John Fox

Writing R Commander Plug-in Packages

Version 3.3-1

2017-01-04

Contents

1

Introduction

As its title implies, *Using the R Commander* (Fox, 2017) is written from the point of view of the *user* of the R Commander graphical user interface (GUI) to R. The book includes a brief introduction to the *use* of R Commander plug-in packages, employing the **RcmdrPlugin.TeachingDemos** and **RcmdrPlugin.survival** packages as examples. In contrast, the current document aims to be a comprehensive manual for R Commander plug-in package *authors*.

Support for plug-in packages has been incorporated in the R Commander for about 10 years, but previous descriptions of how to write R Commander plug-ins, in Fox (2007) and Fox and Carvalho (2012), are relatively brief, incomplete, and out-of-date. Despite these deficiencies, there are currently about 40 R Commander plug-in packages on CRAN (the Comprehensive R Archive Network). I hope that this manual will assist writers of new R Commander plug-in packages, and, in certain cases, facilitate the maintenance and improvement of existing plug-ins.

I make three fundamental assumptions about what you already know how to do:

1. I assume that you know how to program in R. It's probably unreasonable to write an R Commander plug-in package as your first R programming project. There are many existing introductions to programming in R. You'll find a fairly recent bibliography of some of these sources at http://tinyurl.com/ICPSR-R-course.

2. I also assume that you know how to write R packages. An R Commander plug-in package is, in essence, a standard R package with a few special components. If you are new to writing R packages, it's probably a good idea to start with a simple package, even if it's just a toy example, before writing an R Commander plug-in package, but it should be feasible to write an R Commander plug-in as your first serious R package.

   In comparison to R programming, there's a relative dearth of information available on writing R packages. The definitive reference is the *Writing R Extensions* manual (R Core Team, 2016) that comes with the standard R distribution, but, like most manuals, it's not an ideal source for learners. The most ambitious book-length treatment of the subject is Wickham (2015) (also available on-line at http://r-pkgs.had.co.nz/), which describes an idiosyncratic approach to package development. The book is clear, detailed, and comprehensive, but you must buy into Hadley Wickham's approach—an approach that, I should add, has become increasingly popular.

3. Finally, I assume that you have some familiarity with Tcl/Tk, the GUI builder used by the R Commander. The **tcltk** package, which is part of the standard R distribution, provides a convenient interface to Tcl/Tk. It's reasonable to learn Tcl/Tk in order to write an R Commander plug-in. Indeed, I originally learned to use Tcl/Tk in order to write the **Rcmdr** package.

   The **tcltk** package is described by Peter Dalgaard, the author of the package, in two *R News* articles (Dalgaard, 2001, 2002); although these articles are

somewhat dated, they're still useful. Philippe Grosjean maintains a variety of very helpful "R Tcl/Tk recipes," which originated with James Wettenhall, at http://www.sciviews.org/recipes/tcltk/toc/.

There are several books available on Tcl/Tk; although these don't make direct reference to the R **tcltk** package, they are still very useful. My favourite reference is Ousterhout and Jones (2010). There are also valuable on-line resources, such as the manual at https://www.tcl.tk/man/tcl8.4/.

Subsequent chapters of *Writing R Commander Plug-in Packages* deal with the following topics:

Chapter 2 provides an overview of the structure of an R Commander plug-in package.

Chapter 3 explains how the R Commander menus work and how plug-in packages add to and modify them.

Chapter 4 shows you how to construct R Commander dialog boxes.

Chapter 5 shows how to add new classes of statistical models to the R Commander.

Chapter 6 deals with the special challenges that arise in debugging R Commander plug-ins.

Appendix A is a systematic reference for the R Commander utility functions useful in constructing plug-in packages.

I suggest that you download and unpack the sources (i.e., the `.tar.gz` files) for the **Rcmdr**, **RcmdrPlugin.TeachingDemos**, and **RcmdrPlugin.survival** packages, because I'll make repeated references to these packages in subsequent chapters. The three packages are available on CRAN at https://cran.r-project.org/web/packages/ and from CRAN mirrors (e.g., https://cloud.r-project.org/web/packages/).

2

The General Structure of R Commander Plug-in Packages

This chapter provides an overview of the structure of an R Commander plug-in package, with the details elaborated in subsequent chapters. The chapter begins with a general explanation of how the R Commander GUI works.

2.1 The R Commander GUI

Actions in the R Commander are (in most cases[1]) initiated via its *menu bar*, which is illustrated in Figure 2.1. The several *top-level menus* in the menu bar—*File*, *Edit*, *Data*, and so on—each leads to *sub-menus* and *menu items*, as illustrated in Figure 2.2 for the *Statistics* menu: The *Statistics* menu contains a number of sub-menus—*Summaries*, *Contingency tables*, *Means*, etc.—and in each case, the sub-menu includes menu items, as shown in the figure for the *Summaries* sub-menu.[2] Taken as a whole, the R Commander menus comprise a *tree structure*, with the menu bar at the *root*, the top-level menus as the *main branches*, and menu items as the *leaves* (or ultimate branches).

Each menu item in the R Commander is associated with a *callback function*. This is an ordinary R function that is called (with no arguments) when the corresponding menu item is selected. The callback function, in turn, does one of two things:

1. It directly initiates an action. An example of a callback function of this kind is the function invoked by *Statistics* > *Summaries* > *Active data set*, which, in turn, calls the R `summary()` function with the current data set as its argument—e.g., `summary(Duncan)`.
2. It constructs a Tcl/Tk *modal dialog box* seeking additional user input.[3] By convention, menu items leading to dialogs end in ellipses (. . .); for example *Statistics* > *Summaries* > *Numerical summaries*....

Menu items in the R Commander may either be *active* or *inactive* ("grayed-out"), depending upon whether or not they are appropriate in the current context. For example, in the absence of an active data set, almost all of the menu items under the *Statistics* menu would be inactive.[4]

[1] Actions may also be initiated by pressing buttons in the R Commander *toolbar*: to change the active data set or active statistical model, or to edit or view the active data set. Additionally, the user may enter and execute commands in the R Commander *R Script* tab.

[2] In one case, a sub-menu of the *Summaries* menu also contains a sub-sub-menu: *Summaries* > *Dimensional analysis* > *Cluster analysis*.

[3] A *modal dialog* causes the main application—in this case, the R Commander—to wait until the user dismisses the dialog, typically by pressing the *OK* or *Cancel* button in the dialog.

[4] The lone exception is *Statistics* > *Contingency tables* > *Enter and analyze two-way table...* .

FIGURE 2.1: The R Commander menu bar and top-level menus.

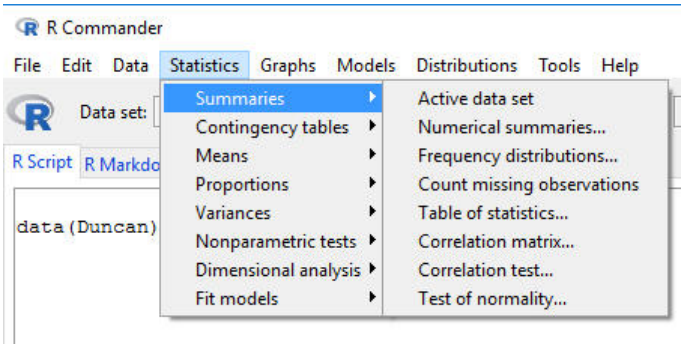

FIGURE 2.2: The *Statistics* menu, showing the expanded *Summaries* sub-menu. The data set from the **car** package was previously made the active data set in the R Commander via the *Data* > *Data in packages* > *Read data from an attached package...* menu item and associated dialog.

2.2 Elements of R Commander Plug-in Packages

An R Commander plug-in package is, in the first instance, an ordinary R package, and so the source tree for a plug-in package includes the usual `DESCRIPTION` and `NAMESPACE` files, along with `R` and `man` subdirectories. Other standard package files and directories may also be present, such as a `NEWS` file, detailing the history of changes to the package, and a `data` subdirectory, if the package provides data sets.

The contents of these files and directories is more or less standard, with two exceptions:

1. The package `DESCRIPTION` file may contain an `RcmdrModels:` field defining new classes of statistical models provided by the package, as explained in Chapter 5.
2. At least some of the `.R` files in the `R` directory will define callback functions, to be invoked by menu items provided by the plug-in package. R Commander dialogs are discussed in Chapter 4.

Plug-in packages can also add to and otherwise modify the R Commander menu tree, via the file `menus.txt` in the source package's `inst/etc` subdirectory.[5] The structure of the `menus.txt` file, and more generally how the R Commander menus work, are described in detail in Chapter 3.

As an illustration, the directory and file tree for the **RcmdrPlugin.survival** package is shown in slightly abbreviated form in Figure 2.3:

- A few inessential directories and files are omitted. For example, there is, in addition to the directories shown, a `po` directory containing files for translating messages from English into other languages.
- The ellipses (`...`) in the file and directory tree represent additional `.Rd` and `.R` files that are omitted for brevity.
- The file `v49i07.pdf` in the `doc` subdirectory contains documentation for the package, in the form of a paper (Fox and Carvalho, 2012) that appeared in the *Journal of Statistical Software*.

To reiterate, with the exception of the file `inst/etc/menus.txt`, all of the other directories and files *could* appear in any R package. In subsequent chapters, I'll ignore aspects that are common to all R packages, such as `.Rd` documentation files, files in the `data` subdirectory, and most of the content of the `DESCRIPTION` and `NAMESPACE` files. Using the **RcmdrPlugin.TeachingDemos** and **RcmdrPlugin.survival** packages as examples, I'll focus instead on what's unique to R Commander plug-in packages.

[5] Under a Windows system, this subdirectory would typically be represented as `inst\etc`; in this manual, I'll use forward-slashes (`/`) to separate directories.

```
DESCRIPTION
NAMESPACE
NEWS
data
    Dialysis.rda
    Rossi.rda
inst
    CITATION
    doc
        v49i07.pdf
    etc
        menus.txt
man
    Dialysis.Rd
    mfrow.Rd
    plot.coxph.Rd
    ...
R
    CoxModel.R
    diagnostics.R
    globals.R
    ...
```

FIGURE 2.3: Abbreviated directory and file tree for the **RcmdrPlugin.survival** source package.**RcmdrPlugin.survival** Directories are shown in italics, and the ellipses (...) represent omitted `.Rd` and `.R` files.

