)
  tkgrid.configure(yscr, sticky = "nsw")
  tkgrid(xscr, sticky = "new")
  tkgrid.rowconfigure(win, 0, weight = 1)
  tkgrid.columnconfigure(win, 0, weight = 1)
  tkconfigure(table, variable = tclArrayName,
    background = "white", selectmode = "extended")
}

`[.tclArrayVar` <- function(object, i, j = NULL) {
  library(tcltk2)

  if (is.null(j) && object$ndim != 1)
    stop("Object is not a one-dimensional tclArrayVar")
```

```
  if (!is.null(j) && object$ndim != 2)
    stop("Object is not a two-dimensional tclArrayVar")

  if (object$ndim == 1) j <- 1
  tclArrayName <- ls(object$env)
  tclvalue(paste0(tclArrayName, "(", i, ",", j, ")"))
}

`[<-.tclArrayVar` <- function(object, i, j = NULL, value) {
  library(tcltk2)

  if (is.null(j) && object$ndim != 1)
    stop("Object is not a one-dimensional tclArrayVar")
  if (!is.null(j) && object$ndim != 2)
    stop("Object is not a two-dimensional tclArrayVar")

  if (object$ndim == 1) j <- 1
  tclArrayName <- ls(object$env)
  .Tcl(paste0("set ", tclArrayName, "(", i, ",", j, ") ", value))
  if (i > object$nrow) object$nrow <- i
  object
}
```

1.5.7.3 Additional notes

1.5.7.3.1 Copying to external spreadsheet programs

To allow copying from a table widget and pasting into a spreadsheet program such as Excel, use:

```
tkconfigure(table1, selectmode = "extended",
  rowseparator = "\"\n\"", colseparator = "\"\t\"")
```

To control whether rows and/or columns can be resized, use:

```
tkconfigure(table1, resizeborders = "none")    # OR
tkconfigure(table1, resizeborders = "both")    # OR
tkconfigure(table1, resizeborders = "row")     # OR
tkconfigure(table1, resizeborders = "col")
```

1.5.7.3.2 Line-wrapping within cells

To prevent line-wrapping within cells, use:

```
tkconfigure(table1, multiline = FALSE)
```

1.5.7.3.3 Adding/inserting rows and columns

To add a row at the end of the table, use:

```
tkinsert(table1, "rows", "end", 1)
```

To add a column at the end of the table, use:

```
tkinsert(table1, "cols", "end", 1)
```

To insert a row before the current row, use:

```
tkinsert(table1, "rows", tclvalue(tkindex(table1, "active", "row")), -1)
```

(The negative sign means insert before the current row, not after).

To insert a columnm before the current column, use:

```
tkinsert(table1, "cols", tclvalue(tkindex(table1, "active", "col")), -1)
```

1.5.7.3.4 Deleting rows and columns

To delete a row at the end of the table, use:

```
tkdelete(table1, "rows", "end", 1)
```

To delete a column at the end of the table, use:

```
tkdelete(table1, "cols", "end", 1)
```

To delete the current row, use:

```
tkdelete(table1, "rows", tclvalue(tkindex(table1, "active", "row")), 1)
```

To delete the current columnm, use:

```
tkdelete(table1, "cols", tclvalue(tkindex(table1, "active", "col")), 1)
```

1.5.8 Using the tree (drill-down) widget

Note: this has not been edited yet from the original form

The tree widget from the BWidget package is useful for displaying hierarchical information. Because it can be drilled-down or drilled-up it can be a way to save space in a user-interface while still providing the user with all of the information that they need.

In the example below, we create a simple tree with four records (nodes), each containing two child nodes ("Name" and "Age"), each of which contains one child node, (the corresponding value for its parent node).

```
library(tcltk2)
tclRequire("BWidget")

tt <- tktoplevel()
tkwm.title(tt, "Tree (Drill-Down) Widget")

xScr <- tkscrollbar(tt, command = function(...)
  tkxview(treeWidget, ...), orient = "horizontal")
yScr <- tkscrollbar(tt, command = function(...)
  tkyview(treeWidget, ...), orient = "vertical")
treeWidget <- tkwidget(tt, "Tree",
  xscrollcommand = function(...) tkset(xScr, ...),
  yscrollcommand = function(...) tkset(yScr, ...),
  width = 30, height = 15)
```

```
tkgrid(treeWidget, yScr)
tkgrid.configure(yScr, stick = "nsw")
tkgrid(xScr)
tkgrid.configure(xScr, stick = "new")

# Insert at the end of the nodes in "root" a new node, called
# "Record1Node", which displays the text "Record 1", etc.
tkinsert(treeWidget, "end", "root", "Record1Node", text = "Record 1")
tkinsert(treeWidget, "end", "root", "Record2Node", text = "Record 2")
tkinsert(treeWidget, "end", "root", "Record3Node", text = "Record 3")
tkinsert(treeWidget, "end", "root", "Record4Node", text = "Record 4")

tkinsert(treeWidget, "end", "Record1Node", "Name1Node", text = "Name")
tkinsert(treeWidget, "end", "Record2Node", "Name2Node", text = "Name")
tkinsert(treeWidget, "end", "Record3Node", "Name3Node", text = "Name")
tkinsert(treeWidget, "end", "Record4Node", "Name4Node", text = "Name")

tkinsert(treeWidget, "end", "Record1Node", "Age1Node", text = "Age")
tkinsert(treeWidget, "end", "Record2Node", "Age2Node", text = "Age")
tkinsert(treeWidget, "end", "Record3Node", "Age3Node", text = "Age")
tkinsert(treeWidget, "end", "Record4Node", "Age4Node", text = "Age")

tkinsert(treeWidget, "end", "Name1Node", "Name1Val", text = "Fred")
tkinsert(treeWidget, "end", "Name2Node", "Name2Val", text = "Jane")
tkinsert(treeWidget, "end", "Name3Node", "Name3Val", text = "Tim")
tkinsert(treeWidget, "end", "Name4Node", "Name4Val", text = "Alex")

tkinsert(treeWidget, "end", "Age1Node", "Age1Val", text = "14")
tkinsert(treeWidget, "end", "Age2Node", "Age2Val", text = "35")
tkinsert(treeWidget, "end", "Age3Node", "Age3Val", text = "63")
tkinsert(treeWidget, "end", "Age4Node", "Age4Val", text = "52")
```

The tree can be drilled down.

1.5.8.1 Determining the currently selected node

An item in the tree can be selected by clicking with the mouse. To find out which node is currently selected, use:

```
tclvalue(tkcmd(treeWidget, "selection", "get"))
## [1] "Age2Val"
```

1.5.8.2 Deleting A Node

To delete a node (and all of its children), use:

```
tkdelete(treeWidget, "Age1Node")
```

Now the age has been deleted from the first record.

1.5.8.3 Images

The tree widget can also be used to display images, by using the image option in `tkinsert()` instead of the text option (not shown).

1.5.9 The date entry and calendar widgets

1.5.9.1 The Date Entry Widget

A basic date entry widget is available in the **tcltk2** package. Here is how you can use it:

```
library(tcltk2)

win1 <- tktoplevel()

# The variable that will contain my date
mydate <- tclVar()

# Use the datefiled widget (+ a label)
tclRequire("datefield")
win1$env$date <- tkwidget(win1, "datefield::datefield",
  textvariable = mydate) #, text = "11/23/2010")
tkgrid(tk2label(win1, text = "Enter a date:"), win1$env$date,
  padx = 10, pady = 10)

# Initialize it at 11/24/2010 (MM/DD/YYYY)
tclvalue(mydate) <- "11/24/2010"

# Use an OK button to get the date
onOK <- function() {
  tkdestroy(win1)
    print(as.Date(tclvalue(mydate), format = "%m/%d/%Y"))
}
win1$env$butOK <- tk2button(win1, text = "OK", width = -6, command = onOK)
tkgrid(win1$env$butOK, columnspan = 2, pady = 15)
tkbind(win1$env$date, "<Return>", onOK)
tkfocus(win1$env$date)
```

The code above produces the following date entry widget:

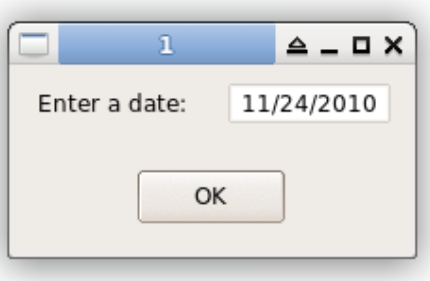


Figure 1.71: date entry widget

It does not let the user modify the slashes, only the numbers. It also ensures that the month is between 1 and 12 inclusive and that the day is between 1 and the number of days for that month, e.g. 31 for January. The date format is in `MM/DD/YYYY` (and unfortunatelly cannot be modified). Change the date:

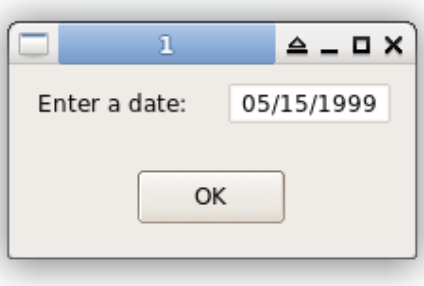


Figure 1.72: date entry widget modified

. . . then hit `<Enter>` or click the `OK` button, and you got:

```
## [1] "1999-05-15"
```

1.5.9.2 The Calendar Widget

Note: this has not been edited yet from the original form

This example was provided by Dirk Eddelbuettel.

The calendar widget displays the current date and allows the user to click on a date to select it.