2.3 Making R Commander Plug-ins Self-Starting

On start-up, the R Commander searches for plug-in packages in the user's R package library, recognizing a plug-in package by the presence of the tell-tale `etc/menus.txt` (described in the next chapter) in the installed package. This allows the user to load the plug-in package and restart the R Commander interface via the menu selection *Tools* > *Load Rcmdr plug-in(s)*.

An R Commander plug-in can also be made *self-starting* by including the `.onAttach()` function in Figure 2.4 in the plug-in package's sources.[6] A self-starting plug-in can be loaded directly via the `library()` command (e.g., `library(RcmdrPlugin.survival)`), which also loads the **Rcmdr** package and starts the R Commander GUI with the plug-in activated.

[6] I'm grateful to Richard Heiberger for contributing this self-starting mechanism to the R Commander.

```
.onAttach <- function(libname, pkgname){
    if (!interactive()) return()
    Rcmdr <- options()$Rcmdr
    plugins <- Rcmdr$plugins
    if (!pkgname %in% plugins) {
        Rcmdr$plugins <- c(plugins, pkgname)
        options(Rcmdr=Rcmdr)
        if("package:Rcmdr" %in% search()) {
            if(!getRcmdr("autoRestart")) {
                closeCommander(ask=FALSE, ask.save=TRUE)
                Commander()
            }
        }
        else {
            Commander()
        }
    }
}
```

FIGURE 2.4: The `.onAttach()` function to make an R Commander plug-in package self-starting. The code for this function is available for download at http://socserv.mcmaster.ca/jfox/Books/RCommander/onAttach.R. This is a very slightly modified version of a function originally contributed by Richard Heiberger.

2.4 The DESCRIPTION and NAMESPACE Files for a Plug-in Package

Most of what's involved in writing `NAMESPACE` and `DESCRIPTION` files for R Commander plug-ins is general to R packages, but there are a few special considerations. To focus the discussion, the `DESCRIPTION` and `NAMESPACE` files for the **RcmdrPlugin.TeachingDemos** and **RcmdrPlugin.survival** packages appear in Figures 2.5 and Figures 2.6.

The following points are noteworthy:

- Because you'll almost surely use many of the utility functions in the **Rcmdr** package to construct dialog boxes for your plug-in, and because the **Rcmdr** package re-exports many functions from the **tcltk** and **tcltk2** packages that you may want to use, it generally makes sense to import the complete namespace of the **Rcmdr** package. This is reflected in the `Imports:` fields of both illustrative `DESCRIPTION` files and in the `import()` directives of both `NAMESPACE` files.

- You may have to import additional functions from the **tcltk** or **tcltk2** packages that aren't re-exported by the R Commander package. You can, but need not, list these packages under `Imports:` in the `DESCRIPTION` file, because they are already dependencies of the **Rcmdr** package, but you should use appropriate `importFrom()` directives in your `NAMESPACE` file, as illustrated for the **RcmdrPlugin.TeachingDemos** and **RcmdrPlugin.survival** packages in Figure 2.6, both of which import specific functions from the **tcltk** package.

- If your plug-in package provides a GUI for another R package—as is the case for the two illustrative plug-ins, which create menus and dialogs respectively for the **TeachingDemos** (Snow, 2016) and **survival** (Therneau, 2015) packages—it likely makes sense to list the

```
Package: RcmdrPlugin.TeachingDemos
Type: Package
Title: Rcmdr Teaching Demos Plug-in
Version: 1.1-0
Date: 2015-12-08
Author: John Fox <jfox@mcmaster.ca>
Maintainer: John Fox <jfox@mcmaster.ca>
Depends: rgl, TeachingDemos (>= 2.9), tkrplot
Imports: Rcmdr
Description: Provides an Rcmdr "plug-in" based on the TeachingDemos package,
  and is primarily for illustrative purposes.
License: GPL (>= 2)
```

```
Package: RcmdrPlugin.survival
Type: Package
Title: R Commander Plug-in for the 'survival' Package
Version: 1.1-1
Date: 2016-08-15
Author: John Fox
Maintainer: John Fox <jfox@mcmaster.ca>
Depends: survival, date, stats
Imports: Rcmdr (>= 2.2-1)
Description: An R Commander plug-in for the survival
  package, with dialogs for Cox models, parametric survival regression models,
  estimation of survival curves, and testing for differences in survival
  curves, along with data-management facilities and a variety of tests,
  diagnostics and graphs.
License: GPL (>= 2)
LazyLoad: yes
LazyData: yes
RcmdrModels: coxph, survreg, coxph.penal
```

FIGURE 2.5: Package `DESCRIPTION` files for the **RcmdrPlugin.TeachingDemos** (top) and **RcmdrPlugin.survival** (bottom) plug-in packages (slightly edited for clarity).

```
# RcmdrPlugin.TeachingDemos: last modified 2015-12-08

import(stats, Rcmdr, TeachingDemos, rgl, tkrplot)
importFrom("tcltk", "tkbutton")
importFrom("graphics", "plot", "title")
exportPattern("^[^\\.]")
```

```
# RcmdrPlugin.survival: last modified 2015-08-26

import(stats, Rcmdr, survival, date)
importFrom("grDevices", "palette")
importFrom("graphics", "legend", "plot")
importFrom(tcltk, tkfont.create, ttkprogressbar, tkwidget, setTkProgressBar,
    ttknotebook, tkadd, tkselect)

exportPattern("^[^\\.]")

S3method(plot, coxph)
S3method(unfold, data.frame)
```

FIGURE 2.6: Package `NAMESPACE` files for the **RcmdrPlugin.TeachingDemos** (top) and **RcmdrPlugin.survival** (bottom) plug-in packages (slightly edited for clarity).

package in the `Depends:` field in the `DESCRIPTION` file (as in Figure 2.5) and to import the entire package namespace (as in Figure 2.6).

- In other cases, you can be less promiscuous about imports, as you can be for any R package.
- Your plug-in should export all callback functions, so that these will be available to the **Rcmdr** package for building menus. Also export whatever else should be available globally to a user duplicating the commands that your plug-in generates. In the case of the two illustrative packages, it was sensible and simplest to use the `exportPattern()` directive to export all objects defined in the package (whose names don't start with a period, "."). As illustrated, it's also necessary to declare method functions that should be publicly available—`plot.coxph()` and `unfold.data.frame()` in the case of the **RcmdrPlugin.survival** package.
- The `RcmdrModels:` field, which appears in the `DESCRIPTION` file for the **RcmdrPlugin.survival** package, is special, and is explained in Section 5.4.

Finally, the macro-like R Commander utility functions used to construct dialog boxes (described in Appendix A and many other places in this manual) create local variables in the environments of your callback functions the existence of which isn't apparent to `R CMD check` when you—or CRAN—check your package. There may also be other sources of apparently missing global objects that aren't really missing.

To avoid "`no visible binding for global variable`" notes during the package-checking process, which may prevent the CRAN maintainers from accepting your package, use the `globalVariables()` function in the package sources to flag these objects. Both of the illustrative plug-ins do this in a file named `globals.R` in the package sources, as shown

```
# RcmdrPlugin.TeachingDemos: created 2012-08-28 by J. Fox

globalVariables(c('top', 'buttonsFrame', 'slider.env')
```

```
# RcmdrPlugin.survival: last modified 2015-08-27

globalVariables(c('.dfbeta', '.mfrow', '.dfbetas',
    'top', 'tiesVariable', 'robustVariable', 'subsetVariable', 'rhsVariable',
    'tiesFrame', 'robustFrame', 'xBox', 'outerOperatorsFrame', 'operatorsFrame',
    'formulaFrame', 'subsetFrame', 'buttonsFrame', 'rhsEntry', '.CoxZPH', '.b',
    '.residuals', '.X', '.fitted', '.tab.1.1', 'confintVariable', '.newdata',
    'confintFrame', 'typeFrame', 'typeVariable', 'errorVariable', 'survtypeVariable',
    'detailVariable', 'conftypeVariable', 'plotconfVariable', '.Survfit',
    'survtypeFrame', 'detailFrame', 'conftypeFrame', 'plotconfFrame', 'errorFrame',
    'factorsButton', 'allButton', 'clusterButtonsFrame', 'newVar',
    'distributionVariable', 'distributionFrame', 'dataTab', 'optionsTab',
    'modelTab', 'notebook')
```

FIGURE 2.7: The `globals.R` source files for the **RcmdrPlugin.TeachingDemos** (top) and **RcmdrPlugin.survival** (bottom) plug-in packages (slightly edited for clarity).

in Figure 2.7. To discover which objects to include, simply examine the initial package-check report for "`no visible binding for global variable`" notes.

3

R Commander and Plug-in Menus

The R Commander menus are defined in the file `Rcmdr-menus.txt`, which resides in the `inst/etc` subdirectory of the **Rcmdr** package sources. When the **Rcmdr** package is installed in a user's library, `Rcmdr-menus.txt`, therefore, is in the `Rcmdr/etc` subdirectory. With one exception, described in Section 3.2, the structure of a plug-in package's `menus.txt` file is identical to `Rcmdr-menus.txt`.

3.1 The `Rcmdr-menus.txt` File

Figure 3.1 shows the lines in the `Rcmdr-menus.txt` file that define the R Commander *File* menu, which is displayed fully expanded in Figure 3.2. These lines appear near the top of `Rcmdr-menus.txt`.

It may help to know that the R Commander uses the `read.table()` command to input the `Rcmdr-menus.txt` file, with all columns of the resulting data frame declared as `"character"`. As well, trailing empty strings (`""`) are implied if there are fewer than seven fields in a line, although it is clearer to put in the empty strings explicitly, as I've done in Figure 3.1.

Each line (row) of the `Rcmdr-menus.txt` file represents a *menu directive*. The meaning of each *field* (column) is indicated in the comment (i.e., preceded by `#`) in the first line:

`type` Whether the menu directive defines a `menu` or a menu `item`. Two menus are defined in Figure 3.1—the top-level *File* menu and its *Exit* sub-menu—and all of the other menu directives define menu "items" (but see the discussion of `operation/parent` below). If the type is `remove`—a third type—then the corresponding menu or menu item is deleted. Menu or item deletion is, of course, only sensible for a plug-in package, not in the **Rcmdr** package itself, and so there are no `delete` menu directives in the `Rcmdr-menus.txt` file.

`menu/item` In the case of a *menu* definition, this is the name of the menu-object to be defined; the name is arbitrary, but it should be unique, and any legal R name will do. There are two menu objects defined in the example, named `fileMenu` and `exitMenu`.

In the case of a menu *item* definition, this field gives the name of the menu-object to which the item belongs.

In the case of a *cascade* operation for a menu (explained further immediately below), the `menu/item` field specifies the name of the higher-level menu under which the menu is to be inserted. For example, in the last two lines of Figure 3.1, `exitMenu` is inserted under `fileMenu`, and `fileMenu` is inserted under `topMenu` (representing the R Commander menu bar).

Finally, for a menu or a menu-item removal, this field gives the name of the menu to be deleted or the name of the callback function (see below) for the item to be deleted.

```
#   type    menu/item     operation/parent  label                          command/menu              activation  install?

    menu    fileMenu        topMenu         ""                             ""                        ""          ""
    item    fileMenu        command         "Change working directory..."  Setwd                     ""          ""
    item    fileMenu        separator       ""                             ""                        ""          ""
    item    fileMenu        command         "Open script file..."          loadLog                   ""          ""
    item    fileMenu        command         "Save script..."               saveLog                   ""          ""
    item    fileMenu        command         "Save script as..."            saveLogAs                 ""          ""
    item    fileMenu        separator       ""                             ""                        ""          "getRcmdr('use.markdown') || getRcmdr('use.knitr')"
    item    fileMenu        command         "Open R Markdown file..."      loadRmd                   ""          "getRcmdr('use.markdown')"
    item    fileMenu        command         "Open knitr file..."           loadRnw                   ""          "getRcmdr('use.knitr')"
    item    fileMenu        command         "Save R Markdown file..."      saveRmd                   ""          "getRcmdr('use.markdown')"
    item    fileMenu        command         "Save knitr file..."           saveRnw                   ""          "getRcmdr('use.knitr')"
    item    fileMenu        command         "Save R Markdown file as..."   saveRmdAs                 ""          "getRcmdr('use.markdown')"
    item    fileMenu        command         "Save knitr file as..."        saveRnwAs                 ""          "getRcmdr('use.knitr')"
    item    fileMenu        separator       ""                             ""                        ""          ""
    item    fileMenu        command         "Save output..."               saveOutput                ""          ""
    item    fileMenu        command         "Save output as..."            saveOutputAs              ""          ""
    item    fileMenu        separator       ""                             ""                        ""          ""
    item    fileMenu        command         "Save R workspace..."          saveWorkspace             ""          ""
    item    fileMenu        command         "Save R workspace as..."       saveWorkspaceAs           ""          ""
    item    fileMenu        separator       ""                             ""                        ""          ""
    menu    exitMenu        fileMenu        ""                             ""                        ""          ""
    item    exitMenu        command         "From Commander"               CloseCommander            ""          ""
    item    exitMenu        command         "From Commander and R"         closeCommanderAndR        ""          ""
    item    fileMenu        cascade         "Exit"                         exitMenu                  ""          ""
    item    topMenu         cascade         "File"                         fileMenu                  ""          ""
```

FIGURE 3.1: The lines (slightly edited) in the `Rcmdr-menus.txt` file that define the R Commander *File* menu, its *Exit* sub-menu, and their menu items.

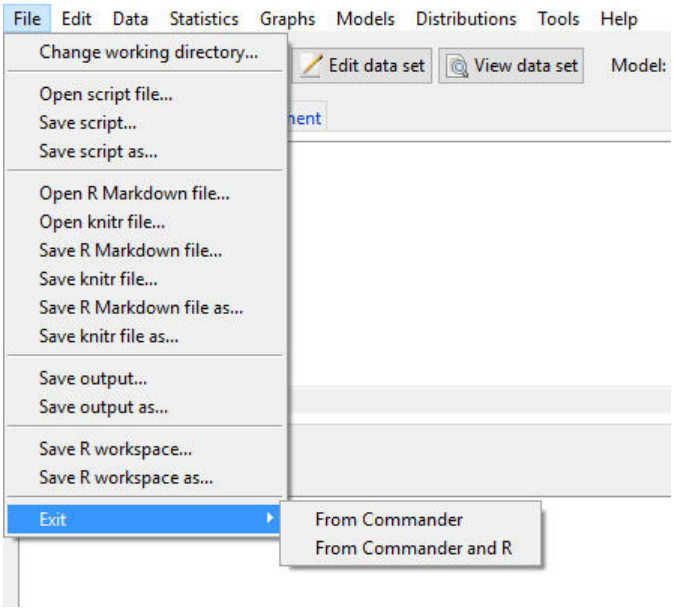

FIGURE 3.2: The R Commander *File* menu and its *Exit* sub-menu.

`operation/parent` In the case of a *menu* definition, this is the name of the parent menu to which the menu belongs. For a top-level menu like `fileMenu`, the parent is `topMenu`. For the sub-menu `exitMenu`, the parent is `fileMenu`.

Otherwise, one of three operations is specified:

`command` The resulting menu item will dispatch a callback function. For example, the second menu directive defines a menu item that dispatches the function `Setwd()` (see the explanation of `command/menu` below).

`cascade` The menu directive inserts a menu or sub-menu (and associated menu items) under its parent. For example, the penultimate menu directive in Figure 3.1 inserts the *Exit* menu under the *File* menu, and the last line inserts the *File* menu in the menu-bar.

`separator` A horizontal separator is inserted into the corresponding menu. Separators are used to group related items (and sub-menus). For example, there are separators demarcating the three items in the *File* menu relating to script files.

`label` The text to be displayed for a menu item or menu. Conventionally, this text ends in ellipses (`...`) if the corresponding menu item leads to a dialog box.

`command/menu` In the case of a `command` operation, this is the name of the callback function corresponding to the menu item. Callback functions are defined in the **Rcmdr** package and are not exported from the package's namespace. For example, the second directive Figure 3.1 creates a menu item that calls the `Setwd()` function, and the fourth directive an item that calls the `loadLog()` function, both of which are defined by the **Rcmdr** package.

In a plug-in package, in contrast, it's necessary to export callback functions defined by the package so that the R Commander can find them when it builds the menus (see Section 2.4).

`activation` This is an R expression, given as a character string, that should evaluate to `TRUE` or `FALSE`. If the expression evaluates to `FALSE`, then the corresponding menu item is *inactive*—i.e., grayed out. The empty string `""` is treated as unconditionally `TRUE`, and so all of the menu items in the *File* menu are always active.

The **Rcmdr** package defines and exports specialized *predicate functions* for testing particular conditions. For example, `factorsP()` returns `TRUE` if there are any factors in the active data set and `FALSE` otherwise, while `factorsP(2)` returns `TRUE` if there are two or more factors in the active data set.

We'll see examples of conditional activation in Section 3.2.2, when we examine the `menus.txt` files for the **RcmdrPlugin.survival** package.[1]

`install?` An R expression, also given as a character string, that causes the corresponding menu or menu item to be installed if the expression evaluates to `TRUE` or to be suppressed if it evaluates to `FALSE`. Again, the empty character string `""` is treated as unconditionally `TRUE`. Thus, the *Exit* and *File* menus, and many of the menu items under these menus, are installed in any event, but some menu items are installed only if the *R Markdown* or *knitr Document* tabs are in use.[2]

Menu directives are processed in order, and the sequence of directives must therefore make sense. For example, a menu (e.g., `fileMenu` in Figure 3.1) must be defined before menu items can be placed in it; and the menu items should be placed in the menu before the menu is installed under its parent via a `cascade` operation.

3.2 The `menus.txt` File for an R Commander Plug-in Package

When the R Commander loads a plug-in package, it restarts the R Commander interface. The menu directives in the plug-in's `menus.txt` file are processed after the directives in `Rcmdr-menus.txt`, although the R Commander is careful to merge the plug-in's directives with the R Commander menus in a sensible manner. If several plug-ins are loaded, they are processed sequentially.[3]

Figures 3.3 and 3.5 show the `menus.txt` files for the **RcmdrPlugin.TeachingDemos** and **RcmdrPlugin.survival** packages; I edited these files slightly for clarity (but the content of the menu directives wasn't changed). Figures 3.4 and 3.6 show the modifications and additions that the two plug-ins make to the R Commander menus.

3.2.1 The RcmdrPlugin.TeachingDemos Package

I originally developed the **RcmdrPlugin.TeachingDemos** package to demonstrate the process of writing an R Commander plug-in (see Fox, 2007). It is the simpler of the two packages, and so I will start with it. The **RcmdrPlugin.TeachingDemos** package uses the

[1] Also see Section A.4 in Appendix A for a complete list of R Commander predicate functions.

[2] Recall (Fox, 2017, Section 3.3.6.2) that by default the R Commander constructs an R Markdown document to create a report of the current session, and that a knitr LaTeX document may be constructed optionally. The use of the `getRcmdr()` function for retrieving state information is described in Section 4.2 and Section A.1 in Appendix A.

[3] Although each plug-in package will work individually if it is written properly, different plug-ins aren't necessarily compatible with each other, and compatibility *may* depend on the order in which the plug-ins are loaded. For example, if an earlier plug-in removes a menu into which a later plug-in tries to install a menu item, then an error will result.

TeachingDemos package (Snow, 2016) to create a variety of demonstrations appropriate for a basic statistics course.

The first three menu directives in the `menus.txt` file for the **RcmdrPlugin.TeachingDemos** plug-in package remove three menu items from the standard R Commander menus, items for plotting the normal, t, and gamma distributions. The fourth menu directive removes the entire *Discrete distributions* sub-menu from the R Commander *Distributions* menu. To know what the removed items—`normalDistributionPlot`, `tDistributionPlot`, `gammaDistributionPlot`, and `discreteMenu`—represent, you have to examine the R Commander `Rcmdr-menus.txt` file.