```
library(tcltk2)
tclRequire("Iwidgets")
tt <- tktoplevel()
cal <- tkwidget(tt, "iwidgets::calendar")
tkconfigure(cal, command = function(...) cat(tclvalue(tkget(cal))))
tkpack(cal)
```

(This example was run on July 23rd). Clicking on July 1 gives:

```
## <Tcl>
## 07/01/2003
```

1.5.10 The tabbed notebook widget

The R code below creates a window with two tabs (with a label in the first one and a button in the second one):

```
library(tcltk2)

win1 <- tktoplevel()

# Create two tabs
win1$env$nb <- tk2notebook(win1, tabs = c("Test", "Button"))
```

```
tkpack(win1$env$nb, fill = "both", expand = TRUE)

# Populate these tabs with various widgets
win1$env$tb1 <- tk2notetab(win1$env$nb, "Test")
win1$env$lab <- tk2label(win1$env$tb1, text = "Nothing here.")
tkpack(win1$env$lab)

win1$env$tb2 <- tk2notetab(win1$env$nb, "Button")
win1$env$but <- tk2button(win1$env$tb2, text = "Click me",
  command = function() tkdestroy(win1))
# You can use a different manager than for the notebook
tkgrid(win1$env$but, padx = 50, pady = 30)

# Select a tab programmatically
tk2notetab.select(win1$env$nb, "Button")
tk2notetab.text(win1$env$nb) # Text of the currently selected tab
## [1] "Button"
```

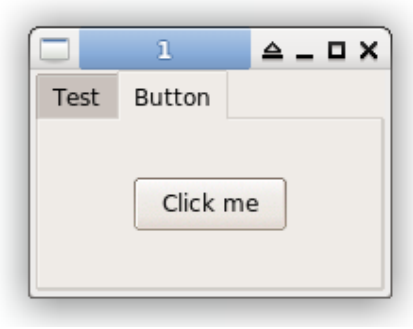


Figure 1.73: notebook with the "Button" tab selected

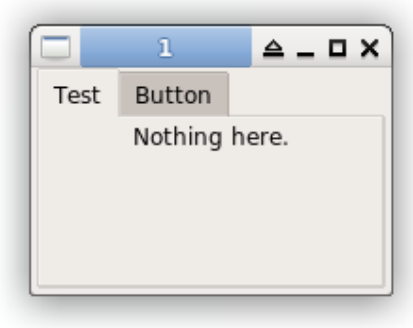


Figure 1.74: notebook with the "Test" tab selected

1.5.11 The scrollable frame

Note: this has not been edited yet from the original form

This example shows a scrollable frame (from the BWidget package). This is useful if you have an unknown number of entry widgets on a dialog and you don't know whether they will fit in a normal-sized dialog.

The BWidget package is not included in the minimal installation of Tcl/Tk which comes with R. You will have to install it separately.

```
library(tcltk2)
tclRequire("BWidget")
tt <- tktoplevel()
tkpack(tklabel(tt, text = "This is not part of the scrollable frame"))
sw <- tkwidget(tt, "ScrolledWindow", relief = "sunken", borderwidth = 2)
sf <- tkwidget(sw, "ScrollableFrame")
tkcmd(sw, "setwidget", sf)
subfID <- tclvalue(tkcmd(sf, "getframe"))
lab <- tkcmd("label", paste0(subfID, ".lab"),
  text = "This is a Scrollable Frame")
tkpack(lab)
entryList <- list()
for (i in (1:20)) {
  entryList[[i]] <- tkcmd("entry", paste(subfID, i, sep = "."), width = 50)
  tkpack(entryList[[i]], fill = "x", pady = 4)
  tkbind(entryList[[i]], "<FocusIn>",
    function() tkcmd(sf, "see", entryList[[i]]))
  tkinsert(entryList[[i]], "end", paste("Text field", i))
}
tkpack(sw, fill = "both", expand = "yes")
```

The R code above produces a window with an area that can be scrolled up and down and that contains the 20 entries.

1.6 Advanced tcltk coding

1.6.1 Layout in R TclTk

There are two main commands in Tk which are used to specify the layout of widgets on a window, `tkpack()` and `tkgrid()`. `tkgrid()` is newer and more flexible. Here is how it can be used.

With `tkgrid()`, "the grid manager", it is possible to specify absolute row and column numbers for widgets on a window, but this is not recommended, because it makes it difficult to insert new widgets later on (i.e. you would have to update all the row/column numbers). It ispossible to leave postion empty by inserting blank labels. It may be useful to sketch the grid on paper before starting coding your Tk widgets latyout.

Generally when using `tkgrid()` without specifying a row or column number, multiple arguments in the same call to `tkgrid()` are placed sequentially on the same row, and a subsequent call to `tkgrid()` will place widgets on the next row.

One very useful option in the `tkgrid()` function is the `sticky =` option. The value of this option can be an empty string, or any combination of the letters `"n"`, `"e"`, `"s"` and `"w"`, e.g. `"nsw"`. If just one letter is specified, then the widget is aligned at that edge of the grid cell (north, east, south or west). If two opposite directions are specified, e.g. `"ns"`, then the widget is stretched from the top of the cell to the bottom. If this is impossible, then it is just centered vertically, as it would be if neither `"n"` nor `"s"` were specified. If

three letters are specified, e.g. `"sew"`, then the widget is stretched in one direction (in this case horizontally - between east and west), and aligned at the bottom (south) edge of the grid cell.

```
library(tcltk2)

win1 <- tktoplevel()

tkgrid(tk2label(win1, text = "Here is a centered string of text."))
tkgrid(tk2label(win1, text = "Left"), sticky = "w")
tkgrid(tk2label(win1, text = "Right"), sticky = "e")

tkgrid(tk2label(win1, text = "    ")) # Blank line
tkgrid(tk2label(win1, text = "    ")) # Blank line

tkgrid(tklabel(win1, text =
  "Here is a much longer string of text, which takes up two columns."),
  columnspan = 2)

win1$env$labLeft <- tk2label(win1, text = "Left")
win1$env$labRight <- tk2label(win1, text = "Right")
tkgrid(win1$env$labLeft, win1$env$labRight)
tkgrid.configure(win1$env$labLeft, sticky = "w")
tkgrid.configure(win1$env$labRight, sticky = "e")

win1$env$labLeft2 <- tk2label(win1, text = "LeftAligned")
win1$env$labRight2 <- tk2label(win1, text = "RightAligned")
tkgrid(win1$env$labRight2, win1$env$labLeft2)
tkgrid.configure(win1$env$labLeft2, sticky = "w")
tkgrid.configure(win1$env$labRight2 ,sticky = "e")

tkgrid(tk2label(win1, text = "    ")) # Blank line
tkgrid(tk2label(win1, text = "    ")) # Blank line

tkgrid(tk2label(win1, text =
  "This sentence takes up two rows,\n but only one column"),
  rowspan = 2)
```

The code above produces the following window:

1.6.2 Focusing a window

The following example makes window `win1` the active window.