The *ostensible* rationale for deleting the three distribution-plot menu items (and, less credibly, the entire *Discrete distributions* sub-menu) is that the **RcmdrPlugin.TeachingDemos** package provides superior replacements, but the *real* reason is to demonstrate how to delete menu items and menus. To editorialize slightly, I urge you to think carefully about whether you really want to remove standard R Commander menus or menu items. Doing so may confuse your users and make your plug-in incompatible with other R Commander plug-ins. A good reason to remove an R Commander menu item, however, is if you think that you have a superior replacement for it *that will be installed in the same menu and with the same label* as the standard menu item.

Continuing with the `menus.txt` file, the next few lines define a new *Demos* top-level menu, define several menu items under this menu (*Central limit theorem...*, *Confidence interval for the mean...*, etc.), and install the *Demos* menu in the menu-bar.

The final block of menu directives creates a new *Visualize distributions* sub-menu under the R Commander *Distributions* menu; defines several menu items for visualizing various distributions (*Binomial distribution*, etc.); and installs the *Visualize distributions* sub-menu under *Distributions*.

The various callback functions—`centralLimitTheorem()`, `simulateConfidenceIntervals()`, and so on—are defined in the **RcmdrPlugin.TeachingDemos** package (as discussed in Chapter 4). As is apparent from the `""` entry in the *activation* field of each menu directive, all of the menu-items are always active. The `packageAvailable()` function, used in the `install?` field, is provided by the **Rcmdr** package (see Appendix A), and returns `TRUE` if the **TeachingDemos** package is installed in the user's library and `FALSE` if it is not.[4]

3.2.2 The RcmdrPlugin.survival Package

The **RcmdrPlugin.survival** package (Fox and Carvalho, 2012) was developed for a more serious purpose: to provide a graphical user interface to many of the facilities of the **survival** package (Therneau and Grambsch, 2000; Therneau, 2015), which is state-of-the-art software for survival analysis and part of the standard R distribution.

As can be seen in Figures 3.5 and 3.6, the `menus.txt` file for the **RcmdrPlugin.survival** package:

- Creates a new *Survival analysis* sub-menu under the R Commander *Statistics* top-level menu and a new *Survival data* sub-menu under the R Commander *Data* top-level menu. Each of these new menus includes three menu-items.
- Places new menu items for Cox regression and parametric survival regression under the R Commander *Statistics* > *Fit models* menu, preceded by a menu separator.

[4] Because the **TeachingDemos** package is a dependency of the **RcmdrPlugin.TeachingDemos** package, the former is unlikely to be absent if the latter is present.

```
# menus for the RcmdrPlugin.TeachingDemos package

# last modified: 22 July 2008 by J. Fox

# type menu/item              operation/parent  label                              command/menu               activation install?

remove normalDistributionPlot "" "" "" "" ""
remove tDistributionPlot      "" "" "" "" ""
remove gammaDistributionPlot  "" "" "" "" ""
remove discreteMenu           "" "" "" "" ""

menu   demosMenu              topMenu           ""                                 ""                         ""         ""
item   demosMenu              command           "Central limit theorem..."         centralLimitTheorem        ""         "packageAvailable('TeachingDemos')"
item   demosMenu              command           "Confidence interval for the mean..." simulateConfidenceIntervals ""     "packageAvailable('TeachingDemos')"
item   demosMenu              command           "Power of the test"                powerExample               ""         "packageAvailable('TeachingDemos')"
item   demosMenu              command           "Flip a coin"                      flipCoin                   ""         "packageAvailable('TeachingDemos')"
item   demosMenu              command           "Roll a die"                       rollDie                    ""         "packageAvailable('TeachingDemos')"
item   demosMenu              command           "Simple linear regression"         linearRegressionExample    ""         "packageAvailable('TeachingDemos')"
item   demosMenu              command           "Simple correlation"               correlationExample         ""         "packageAvailable('TeachingDemos')"
item   topMenu                cascade           "Demos"                            demosMenu                  ""         "packageAvailable('TeachingDemos')"

menu   visualMenu             distributionsMenu ""                                 ""                         ""         ""
item   visualMenu             command           "Binomial distributions"           visBinom                   ""         "packageAvailable('TeachingDemos')"
item   visualMenu             command           "Normal distributions"             visNormal                  ""         "packageAvailable('TeachingDemos')"
item   visualMenu             command           "t distributions"                  vist                       ""         "packageAvailable('TeachingDemos')"
item   visualMenu             command           "Gamma distributions"              visGamma                   ""         "packageAvailable('TeachingDemos')"
item   distributionsMenu      cascade           "Visualize distributions"          visualMenu                 ""         "packageAvailable('TeachingDemos')"
```

FIGURE 3.3: The `menus.txt` file for the **RcmdrPlugin.TeachingDemos** package, version 1.1-0 (slightly edited).

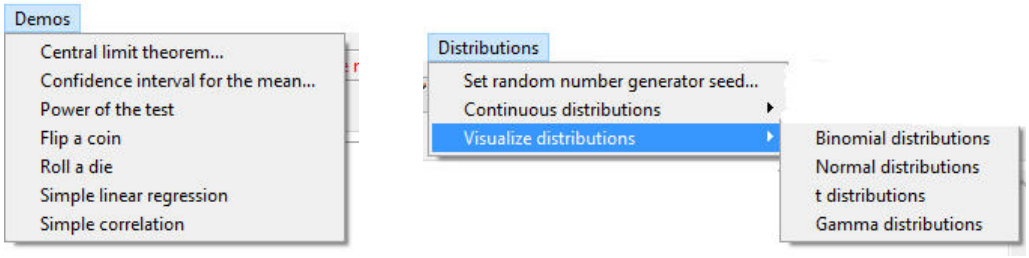

FIGURE 3.4: The *Demos* and *Distributions* > *Visualize distributions* menus after loading the **RcmdrPlugin.TeachingDemos** package. Notice that the *Discrete distributions* submenu is gone from the *Distributions* menu.

- Places a new menu item to test for proportional hazards under the R Commander *Models* > *Numerical diagnostics* menu, preceded by a separator.
- Defines several new menu items under the R Commander *Models* > *Graphs* menu, preceded by a separator.

The various callback functions for the new menu items are defined in the **RcmdrPlugin.survival** package (as discussed in Chapter 4).

Three of the predicate functions used for menu activation and installation are defined in the **Rcmdr** package (see Section A.4 in Appendix A for details):

- `activeDataSetP()` returns `TRUE` if there is an active data set and `FALSE` otherwise.
- `factorsP()` returns `TRUE` if there are one or more factors in the current data set and `FALSE` if there are no factors.
- `packageAvailable('survival')` returns `TRUE` if the **survival** package is in the user's library and `FALSE` otherwise. (Because the **survival** package is part of the R distribution, it would be very odd for it to be missing!)

The other predicates used for menu-item activation are provided by the **RcmdrPlugin.survival** package:

- `coxphP()` returns `TRUE` if the current statistical model is a Cox model, and `FALSE` otherwise—i.e., if there is *no* current model or if the current model is of a different class.
- `survregP()` returns `TRUE` if the current model is a parametric survival regression model and `FALSE` otherwise.
- `highOrderTermsP()` returns `TRUE` if the current model includes interactions and `FALSE` if it is additive.

```
# menus for the RcmdrPlugin.survival package

# last modified: 2013-04-24 by J. Fox

# type menu/item        operation/parent label                              command/menu   activation                          install?

menu   survivalMenu     statisticsMenu   ""                                 ""             ""                                  ""
item   survivalMenu     command          "Estimate survival function..."    Survfit        "activeDataSetP()"                  "packageAvailable('survival')"
item   survivalMenu     command          "Compare survival functions..."    Survdiff       "factorsP()"                        "packageAvailable('survival')"
item   statisticsMenu   cascade          "Survival analysis"                survivalMenu   ""                                  "packageAvailable('survival')"

menu   survDataMenu     dataMenu         ""                                 ""             ""                                  ""
item   survDataMenu     command          "Survival data definition..."      SurvivalData   "activeDataSetP()"                  "packageAvailable('survival')"
item   survDataMenu     command          "Convert wide to long data..."     Unfold         "activeDataSetP()"                  "packageAvailable('survival')"
item   survDataMenu     command          "Convert variable to date..."      toDate         "activeDataSetP()"                  "packageAvailable('survival')"
item   dataMenu         cascade          "Survival data"                    survDataMenu    ""                                 "packageAvailable('survival')"

item   statModelsMenu   separator        ""                                 ""             ""                                  ""
item   statModelsMenu   command          "Cox regression model..."          CoxModel       "activeDataSetP()"                  "packageAvailable('survival')"
item   statModelsMenu   command          "Parametric survival model..."     survregModel   "activeDataSetP()"                  "packageAvailable('survival')"

item   diagnosticsMenu  separator        ""                                 ""             ""                                  ""
item   diagnosticsMenu  command          "Test proportional hazards"        CoxZPH         "coxphP()"                          "packageAvailable('survival')"

item   modelsGraphsMenu separator        ""                                 ""             ""                                  ""
item   modelsGraphsMenu command          "Cox-model survival function..."   PlotCoxph      "coxphP()"                          "packageAvailable('survival')"
item   modelsGraphsMenu command          "Plot terms in Cox model"          TermPlots      "coxphP() && !highOrderTermsP()"    "packageAvailable('survival')"
item   modelsGraphsMenu command          "Plot survival-regression dfbetas" CoxDfbetas     "coxphP() || survregP()"            "packageAvailable('survival')"
item   modelsGraphsMenu command          "Plot survival-regression dfbeta"  CoxDfbeta      "coxphP() || survregP()"            "packageAvailable('survival')"
item   modelsGraphsMenu command          "Plot null Martingale residuals"   MartingalePlots "coxphP()"                        "packageAvailable('survival')"
item   modelsGraphsMenu command          "Cox-model partial-residual plots" PartialResPlots "coxphP()"                        "packageAvailable('survival')"
```

FIGURE 3.5: The `menus.txt` file for the **RcmdrPlugin.survival** package, version 1.1-1 (slightly edited).

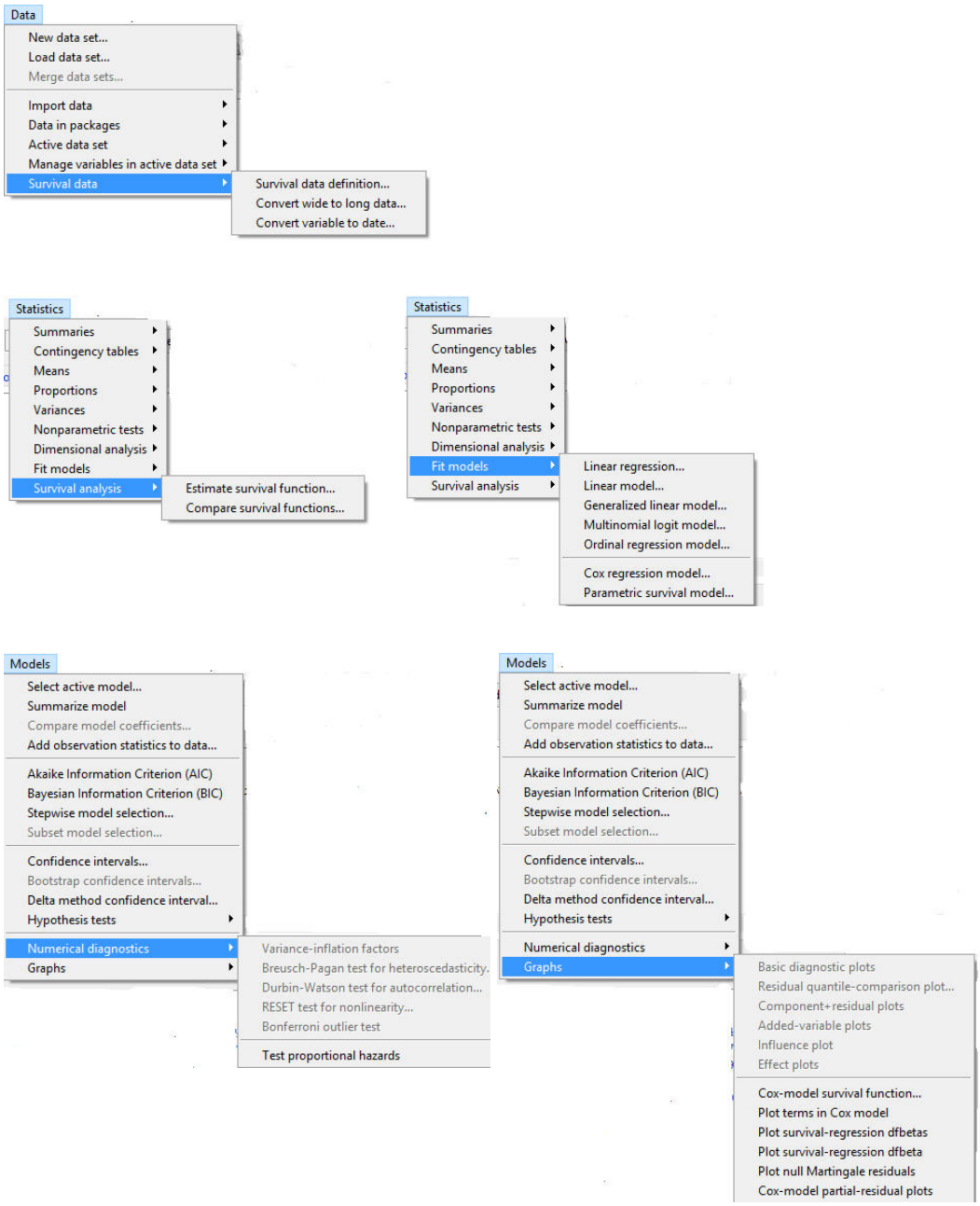

FIGURE 3.6: Additions to the R Commander menus by the **RcmdrPlugin.survival** package.

4

Building R Commander Dialog Boxes

Constructing Tcl/Tk dialog boxes is substantially more complicated than specifying R Commander menus. As explained in the preceding chapter, selecting an R Commander or plug-in menu item dispatches a callback function—an R function that is called with no arguments. Normally, a callback function either composes and executes an R command within the R Commander (a process described in this chapter), or brings up a Tcl/Tk dialog.

As a formal matter, however, a callback function can be *any* argument-less R function, raising the possibility, for example, of using a GUI-builder in R *other than* Tcl/Tk (see, e.g., Lawrence and Verzani, 2012) to construct a dialog box called from the R Commander. That said, there are several arguments in favor of sticking with Tcl/Tk:

1. The **tcltk** package is part of the standard R distribution, and on Windows and Mac OS X systems, Tcl/Tk is installed along with R. It's therefore generally simple to get Tcl/Tk-based GUIs to work in R, which, in my experience, is not necessarily true of other GUI toolkits. This is why I employed Tcl/Tk for the R Commander.
2. Using Tcl/Tk will make your dialog boxes appear similar to the R Commander dialogs. Uniformity in appearance is desirable aesthetically and is less likely to confuse users.
3. The **Rcmdr** package includes many utility functions (described in this and the next chapter, and in Appendex A) to assist you in constructing dialogs using Tcl/Tk, including for initializing and finalizing dialogs, adding *OK*, *Cancel*, *Help*, *Apply*, and *Reset* buttons, creating related sets of radio buttons and check boxes, and so on.

In the balance of this chapter, I'll explain the process of constructing plug-in dialog boxes using dialog-building functions from the R Commander package as illustrations. I'll also explain how the R Commander stores and retrieves state information, and how R Commander dialogs interact with the R interpreter.

4.1 Examples of R Commander Dialogs

A reasonable strategy for creating your own dialogs is to find similar R Commander dialogs and modify the functions in the R Commander package that build them—that is, treat the R Commander dialog-building functions as "templates" or points of departure for your dialogs. You can locate the callback function in the **Rcmdr** package sources for a particular dialog of interest by examining the `Rcmdr-menus.txt` file, as described in the preceding chapter.

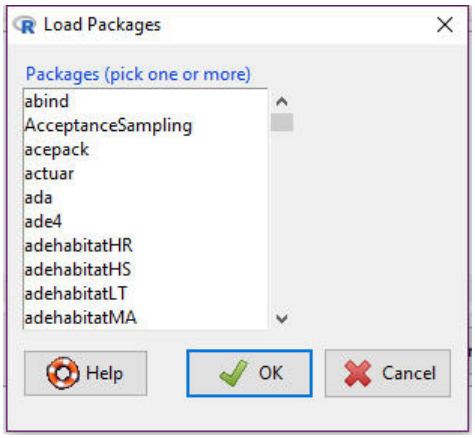

FIGURE 4.1: The R Commander *Load Packages* dialog box.

4.1.1 *Load Packages*: A Simple Dialog With Basic Buttons

I'll begin with a very simple example: Figure 4.1 shows the *Load Packages* dialog, dispatched from the R Commander menus via *Tools* > *Load package(s)...* . The user completes the dialog by selecting one or more packages from the listbox in the usual manner (by, recall, some combination of left-clicking, *Ctrl*-clicking, and *Shift*-clicking) and then clicks the *OK* button. Clicking *OK without* selecting one or more packages prints an error message in the R Commander *Messages* pane and reopens the *Load Packages* dialog.

Alternatively, the user may dismiss the dialog by clicking either the *Cancel* button or the X in its upper-right-hand corner. Finally, clicking the *Help* button brings up the help page for the `library()` command in the user's default web browser, leaving the *Load Packages* dialog open.

Figure 4.2 shows the `loadPackages()` callback function that creates the dialog box in Figure 4.1:

- Like all callback functions, `loadPackages()` has no arguments.
- Called without any arguments, the standard R function `.library()` returns the names of currently attached packages; called with the argument `all.available=TRUE`, it returns the names of all packages in the user's library (or libraries). Consequently, the variable `availablePackages` is a character vector of names of packages not currently attached.

  If `availablePackages` is empty, there are no additional packages to load; `loadPackages()` calls the `errorCondition()` function to print an error message in the R Commander *Messages* pane and stops without creating the *Load Packages* dialog box.
- Otherwise, `loadPackages()` calls `initializeDialog()`, which has one required argument (`title`, which should be specified by name) and several optional arguments. We'll encounter some of these arguments later in this chapter.[1]. The `initializeDialog()` utility performs several operations, including creating a top-level Tk widget, named `top` by default.

[1] As is generally the case for the dialog-box utility functions discussed in this chapter, to see all of the arguments of `initializeDialog()`, consult Appendix A, use the command `args(initializeDialog)`, or examine the `?Rcmdr.Utilities` help page.

```
loadPackages <- function(){
        availablePackages <- sort(setdiff(.packages(all.available=TRUE), .packages()))
        if (length(availablePackages) == 0){
                errorCondition(message=gettextRcmdr("No packages available to load."))
                return()
        }
        initializeDialog(title=gettextRcmdr("Load Packages"))
        packagesBox <- variableListBox(top, availablePackages,
            title=gettextRcmdr("Packages (pick one or more)"),
                        selectmode="multiple", listHeight=10)
        onOK <- function(){
                packages <- getSelection(packagesBox)
                closeDialog(top)
                if (length(packages) == 0){
                        errorCondition(recall=loadPackages,
                message=gettextRcmdr("You must select at least one package."))
                        return()
                }
                for (package in packages) {
                        Library(package)
                }
                Message(paste(gettextRcmdr("Packages loaded:"),
            paste(packages, collapse=", ")), type="note")
        }
        OKCancelHelp(helpSubject="library")
        tkgrid(getFrame(packagesBox), sticky="nw")
        tkgrid(buttonsFrame, sticky="w")
        dialogSuffix()
}
```

FIGURE 4.2: The `loadPackages()` callback function, which creates the *Load Packages* dialog. The code is edited slightly for clarity.

- As its name suggests, `variableListBox()` creates a Tk listbox, which it returns in an object of class `"listbox"`. `variableListBox()` has two required arguments: Its first argument, `parentWindow`, which can be given by position (and is `top` here); and the argument `title`, which should be specified by name.

  The second argument to `variableListBox()`, `variableList`, is a vector of character strings comprising the entries of the listbox. Here, that's the vector `availablePackages`; the default is `Variables()`, which returns the names of the variables in the active data set.

  The argument `selectmode="multiple"` allows the user to select more than one entry in the listbox; the default is `selectmode="single"`.

  Finally, `listHeight=10` specifies that up to 10 packages will be displayed in the listbox window; if there are more than 10 available packages, as is almost surely the case, the vertical scrollbar for the listbox will be activated. The default is `listHeight=getRcmdr("variable.list.height")`, which is the number of values specified by the R Commander `variable.list.height` option (and is 6 by default).[2]

- The argument-less local function `onOK()` determines what happens when the user presses the *OK* button in the dialog. Then `getSelection(packagesBox)` returns a character string of selected package names, and the call to `closeDialog(top)` closes the dialog box.