```
win1 <- tktoplevel() # A created windows is automatically focused
```

```
win2 <- tktoplevel() # Now the focus is on win2
```

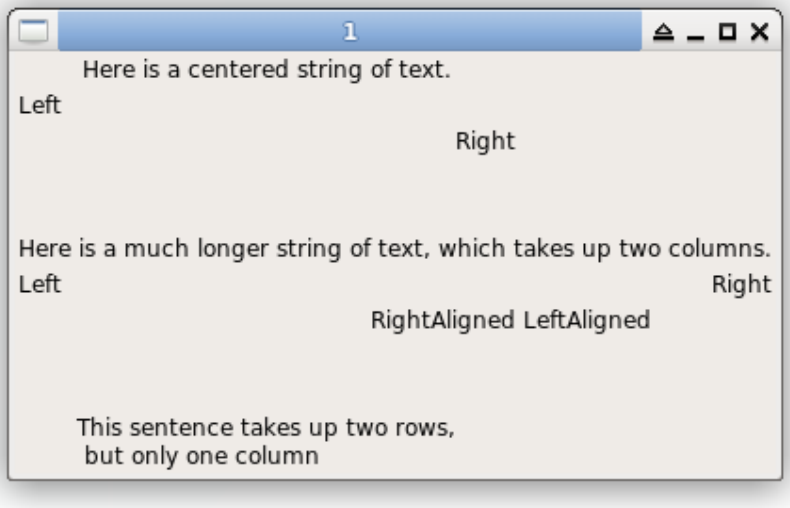


Figure 1.75: various widgets layouts

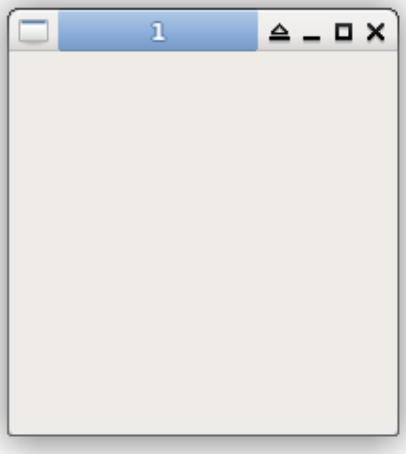


Figure 1.76: a focused Tk window

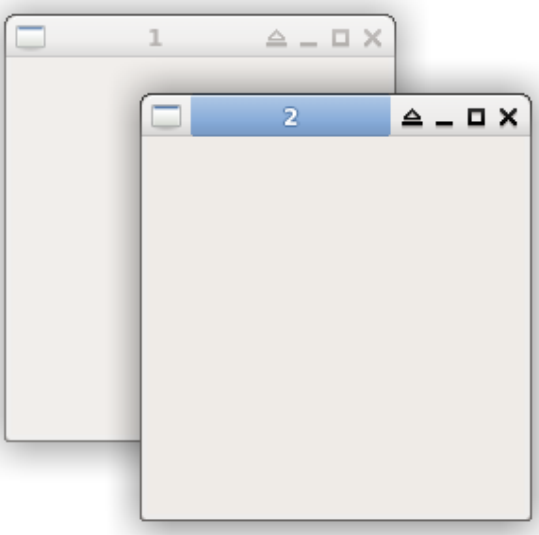


Figure 1.77: a new focused Tk window

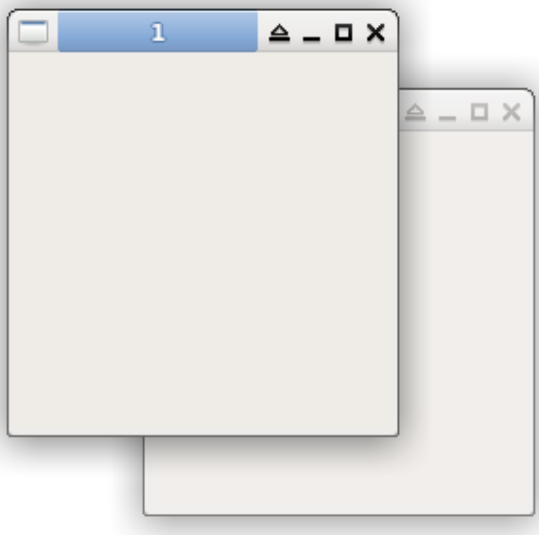


Figure 1.78: refocus on the first window

```
tkraise(win1)        # Reclaim the focus on win1
```

1.6.3 Fonts in R TclTk

The following example illustrates how to specify the font to be used in Tk windows/widgets. If you explicitly add the option `font = myFont` to every widget you create (where possible), then you only have to change `myFont` in one place, if for example you wish to use a bigger font for a presentation with a projector. You may wish to define a few different fonts - one fixed width font, one font for headings, etc.

```
library(tcltk2)

win1 <- tktoplevel()

fontHeading <- tkfont.create(family = "Arial", size = 24,
  weight = "bold", slant = "italic")
fontTextLabel <- tkfont.create(family = "Times New Roman", size = 12)
fontFixedWidth <- tkfont.create(family = "Courier New", size = 12)

tkgrid(tk2label(win1, text = "A Nice Big Font for the Heading",
  font = fontHeading), padx = 10, pady = 15)
tkgrid(tk2label(win1, text = "A normal text label.",
  font = fontTextLabel), padx = 10, pady = 5)
tkgrid(tk2label(win1, text = "A fixed width font.",
  font = fontFixedWidth, background = "white"), padx = 10, pady = c(5, 15))
```

Figure 1.79: fonts

1.6.3.1 Font selector

The **tcltk2** package proposes a font-selector dialog box:

```
fontHeading2 <- tk2chooseFont(font = fontHeading)
tkgrid(tk2label(win1, text = "A heading with the new font",
  font = fontHeading2), padx = 10, pady = 15)
```

Select a different font, then...

... click `OK`.

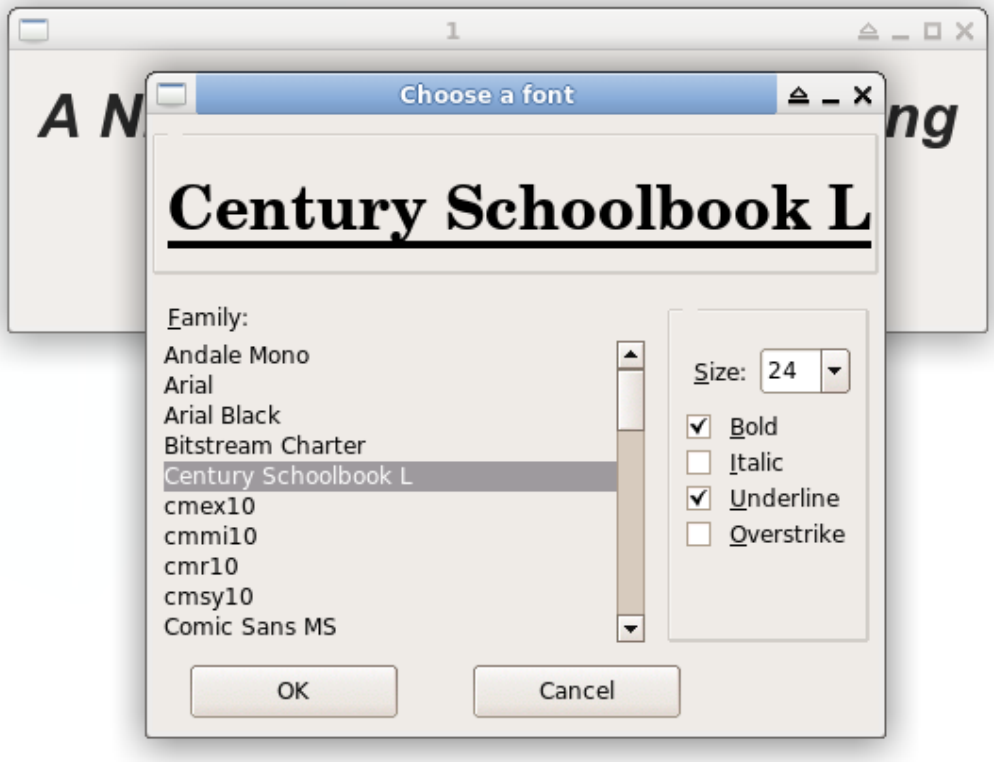


Figure 1.80: font selector

Figure 1.81: fonts updated

1.6.4 Binding Tk events

The table below lists the common events that one would want to capture or generate in a Tk window.

For examples of capturing an event, see the Edit Box example in which the event of the user pressing the `<Enter>` key is captured and mapped to a function, and see the `Dialog Box` with `OK` and `Cancel` example in which the action of destroying the window is made equivalent to pressing the `Cancel` button.

For an example of generating an event, see the `Pop-up Menu` example in which the event of copying text from a text widget into the clipboard is generated.

Event	Description
`<Button-1>`	A mouse button is pressed over the widget. Button 1 is the leftmost button, button 2
`<B1-Motion>`	The mouse is moved, with mouse button 1 being held down (use B2 for the middle
`<ButtonRelease-1>`	Button 1 was released. The current position of the mouse pointer is provided in the
`<Double-Button-1>`	Button 1 was double clicked. You can use Double or Triple as prefixes. Note that if y
`<Enter>`	The mouse pointer entered the widget (this event doesn't mean that the user presse
`<Leave>`	The mouse pointer left the widget.
`<Return>`	The user pressed the Enter key. You can bind to virtually all keys on the keyboard.
`<Key>`	The user pressed any key. The key is provided in the char member of the event obje
`a`	The user typed an "a". Most printable characters can be used as is. The exceptions a
`<Shift-Up>`	The user pressed the Up arrow, while holding the Shift key pressed. You can use pre
`<Configure>`	The widget changed size (or location, on some platforms). The new size is provided

1.6.5 Cursors in R TclTk

When one has to complete a time-consuming computation, it is nice to let the user know that the computer is busy by providing a wait cursor. The following example shows how to change the cursor, although there is no time-consuming computation in this case.