  If no packages are selected, then `errorCondition()` prints an error message, and the argument `recall=loadPackages` reinvokes the `loadPackages()` function to redisplay the dialog.

  Otherwise, the R Commander `Library()` utility function is called for each selected package to load; `Library()`, in turn, generates calls to the standard `library()` function. Finally, a message (of `type="note"`) is printed in the R Commander *Messages* pane, indicating the packages that were loaded.

- The lines after the definition of `onOK()` complete the specification of the dialog: The call to `OKCancelHelp()` creates the *OK*, *Cancel*, and *Help* buttons for the dialog; there are no required arguments to `OKCancelHelp()`, but to include a *Help* button it's necessary to give the `helpSubject` argument, here `helpSubject="library"`, so pressing the *Help* button executes the command `help("library")`, bringing up the help page for the `library()` command.

  The calls to the **tcltk** function `tkgrid()` place the packages listbox and the *OK*, *Cancel*, and *Help* buttons in the dialog box. Here, `getFrame(packagesBox)` returns the Tk frame widget containing the packages listbox, and `buttonsFrame` is the frame widget containing the buttons, which was created by `OKCancelHelp()`.

  The call to `dialogSuffix()`, which has no required arguments, completes the specification of the dialog box.

- A note about the function `gettextRcmdr()`: You'll no doubt have noticed that that all text-sting messages in the `loadPackages()` function are embedded in a call to the `gettextRcmdr()` function. The R Commander uses GNU gettext to translate English messages into other languages, a process that's supported by R (see Ripley, 2005). Unless you wish to provide a similar translation facility for your plug-in package, you can simply supply messages directly as character strings (and ignore the calls to `gettextRcmdr()` in this and subsequent examples).

[2] See Section 4.2 on storing and retrieving R Commander state information.

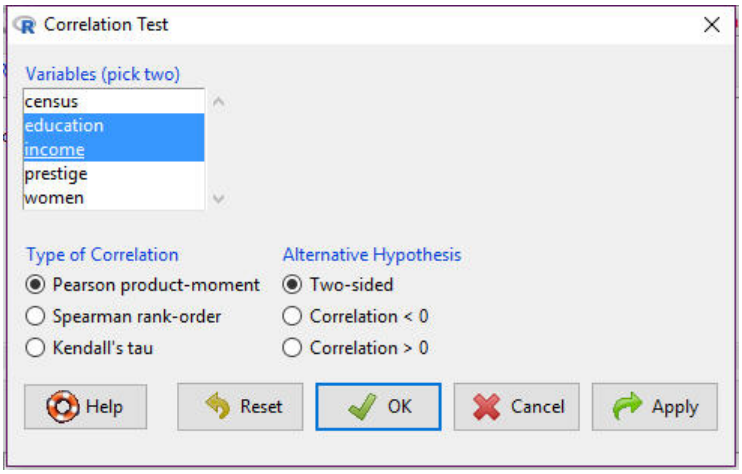

FIGURE 4.3: The R Commander *Correlation Test* dialog box.

Several of the R Commander utilities used in this example are "macro-like" in their behavior, in that, unlike ordinary R functions (which are lexically scoped), they can modify the environment of the function that calls them, here `loadPackages()`. For example, `initializeDialog()` creates the object `top` containing the top-level widget for the dialog box, and `OKCancelHelp()` creates the frame widget `buttonsFrame`, both of which are referenced by `loadPackages()`. This non-standard behavior is convenient to cope with scoping issues that arise in using the **tcltk** package. Macro-like R Commander utilities are marked as such in Appendix A.[3]

4.1.2 *Correlation Test*: A Dialog With Radio Buttons and That Saves Its State

Figure 4.3 shows the R Commander *Correlation Test* dialog, produced by the menu selection *Statistics* > *Summaries* > *Correlation test...*, with `Duncan` (the Duncan occupational prestige data—see, e.g., Section 4.2.2 in the text) as the active data set. I selected the variables `education` and `income` in the variable listbox. Some of the elements of this dialog box are now familiar, such as the listbox, but others are new:

- There are two sets of *radio buttons*—to select the type of correlation to be computed and the alternative hypothesis.
- In addition to *OK*, *Cancel*, and *Help* buttons, the dialog includes *Reset* and *Apply* buttons.

The code for the the `correlationTest()` callback function, which creates the *Correlation Test* dialog, is given in Figure 4.4.[4] We encountered some of the functions used to construct this dialog in the preceding section—for example, `initializeDialog()`, `variableListBox()`, `OKCancelHelp()`, and `dialogSuffix()`. I'll comment here only on arguments to these functions that weren't used previously. Other functions employed in `correlationTest()` are new, such as `getDialog()`, `radioButtons()`, and `putDialog()`.

[3] R Commander macros are created using a slightly modified version of Thomas Lumley's `defmacro()` function (Lumley, 2001).

[4] The `correlationTest()` function was originally contributed by Stefano Calza and subsequently modified by me.

```
correlationTest <- function(){
    defaults <- list(initial.x=NULL, initial.correlations="pearson",
        initial.alternative="two.sided")
    dialog.values <- getDialog("correlationTest", defaults)
    initializeDialog(title=gettextRcmdr("Correlation Test"))
    xBox <- variableListBox(top, Numeric(), selectmode="multiple",
        title=gettextRcmdr("Variables (pick two)"),
        initialSelection=varPosn(dialog.values$initial.x, "numeric"))
    optionsFrame <- tkframe(top)
    radioButtons(optionsFrame, name="correlations",
        buttons=c("pearson", "spearman", "kendall"),
        labels=gettextRcmdr(c("Pearson product-moment",
            "Spearman rank-order", "Kendall's tau")),
        initialValue=dialog.values$initial.correlations,
        title=gettextRcmdr("Type of Correlation"))
    radioButtons(optionsFrame, name="alternative",
        buttons=c("two.sided", "less", "greater"),
        values=c("two.sided", "less", "greater"),
        initialValue=dialog.values$initial.alternative,
        labels=gettextRcmdr(c("Two-sided", "Correlation < 0", "Correlation > 0")),
        title=gettextRcmdr("Alternative Hypothesis"))
    onOK <- function(){
        alternative <- tclvalue(alternativeVariable)
        correlations <- tclvalue(correlationsVariable)
        x <- getSelection(xBox)
        putDialog("correlationTest", list(initial.alternative=alternative,
            initial.correlations=correlations, initial.x=x))
        if (2 > length(x)) {
            errorCondition(recall=correlationTest,
                message=gettextRcmdr("Fewer than 2 variables selected."))
        return()
        }
        if(2 < length(x)) {
            errorCondition(recall=correlationTest,
                message=gettextRcmdr("More than 2 variables selected."))
        return()
        }
       closeDialog()
       .activeDataSet <- ActiveDataSet()
       command <- paste("with(", .activeDataSet, ", cor.test(", x[1], ", ", x[2],
           ', alternative="', alternative, '", method="', correlations, '"))',
           sep="")
       doItAndPrint(command)
       tkfocus(CommanderWindow())
    }
    OKCancelHelp(helpSubject="cor.test", reset="correlationTest", apply="correlationTest")
    tkgrid(getFrame(xBox), sticky="nw")
    tkgrid(labelRcmdr(top, text=""))
    tkgrid(correlationsFrame, labelRcmdr(optionsFrame, text="  "),
        alternativeFrame, sticky="w")
    tkgrid(optionsFrame, sticky="w")
    tkgrid(buttonsFrame, sticky="w")
    dialogSuffix()
}
```

FIGURE 4.4: The `correlationTest()` callback function, which creates the *Correlation Test* dialog. The code is slightly edited.

- R Commander dialog boxes can store *state information* that's preserved from one invocation of the dialog to the next.[5] The first command in the function establishes defaults for the initial, unused state of the dialog—in this instance, the variables selected in the listbox (`NULL` implies no initial selection) and the initial choices for the two sets of radio buttons. The `getDialog()` function takes the name of the dialog-generating function as its first argument[6] and the list of defaults as its second argument, and returns stored values if these exist and the defaults if no values are stored.

- In the call to `variableListBox()`, `Numeric()` returns a character vector with the names of the numeric variables in the current data set. The `initialSelection` argument indicates which (if any) variables are initially selected in the listbox. These are taken from `dialog.values$initial.x` (which, recall, starts out as `NULL`), with the function `varPosn()` translating the name of each such variable into its position within the vector of numeric variable names (indicated by the second argument, `"numeric"`, to `varPosn()`).

- A Tk frame widget, named `optionsFrame`, is created to hold the two sets of radio buttons, which in turn are created by the R Commander `radioButtons()` utility. This is a macro-like function that constructs a set of related radio buttons along with the Tcl/Tk infrastructure that supports them; the function doesn't return a useful value, but rather creates objects in the environment of the calling function, `correlationTest()`.

  The `name` argument establishes a name for the set of radio buttons, `buttons` provides names for the several buttons, `labels` specifies text labels for the buttons, which can (as here) be distinct from the names of the buttons, and `title` supplies a title for the set of radio buttons.

  The `initialValue` argument indicates which radio button is selected when the dialog opens, and this selection is taken from the list of initial values.

- As is typical, the local `onOK()` function is invoked when the user presses the *OK* button (or the *Apply* button) in the dialog.

  The Tcl variables `alternativeVariable` and `correlationVariable` were implicitly created by the calls to `radioButtons()` and hold the values of the currently selected radio buttons. These are extracted via the **tcltk** function `tclvalue()`.

  The call to `putDialog()` stores the current selections in the dialog so that these will be used as initial values if and when the dialog is reopened.[7]

  The `ActiveDataSet()` function returns the name of the active data set, which is used to compose a command in the form of a character string. For example, clicking *OK* in the dialog box in Figure 4.3 produces the command

```
with(Prestige, cor.test(education, income, alternative="two.sided",
    method="pearson"))
```

  This command is passed to the `doItAndPrint()` function which enters the command in the R Commander *R Script* and *R Markdown* tabs, and causes the command to be executed, directing the command and any printed output that results to the *Output* pane.

[5] See Section 4.2 for further information about how the R Commander saves state information.

[6] Actually, this is the name used by `putDialog()` (see below) to store the state information for the dialog, which by convention I set to the name of the callback function that creates the dialog.

[7] Recall that dialog state information is erased when the active data set changes or when the *Reset* button in the dialog is pressed by the user.

If the command produces error or warning messages, these appear in the R Commander *Messages* pane.[8]

- The call to `OKCancelHelp()` includes the arguments `reset="correlationTest"` and `apply="correlationTest"`, creating the *Reset* and *Apply* buttons in the dialog box. The first of these arguments causes state information saved under `"correlationTest"` to be deleted when the *Reset* argument is pressed, and the dialog to reopen in its pristine state. The second argument specifies that `correlationTest()` is to be recalled, reopening the dialog, when the *Apply* button is pressed, after `onOK()` is executed.
- Tk frame widgets for the sets of radio buttons inside `optionsFrame` are created automatically by the `radioButtons()` macro, and are automatically named `correlationsFrame` and `alternativeFrame`.

4.1.3 *Two-Way Table*: A Tabbed Dialog

In this section, I describe the `twoWayTable()` callback function, which creates the *tabbed* R Commander *Two-Way Table* dialog, reachable through *Statistics* > *Contingency tables* > *Two-way table...*. The dialog box includes two tabs, *Data* and *Statistics*, both of which are illustrated in Figure 4.5.

The active data set is the `Adler` data on experimenter effects in psychological research, used in Section 6.1.3 of the text to illustrate two-way analysis of variance. Selecting `expectation` as the row variable for the table, `instruction` as the column variable, and leaving the *Statistics* tab in its default state, produces the commands and associated output in Figure 4.6.

In addition to tabs, the *Two-Way Table* dialog illustrates two features that we haven't encountered previously:

- The *Data* tab includes an R Commander *Subset expression* text-box widget.
- The *Statistics* tab includes a set of check boxes.

In discussing the callback function that generates the dialog, shown in Figure 4.7, I'll stress these new features, and pass over those that we've seen previously. The code for this function is long, and is spread over three pages, but much of it is devoted to composing the text for the commands that the dialog generates. I'll pass over this part of the callback function as well.

The tabs in the *Two-Way Table* dialog are produced by the argument `useTabs=TRUE` in the call to `initializeDialog()` (in the first part of Figure 4.7). The names of the two tabs are given implicitly by the default value of the `tabs` argument, which is `tabs = c("dataTab", "optionsTab")`. To create tabs with different names, or to create more than two tabs, specify the `tabs` argument explicitly. Tk widgets can be placed in the tabs by making reference to `dataTab` and `optionsTab`; for example,

```
variablesFrame <- tkframe(dataTab)
```

creates a frame within the *Data* tab to hold the two variable listboxes in the dialog. In addition to creating the two tabs, `initializeDialog()` constructs a Tk notebook widget to contain them. The default name of this notebook (given by the `notebook` argument to `initializeDialog()`) is `notebook`.

The displayed titles of the tabs are given in the call to `dialogSuffix()` at the very end of the callback function `twoWayTable()` (i.e., in the third part of Figure 4.7), and are distinct from the object-names of the tabs:

[8] For more information about how the R Commander interacts with the R interpreter, see Section 4.3.

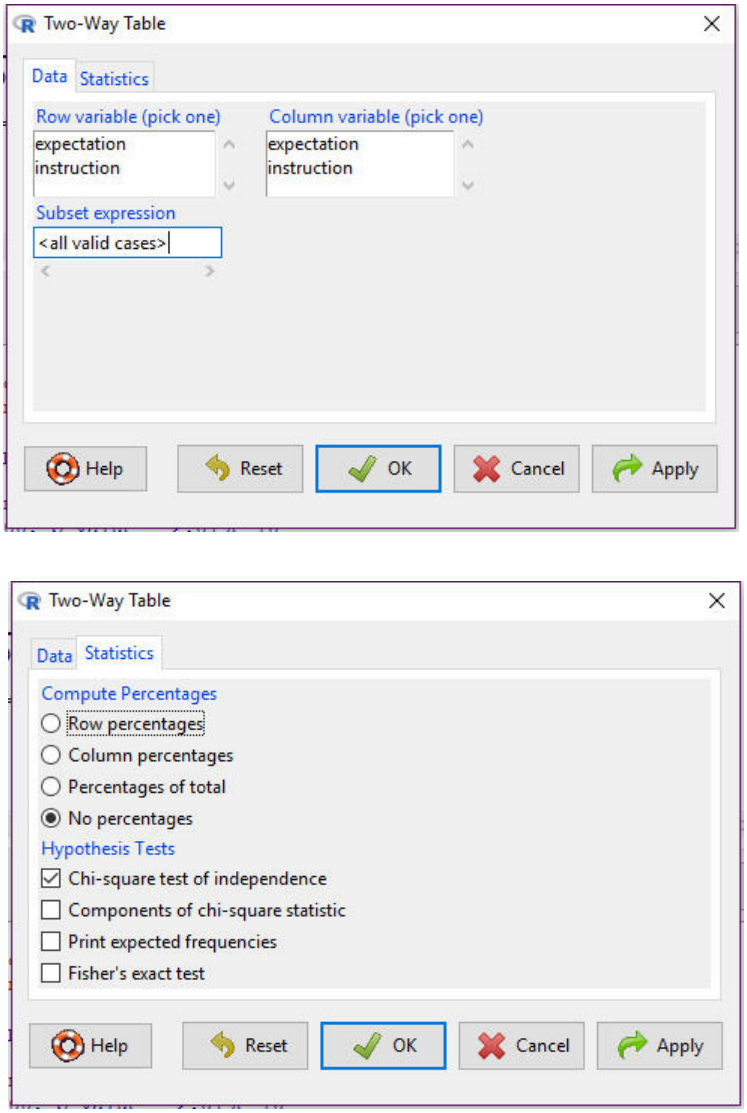

FIGURE 4.5: The *Data* and *Statistics* tabs in the *Two-Way Table* dialog.

```
> local({
+   .Table <- xtabs(~expectation+instruction, data=Adler)
+   cat("\nFrequency table:\n")
+   print(.Table)
+   .Test <- chisq.test(.Table, correct=FALSE)
+   print(.Test)
+ })

Frequency table:
           instruction
expectation GOOD NONE SCIENTIFIC
       HIGH   15   16         18
       LOW    17   18         13

        Pearson's Chi-squared test

data:  .Table
X-squared = 1.0389, df = 2, p-value = 0.5948
```

FIGURE 4.6: Commands and output produced by the *Two-Way Table* dialog.

```
dialogSuffix(use.tabs=TRUE, grid.buttons=TRUE,
             tab.names=c("Data", "Statistics"))
```

Thus the displayed name or label `"Data"` corresponds to `dataTab` and the displayed name `"Statistics"` corresponds to `optionsTab`. Notice that it's necessary to include the argument `use.tabs=TRUE` and `grid.buttons=TRUE`. The latter insures that the standard dialog buttons (*OK*, *Cancel*, and so on) appear properly below the tabs.[9] The `tabs.names` argument to `dialogSuffix()` is necessary here because the default displayed names for the two tabs are `"Data"` and `"Options"`.

The state information saved for the dialog includes the number of the tab (`0` or `1`—Tcl uses zero-based indexing) that's currently displayed. The default is `initial.tab=0`, that is, the *Data* tab. The name `initial.tab` must be used in the list of initial values because it's employed by `dialogSuffix()`, which as a macro-like function, has access to local variables in the environment of `twoWayTable()`. The currently visible tab is retrieved by `if (as.character(tkselect(notebook)) == dataTab$ID) 0 else 1` in the local `onOK()` function.

The call to `subsetBox()` creates a text box inside `dataTab`, with frame, title *Subset expression*, and initial contents `<all valid cases>`. The Tcl variable `subsetVariable` reports the contents of the *Subset expression* text box when the user presses the *OK* or *Apply* button in the dialog.

As mentioned, another new feature of the *Two-Way Table* dialog is the set of check boxes, which are created by the `checkBoxes()` R Commander utility macro function. Usage of this function is similar in many respects to `radioButtons()`:

- Unlike radio buttons, however, check boxes are independent of one-another, in that each may be checked or unchecked. Consequently, each check box has its own initial value,

[9] It may appear as if the `grid.buttons` argument is redundant, in that its value may be inferred from `use.tabs=TRUE`, but this is not the case due to scoping issues arising in macro-like functions such as `dialogSuffix()`.

```
twoWayTable <- function(){
    Library("abind")
    defaults <- list(initial.row=NULL, initial.column=NULL,
        initial.percents="none", initial.chisq=1, initial.chisqComp=0, initial.expected=0,
        initial.fisher=0, initial.subset=gettextRcmdr("<all valid cases>"), initial.tab=0)
    dialog.values <- getDialog("twoWayTable", defaults)
    initializeDialog(title=gettextRcmdr("Two-Way Table"), use.tabs=TRUE)
    variablesFrame <- tkframe(dataTab)
    .factors <- Factors()
    rowBox <- variableListBox(variablesFrame, .factors,
        title=gettextRcmdr("Row variable (pick one)"),
        initialSelection=varPosn(dialog.values$initial.row, "factor"))
    columnBox <- variableListBox(variablesFrame, .factors,
        title=gettextRcmdr("Column variable (pick one)"),
        initialSelection=varPosn(dialog.values$initial.column, "factor"))
    subsetBox(dataTab, subset.expression=dialog.values$initial.subset)
    onOK <- function(){
        tab <- if (as.character(tkselect(notebook)) == dataTab$ID) 0 else 1
        row <- getSelection(rowBox)
        column <- getSelection(columnBox)
        percents <- tclvalue(percentsVariable)
        chisq <- tclvalue(chisqTestVariable)
        chisqComp <- tclvalue(chisqComponentsVariable)
        expected <- tclvalue(expFreqVariable)
        fisher <- tclvalue(fisherTestVariable)
        initial.subset <- subset <- tclvalue(subsetVariable)
        subset <- if (trim.blanks(subset) == gettextRcmdr("<all valid cases>")) ""
        else paste(", subset=", subset, sep="")
        putDialog("twoWayTable", list(
            initial.row=row,
            initial.column=column,
            initial.percents=percents, initial.chisq=chisq, initial.chisqComp=chisqComp,
            initial.expected=expected, initial.fisher=fisher, initial.subset=initial.subset,
            initial.tab=tab))
        if (length(row) == 0 || length(column) == 0){
            errorCondition(recall=twoWayTable,
                message=gettextRcmdr("You must select two variables."))
            return()
        }
        if (row == column) {
            errorCondition(recall=twoWayTable,
                message=gettextRcmdr("Row and column variables are the same."))
            return()
        }
        closeDialog()
```