```
library(tcltk2)
win1 <- tktoplevel()
tkconfigure(win1, cursor = "watch")
```

After the computation has finished, you can change the cursor back to the normal arrow with:

```
tkconfigure(win1, cursor = "left_ptr")
```

1.6.5.1 Other cursors

The following cursors are recognized on all platforms:

```
X_cursor
arrow
based_arrow_down
based_arrow_up
boat
bogosity
```

```
bottom_left_corner
bottom_right_corner
bottom_side
bottom_tee
box_spiral
center_ptr
circle
clock
coffee_mug
cross
cross_reverse
crosshair
diamond_cross
dot
dotbox
double_arrow
draft_large
draft_small
draped_box
exchange
fleur
gobbler
gumby
hand1
hand2
heart
icon
iron_cross
left_ptr
left_side
left_tee
leftbutton
ll_angle
lr_angle
man
middlebutton
mouse
pencil
pirate
plus
question_arrow
right_ptr
right_side
right_tee
rightbutton
rtl_logo
sailboat
sb_down_arrow
sb_h_double_arrow
```

```
sb_left_arrow
sb_right_arrow
sb_up_arrow
sb_v_double_arrow
shuttle
sizing
spider
spraycan
star
target
tcross
top_left_arrow
top_left_corner
top_right_corner
top_side
top_tee
trek
ul_angle
umbrella
ur_angle
watch
xterm
```

1.6.6 Exception handling in R TclTk

Here we describe just one possible strategy for catching errors in R Tcl/Tk applications and displaying them in message boxes. If using R Tcl/Tk in Windows, one quickly notices that the RGui main window frequently "gets in the way", as R Tcl/Tk windows like to hide behind it. It therefore becomes convenient to bypass RGui altogether and just use a batch file to run Rterm with your R TclTk code. Rather than searching for errors in a `.Rout` file, it is nicer to see errors pop up as you are running the application.

1.6.6.1 Try()

We define a `Try()` function which tries to evaluate an R expression. If the expression evaluates succesfully, then the expected result is returned. If the expression causes an error, then this error is displayed in a message box.

```
Try <- function(expr) {
  res <- try(expr, silent = TRUE)
  if (inherits(res, "try-error")) {
    library(tcltk2)
    tkmessageBox(title = "An error has occured!",
      message = as.character(res), icon = "error", type = "ok")
  }
  res
}
```

1.6.6.2 Try() example

```
Try(x <- 5)
## [1] 5

Try(tkmessageBox("Hello, world!\n"))
```

Figure 1.82: error message

We got an error because we should have used:

```
tkmessageBox(message = "Hello, world!\n")
```

Note that if you want to apply `Try()` to more than one expression at once, you must enclose the expressions within braces.

1.6.6.3 A better Try()

This error shown above is useful, but we are not interested in reading the first part of the error message for every single **Tcl** error, so we will use a regular expression to track and eliminate it in a refined version of our `Try()` function for Tcl/Tk commands.

```
Try <- function(expr) {
  res <- try(expr, silent = TRUE)
  if (inherits(res, "try-error")) {
    library(tcltk2)
    res <- sub("^.+\n +\\[tcl\\] ", "Tcl error: ", res)
    tkmessageBox(title = "An error has occured!",
      message = as.character(res), icon = "error", type = "ok")
  }
  res
}
```

1.6.6.4 Try() example, take two

```
Try(tkmessageBox("Hello, world!\n"))
```

Figure 1.83: better error message

1.6.6.5 Failure to load an R package

Below is a function which can be used to try to load a package. If the package cannot be found, an error is displayed in a message box.

```
Require <- function(pkg, ...) {
  res <- try(library(pkg, character.only = TRUE, ...), silent = TRUE)
  if (inherits(res, "try-error")) {
    library(tcltk2)
    tkmessageBox(title = "An error has occured!",
      message = paste0("Cannot find package \"", pkg, "\". ",
        "May be try installing it first with install.packages(\"",
        pkg, "\")?"), icon = "error", type = "ok")
    return (FALSE)
  } else {
    return (TRUE)
  }
}
```

1.6.6.6 Require() example

```
Require("base")
## [1] TRUE
Require("aPackage")
## [1] FALSE
```

1.6.6.7 Failure to load a Tcl package

Below is a function which can be used to try to load/require a Tcl package. If the package cannot be found, an error is displayed in a message box.

```
TclRequire <- function(tclPkg) {
  library(tcltk2)
```

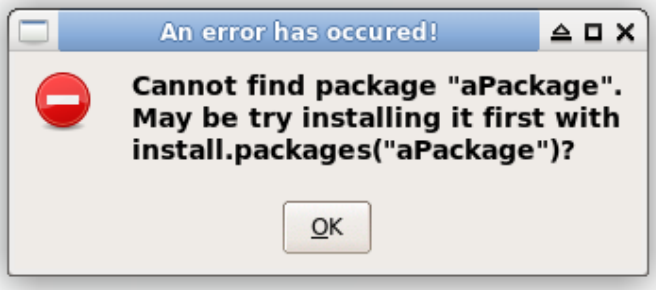


Figure 1.84: missing R package error message

```
  res <- suppressWarnings(tclRequire(tclPkg))
  if (is.logical(res) && res == FALSE) {
    tkmessageBox(title = "An error has occured!",
      message = paste0("Cannot find Tcl package \"", tclPkg,
      "\". To access Tcl/Tk extensions, you must have Tcl/Tk installed ",
      "on your computer, not just the minimal Tcl/Tk installation which ",
      "comes with R. If you do have the full Tcl/Tk installed, make sure ",
      " that R can find the path to the Tcl library, e.g. C:\\Tcl\\lib ",
      "(on Windows) or /usr/local/ActiveTcl/lib (on Linux/Unix) or ",
      "/Library/Tcl on Mac OSX. To tell R where to find the Tcl library, ",
      "use addTclPath(\"<path to Tcl library>\").\n\n",
      "If using Windows, be sure to read the R for windows FAQ at ",
      "https://cran.r-project.org/bin/windows/base/rw-FAQ.html\n\n",
      "Make sure you have the TCL_LIBRARY environment variable set to the ",
      "appropriate path, e.g., C:\\Tcl\\lib\\tcl8.6 and the MY_TCLTK ",
      "environment variable set to a non-empty string, e.g. \"Yes\"."),
      icon = "error", type = "ok")
    return (FALSE)
  } else {
    return (TRUE)
  }
}
```

1.6.7 TclRequire() example

```
TclRequire("Tk")
## [1] TRUE
TclRequire("foo")
## [1] FALSE
```

1.6.8 Evaluating R code from a scripting Tk widget

The following is taken from Peter Dalgaard's article in R News 2002, Volume 3.

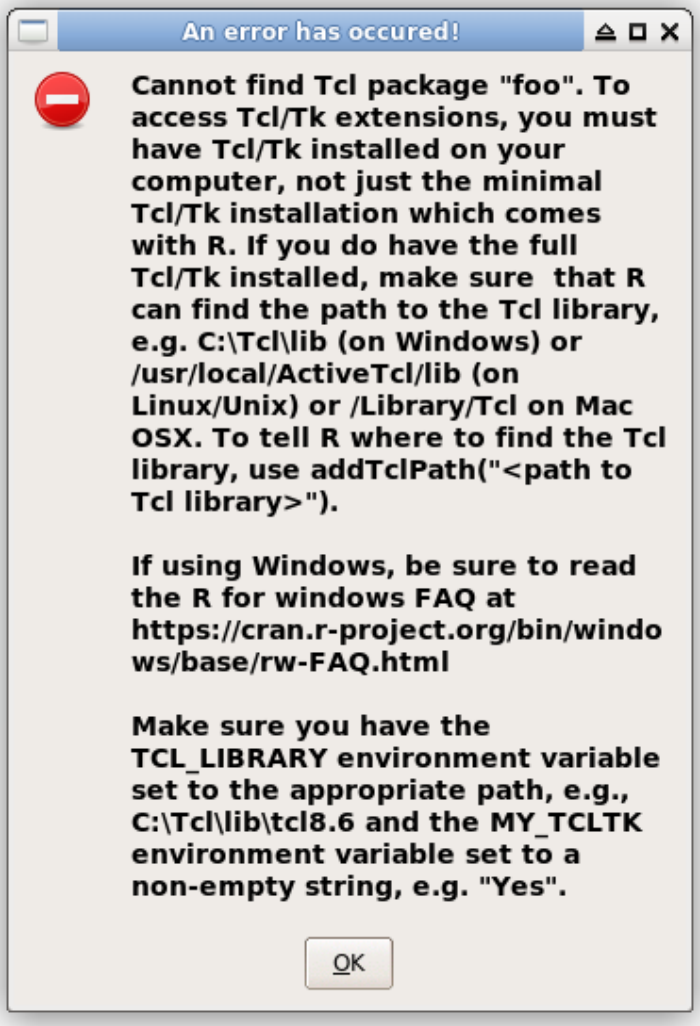


Figure 1.85: missing Tcl package error message

1.6.8.1 A scripting widget

The script-window application in Dalgaard (2001) got hit rather badly by the interface changes. Below is a version that works with the new interface. Notice that it is necessary to insert `tclvalue()` constructions in several places, even when the return values are only used as arguments to Tcl/Tk routines. You can sometimes avoid this because the default treatment of arguments (in `.Tcl.args()`) is to preprocess them with `as.character()`, but for objects of class **tclObj** this only works if there are no whitespace characters in the string representation. The contents of the script window and the files that are read can obviously contain spaces and it is also not safe to assume that file names and directory names are single words.

```
library(tcltk2)

tkscript <- function() {
  wfile <- ""

  win <- tktoplevel()
  tktitle(win) <- "R script editor"

  scrx <- tk2scrollbar(win, orient = "horizontal",
    command = function(...) tkxview(txt, ...))
  scry <- tk2scrollbar(win, orient = "vertical",
    command = function(...) tkyview(txt, ...))
  txt <- tk2text(win, width = 60, height = 10, wrap = "none",
    xscrollcommand = function(...) tkset(scrx, ...),
    yscrollcommand = function(...) tkset(scry, ...))
  tkgrid(txt, scry, sticky = "nsew")
  tkgrid.rowconfigure(win, txt, weight = 1)
  tkgrid.columnconfigure(win, txt, weight = 1)
  tkgrid(scrx, sticky = "ew")

  save <- function() {
    file <- tclvalue(tkgetSaveFile(
      initialfile = tclvalue(tclfile.tail(wfile)),
      initialdir = tclvalue(tclfile.dir(wfile))))

    if (!length(file)) return()

    chn <- tclopen(file, "w")
    on.exit(tclclose(chn))
    tclputs(chn, tclvalue(tkget(txt, "0.0", "end")))
    wfile <<- file
  }

  load <- function() {
    file <- tclvalue(tkgetOpenFile())

    if (!length(file)) return()
```

```
    chn <- tclopen(file, "r")
    on.exit(tclclose(chn))
    tkinsert(txt, "0.0", tclvalue(tclread(chn)))

    wfile <<- file
  }

  run <- function() {
    code <- tclvalue(tkget(txt, "0.0", "end"))
    e <- try(parse(text = code))

    if (inherits(e, "try-error")) {
      tkmessageBox(message = "Syntax error", icon = "error")
      return()
    }

    cat("Executing from script window:",
      "-----", code, "result:", sep = "\n")
    print(eval(e))
  }

  topMenu <- tk2menu(win)
  tkconfigure(win, menu = topMenu)
  fileMenu <- tk2menu(topMenu, tearoff = FALSE)
  tkadd(fileMenu, "command", label = "Load", command = load)
  tkadd(fileMenu, "command", label = "Save", command = save)
  tkadd(topMenu, "cascade", label = "File", menu = fileMenu)
  tkadd(topMenu, "command", label = "Run", command = run)
}

tkscript()
```

The scripting widget is shown below with an example of some trivial R code:

The results are displayed in the R console:

```
## Executing from script window:
## -----
## mean(1:6)
##
## result:
## [1] 3.5
```

Of course, it would also be possible to display the results in another Tk text window, rather than in the R console.

1.6.9 Plotting graphs with tkrplot

The following example shows how to plot a graph in a Tk window, using the **tkrplot** package.

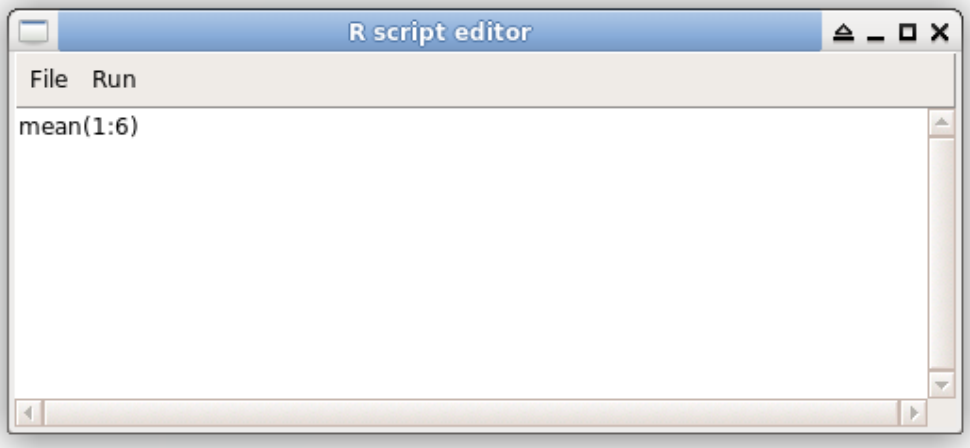


Figure 1.86: script window

```
library(tcltk2)
library(tkrplot)

hscale <- 1.5    # Horizontal scaling
vscale <- 1.5    # Vertical scaling

plotTk <- function() {
  x <- -100:100
  y <- x^2
  plot(x, y, main = "A parabola")
}

win1 <- tktoplevel()
tktitle(win1) <- "A parabola"

win1$env$plot <- tkrplot(win1, fun = plotTk,
  hscale = hscale, vscale = vscale)
tkgrid(win1$env$plot)
```

The code listed above gives the following graph window:

It is worth noting that **tkrplot** places the graph on the clipboard (in Windows or X11) before it plots it on the window. This means that once a graph has been plotted, it can easily be pasted into another program. However, if a second graph is plotted, the first graph will be lost from the clipboard, so the software developer may wish to include a `Copy to Clipboard` button or menu item on the **tkrplot** graph window, so that the user can come back to it later and copy it to the clipboard. This can be done using the `tkrreplot()` function as follows:

```
library(tcltk2)
library(tkrplot)

hscale <- 1.5    # Horizontal scaling
```

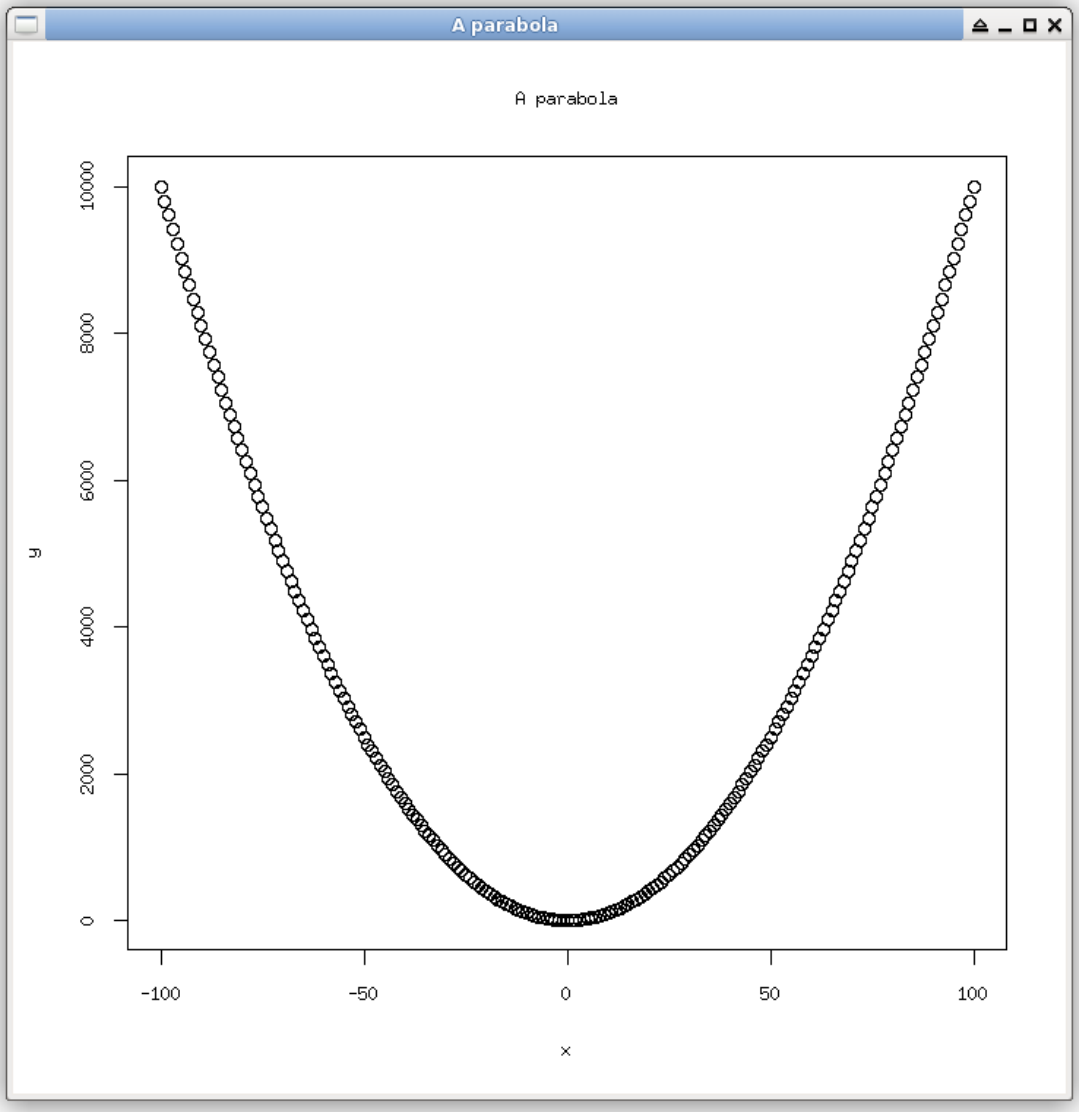


Figure 1.87: R plot in a Tk window

```
vscale <- 1.5    # Vertical scaling

plotTk <- function() {
  x <- -100:100
  y <- x^2
  plot(x, y, main = "A parabola")
}

win2 <- tktoplevel()
tktitle(win2) <- "A parabola"

win2$env$plot <- tkrplot(win2, fun = plotTk,
  hscale = hscale, vscale = vscale)

copyToClipboard <- function() tkrreplot(win2$env$plot)
win2$env$butCopy <- tk2button(win2, text = "Copy to Clipboard",
  command = copyToClipboard)
tkgrid(win2$env$butCopy, padx = 10, pady = 5, sticky = "nw")
tkgrid(win2$env$plot)
```

The code listed above gives the following graph window:

1.6.10 Interactive plots with tkrplot

The R code for this example is a little longer than that of the simpler examples. The basic idea is that we put a scatter plot in a Tk window, using the **tkrplot** package by Luke Tierney. We then allow the user to click on (or near) one of the plotted points in order to attach a label to that point (and replot the graph).

The hardest part is mapping between image (Tk widget) coordinates and R plot coordinates, which is done in the function `onLeftClick()`.

Running the code below gives the following graph window:

Clicking on the upper-left point gives the following message box:

After answering `Yes` to the message box question, the graph is updated. Note the label, `A` above the point that was clicked on.

After labeling all of the points, the graph looks like this:

1.6.10.1 R code for interactive tkrplot example

```
library(tcltk2)
library(tkrplot)

xCoords <- -12:13
yCoords <- xCoords^2
labelsVec <- LETTERS

indexLabeled <- c()
```

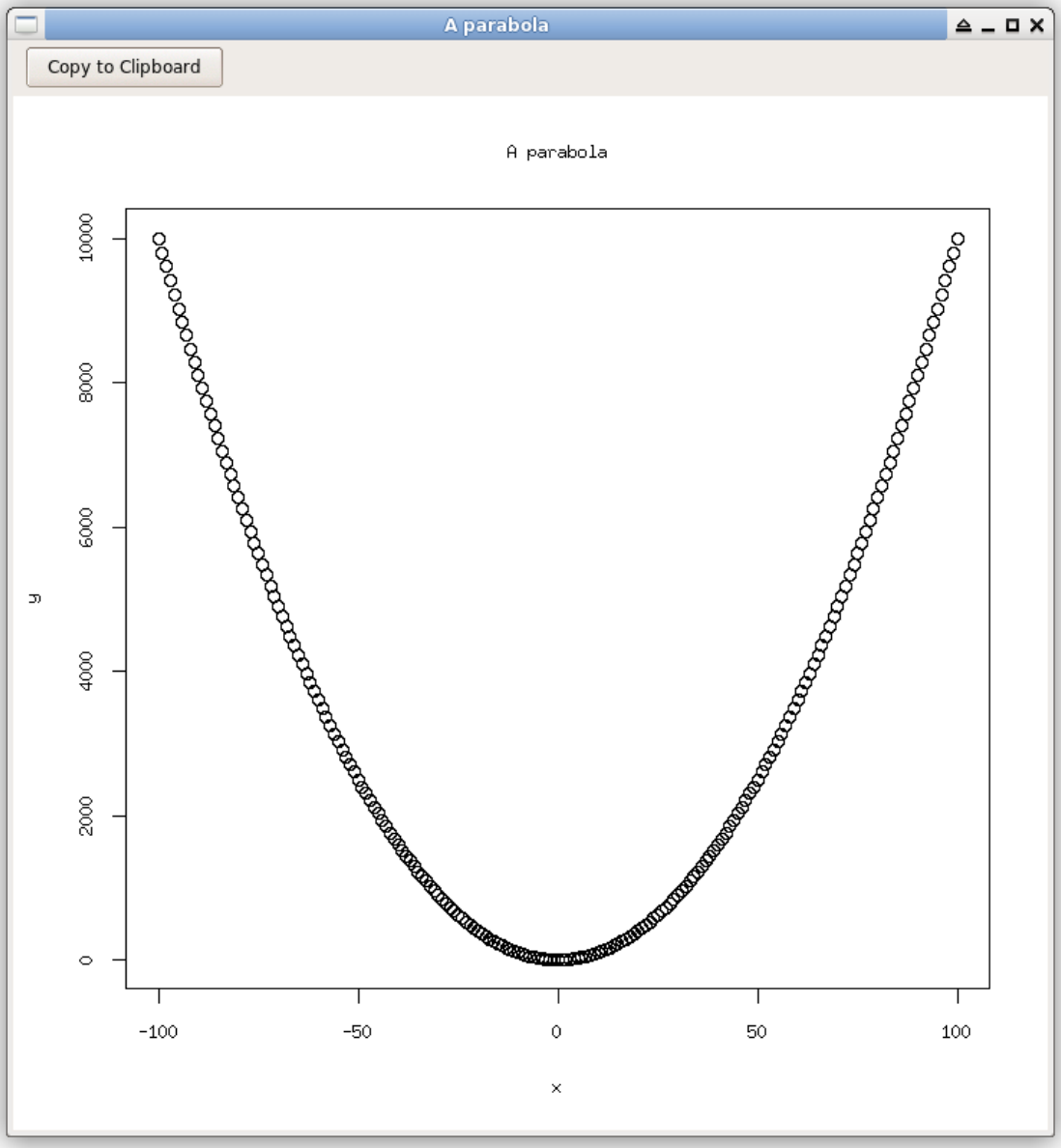


Figure 1.88: R plot with copy button

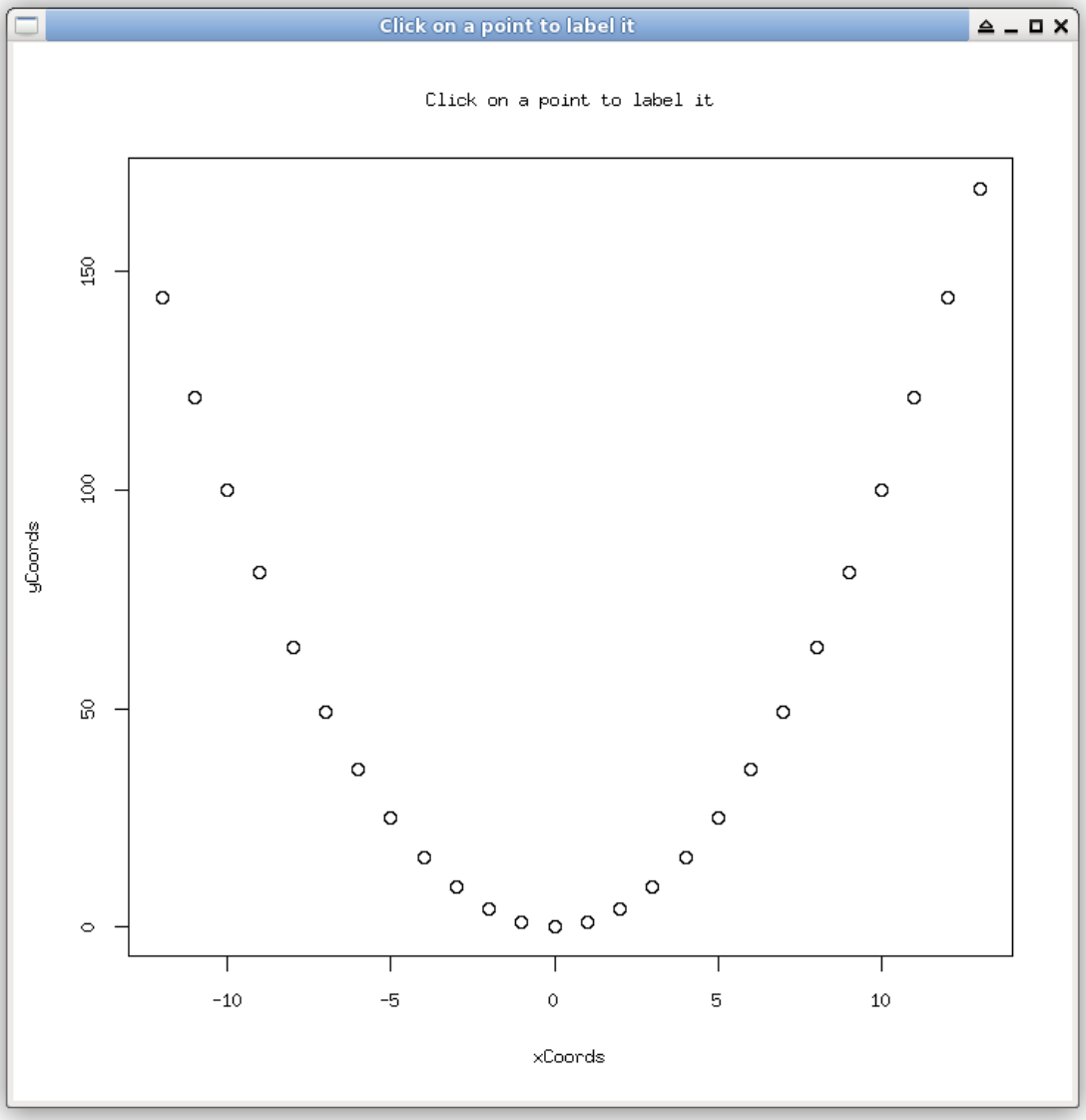


Figure 1.89: R plot

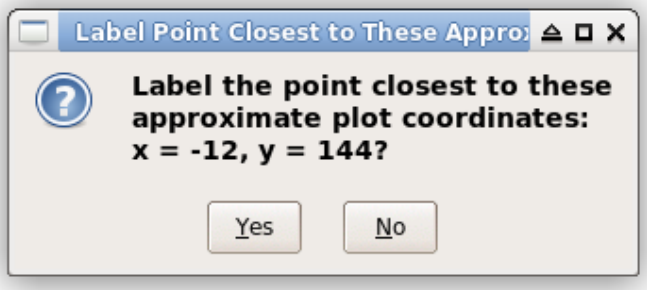


Figure 1.90: Question

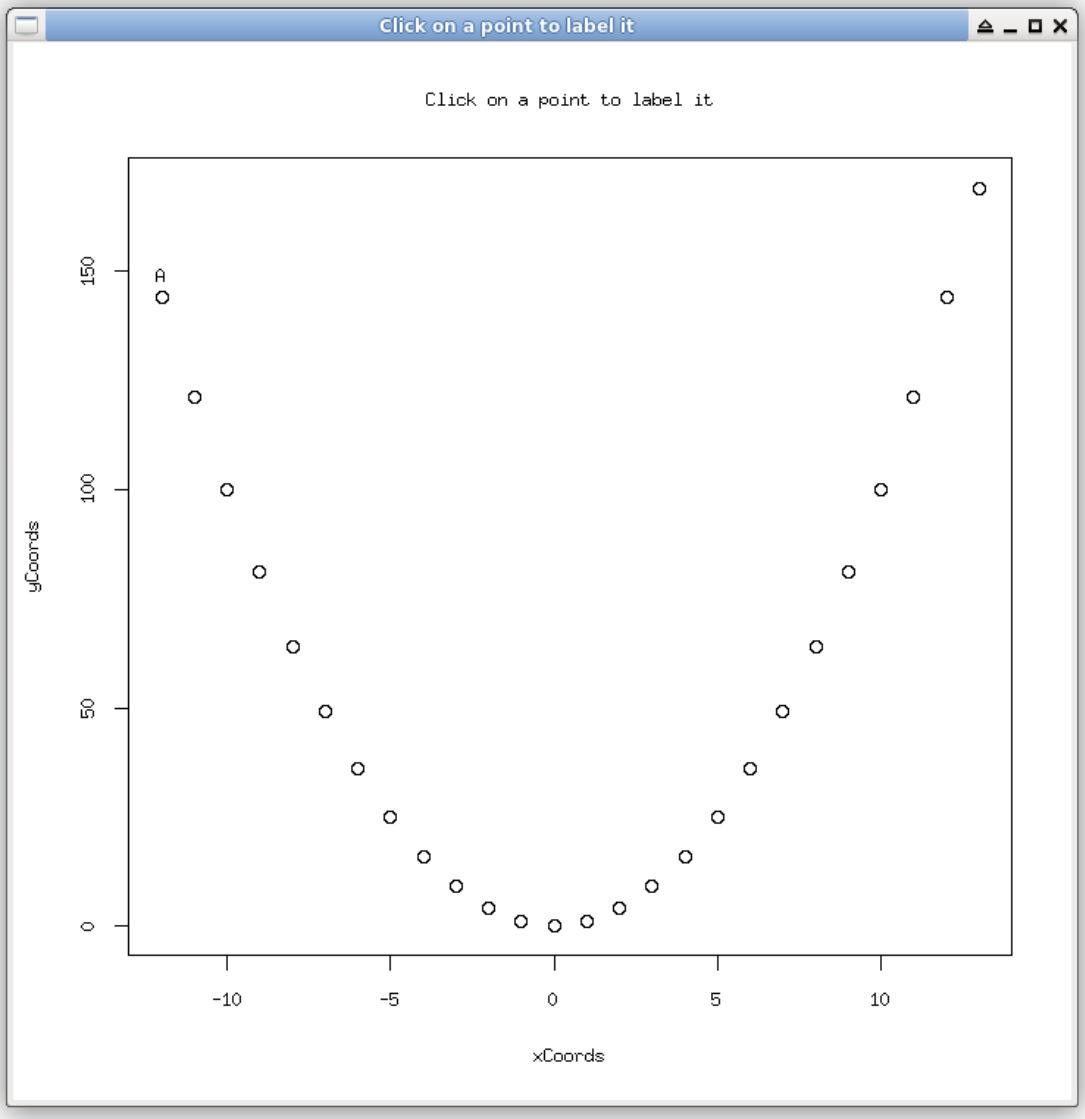


Figure 1.91: R plot with one point labelled

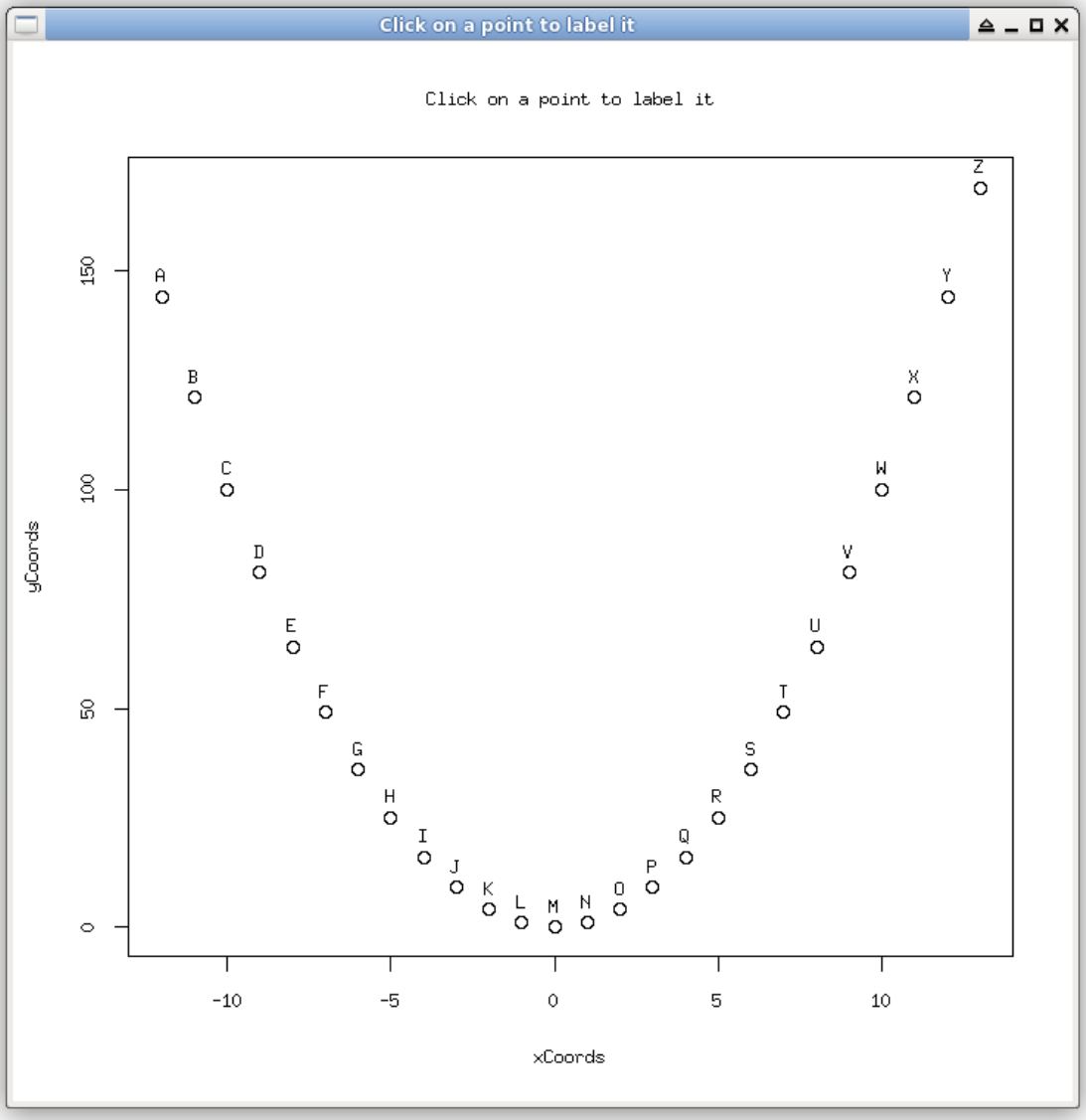


Figure 1.92: R plot with all points labelled

```
labeledPoints <- list()

win1 <- tktoplevel()
tktitle(win1) <- "Click on a point to label it"

parPlotSize <- c()
usrCoords <- c()

plotTk <- function() {
  plot(xCoords, yCoords, main = "Click on a point to label it")
  if (length(indexLabeled)) {
    for (i in (1:length(indexLabeled))) {
      indexClosest <- indexLabeled[i]
      text(xCoords[indexClosest], yCoords[indexClosest],
        labels = labelsVec[indexClosest], pos = 3)
    }
  }
  parPlotSize <<- par("plt")
  usrCoords   <<- par("usr")
}

win1$env$plot <- tkrplot(win1, fun = plotTk, hscale = 1.5, vscale = 1.5)
tkgrid(win1$env$plot)

labelClosestPoint <- function(xClick, yClick, imgXcoords, imgYcoords) {
  squaredDistance <- (xClick - imgXcoords)^2 + (yClick - imgYcoords)^2
  indexClosest <- which.min(squaredDistance)
  indexLabeled <<- c(indexLabeled, indexClosest)
  tkrreplot(win1$env$plot)
}

onLeftClick <- function(x, y) {
  xClick <- x
  yClick <- y
  width  <- as.numeric(tclvalue(tkwinfo("reqwidth", win1$env$plot)))
  height <- as.numeric(tclvalue(tkwinfo("reqheight", win1$env$plot)))

  xMin <- parPlotSize[1] * width
  xMax <- parPlotSize[2] * width
  yMin <- parPlotSize[3] * height
  yMax <- parPlotSize[4] * height

  rangeX <- usrCoords[2] - usrCoords[1]
  rangeY <- usrCoords[4] - usrCoords[3]

  imgXcoords <- (xCoords - usrCoords[1]) * (xMax - xMin) / rangeX + xMin
  imgYcoords <- (yCoords - usrCoords[3]) * (yMax - yMin) / rangeY + yMin

  xClick <- as.numeric(xClick) + 0.5
```

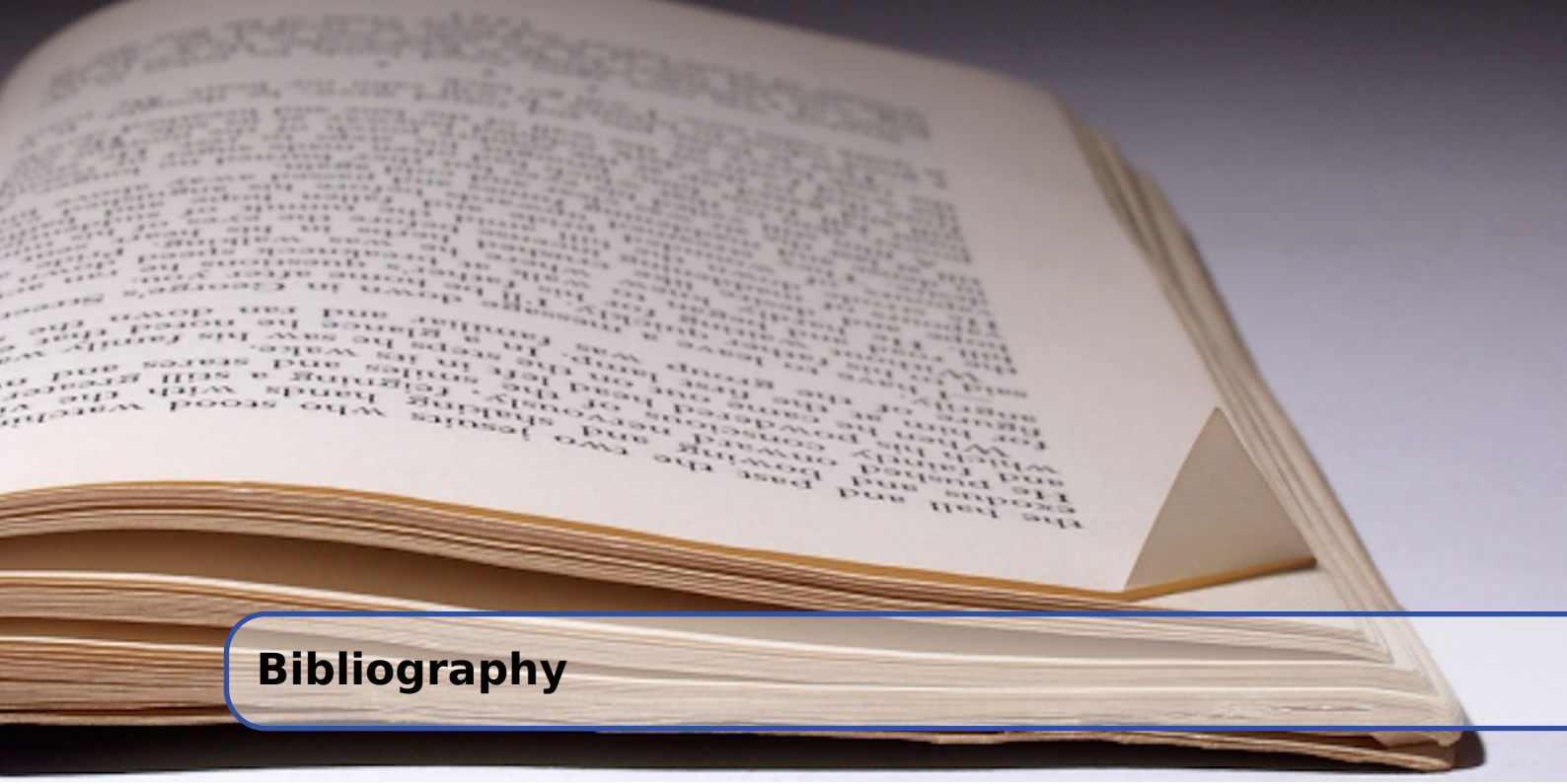
```
  yClick <- as.numeric(yClick) + 0.5
  yClick <- height - yClick

  xPlotCoord <- usrCoords[1] + (xClick - xMin) * rangeX / (xMax - xMin)
  yPlotCoord <- usrCoords[3] + (yClick - yMin)* rangeY / (yMax - yMin)

  msg <- paste0("Label the point closest to these ",
    "approximate plot coordinates: \n",
    "x = ", format(xPlotCoord, digits = 2),
    ", y = ", format(yPlotCoord, digits = 2), "?")
  mbval <- tkmessageBox(title =
    "Label Point Closest to These Approximate Plot Coordinates",
    message = msg, type = "yesno", icon = "question")

  if (tclvalue(mbval)== "yes")
    labelClosestPoint(xClick, yClick, imgXcoords, imgYcoords)
}

tkbind(win1$env$plot, "<Button-1>", onLeftClick)
tkconfigure(win1$env$plot, cursor = "hand2")
```



Bibliography

Dalgaard, P. (2001). A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31.

Dalgaard, P. (2002). Changes to the R-Tcl/Tk package. *R News*, 2(3):25–27.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.