FIGURE 4.7: The `twoWayTable()` callback function (part 1)

```
        command <- paste("local({\n  .Table <- xtabs(~", row, "+", column, ",
            data=", ActiveDataSet(), subset,
            ')\n  cat("\\nFrequency table:\\n")\n  print(.Table)', sep="")
        command.2 <- paste("local({\n  .warn <- options(warn=-1)\n
            .Table <- xtabs(~", row, "+", column, ", data=", ActiveDataSet(),
            subset, ")", sep="")
        if (percents == "row")
          command <- paste(command,
              '\n  cat("\\nRow percentages:\\n")\n  print(rowPercents(.Table))',
              sep="")
        else if (percents == "column")
          command <-  paste(command,
              '\n  cat("\\nColumn percentages:\\n")\n print(colPercents(.Table))', sep="")
        else if (percents == "total")
          command <- paste(command,
              '\n  cat("\\nTotal percentages:\\n")\n  print(totPercents(.Table))', sep="")
        if (chisq == 1) {
            command <- paste(command,
                "\n  .Test <- chisq.test(.Table, correct=FALSE)", sep="")
            command.2 <- paste(command.2,
                "\n  .Test <- chisq.test(.Table, correct=FALSE)", sep="")
            command <- paste(command, "\n  print(.Test)", sep="")
            if (expected == 1)
                command <- paste(command,
                    '\n  cat("\\nExpected counts:\\n")\n  print(.Test$expected)', sep="")
            if (chisqComp == 1) {
             command <- paste(command,
              '\n  cat("\\nChi-square components:\\n")\n  print(round(.Test$residuals^2, 2))',
                  sep="")
            }
        }
        if (fisher == 1) command <- paste(command, "\n  print(fisher.test(.Table))")
        command <- paste(command, "\n})", sep="")
        doItAndPrint(command)
        if (chisq == 1){
          command.2 <- paste(command.2,
              "\nputRcmdr('.expected.counts', .Test$expected)\n options(.warn)\n})")
          justDoIt(command.2)
          warnText <- NULL
          expected <- getRcmdr(".expected.counts")
          if (0 < (nlt1 <- sum(expected < 1)))
              warnText <- paste(nlt1,
                  gettextRcmdr("expected frequencies are less than 1"))
          if (0 < (nlt5 <- sum(expected < 5)))
              warnText <- paste(warnText, "\n", nlt5,
                  gettextRcmdr(" expected frequencies are less than 5"), sep="")
          if (!is.null(warnText)) Message(message=warnText, type="warning")
        }
        tkfocus(CommanderWindow())
    }
```

FIGURE 4.7: The `twoWayTable()` callback function (part 2)

```
    OKCancelHelp(helpSubject="xtabs", reset="twoWayTable", apply="twoWayTable")
    radioButtons(optionsTab, name="percents",
        buttons=c("rowPercents", "columnPercents", "totalPercents", "nonePercents"),
        values=c("row", "column", "total", "none"),
            initialValue=dialog.values$initial.percents,
        labels=gettextRcmdr(c("Row percentages", "Column percentages",
            "Percentages of total", "No percentages")),
        title=gettextRcmdr("Compute Percentages"))
    checkBoxes(optionsTab, frame="testsFrame", boxes=c("chisqTest", "chisqComponents",
            "expFreq", "fisherTest"),
        initialValues=c(dialog.values$initial.chisq, dialog.values$initial.chisqComp,
            dialog.values$initial.expected, dialog.values$initial.fisher),
        labels=gettextRcmdr(c("Chi-square test of independence",
            "Components of chi-square statistic",
            "Print expected frequencies", "Fisher's exact test")))
    tkgrid(getFrame(rowBox), labelRcmdr(variablesFrame, text="    "),
        getFrame(columnBox), sticky="nw")
    tkgrid(variablesFrame, sticky="w")
    tkgrid(percentsFrame, sticky="w")
    tkgrid(labelRcmdr(optionsTab, text=gettextRcmdr("Hypothesis Tests"),
        fg=getRcmdr("title.color"), font="RcmdrTitleFont"), sticky="w")
    tkgrid(testsFrame, sticky="w")
    tkgrid(subsetFrame, sticky="w")
    dialogSuffix(use.tabs=TRUE, grid.buttons=TRUE, tab.names=c("Data", "Statistics"))
}
```

FIGURE 4.7: The `twoWayTable()` callback function (part 3, concluded).

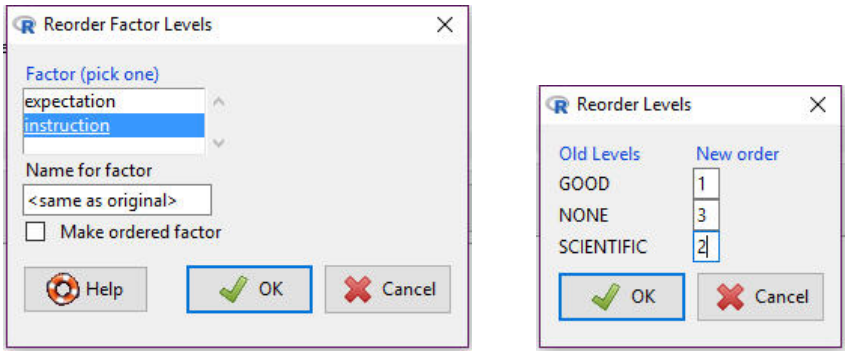

FIGURE 4.8: The *Reorder Factor Levels* main dialog (left) and *Reorder Levels* sub-dialog (right).

given by the `initialValues` argument to `checkBoxes()`, where `1` means checked and `0` unchecked.

- Each check box is given a name via the `boxes` argument to `checkBoxes()`, with an associated Tcl variable indicting the current state of the check box. Thus, for example, `tclvalue(chisqTestVariable)` is `1` (actually, `"1"`) if the `chisqTest` check box is checked and `0` if it's unchecked.
- The `frame` argument to `checkBoxes()` provides a name (here `"testsFrame"`) for the Tk frame containing the check boxes, which is used by `tkgrid()` to place the set of check boxes and their title in the dialog.

4.1.4 *Reorder Factor Levels*: A Dialog With a Subdialog

Shown at the left of Figure 4.8, the *Reorder Factor Levels* dialog is a simple untabbed dialog with a variable listbox, a text box for the name of the factor to be created, and a single checkbox, unchecked by default, to make the new variable an ordered factor. In the illustrative dialog, I click on `instruction` to select it in the variable listbox.

All of the elements of the *Reorder Factor Levels* dialog are familiar, although the single check box uses the Tk *themed widget* function `ttkcheckbutton()` directly rather than calling the R Commander `checkBoxes()` macro.[10] The code for the callback function `reorderFactor()`, which creates the dialog, appears in Figure 4.9, which is divided across two pages. The most important new feature of this dialog is that it invokes a sub-dialog when the user clicks the *OK* button.

As before, I'll concentrate on the as-yet unfamiliar features of the `reorderFactor()` callback function:

- In the local `onOK()` function, the command

```
old.levels <- eval(parse(text=paste("levels(", .activeDataSet,
                "$", variable, ")", sep="")), envir=.GlobalEnv)
```

[10] As a general matter, the R Commander uses Tk themed widgets supplied by the **tcltk** package, when they are available, and picks themes that are compatible with the various computing platforms (Windows, Mac OS X, Linux/Unix). In the **tcltk** package, functions producing themed widgets have names beginning begin with "`ttk`."

```
reorderFactor <- function(){
  initializeDialog(title=gettextRcmdr("Reorder Factor Levels"))
  variableBox <- variableListBox(top, Factors(),
    title=gettextRcmdr("Factor (pick one)"))
  orderedFrame <- tkframe(top)
  orderedVariable <- tclVar("0")
  orderedCheckBox <- ttkcheckbutton(orderedFrame, variable=orderedVariable)
  factorName <- tclVar(gettextRcmdr("<same as original>"))
  factorNameField <- ttkentry(top, width="20", textvariable=factorName)
  onOK <- function(){
    variable <- getSelection(variableBox)
    closeDialog()
    if (length(variable) == 0) {
      errorCondition(recall=reorderFactor,
        message=gettextRcmdr("You must select a variable."))
      return()
    }
    name <- trim.blanks(tclvalue(factorName))
    if (name == gettextRcmdr("<same as original>")) name <- variable
    if (!is.valid.name(name)){
      errorCondition(recall=reorderFactor,
                     message=paste('"', name, '" ',
                       gettextRcmdr("is not a valid name."), sep=""))
      return()
    }
    if (is.element(name, Variables())) {
      if ("no" == tclvalue(checkReplace(name))){
        reorderFactor()
        return()
      }
    }
    .activeDataSet <- ActiveDataSet()
    old.levels <- eval(parse(text=paste("levels(", .activeDataSet, "$", variable, ")",
                                        sep="")), envir=.GlobalEnv)
    nvalues <- length(old.levels)
    ordered <- tclvalue(orderedVariable)
    if (nvalues > 30) {
      errorCondition(recall=reorderFactor,
                     message=sprintf(gettextRcmdr("Number of levels (%d) too large."),
                     nvalues))
      return()
    }
```

FIGURE 4.9: The `reorderFactor()` callback function (part 1). The code for the function is edited slightly.

```
    initializeDialog(subdialog, title=gettextRcmdr("Reorder Levels"))
    order <- 1:nvalues
    onOKsub <- function() {
      closeDialog(subdialog)
      opt <- options(warn=-1)
      for (i in 1:nvalues){
        order[i] <- as.numeric(eval(parse(text=paste("tclvalue(levelOrder", i, ")",
                                                sep=""))))
      }
      options(opt)
      if (any(sort(order) != 1:nvalues) || any(is.na(order))){
        errorCondition(recall=reorderFactor,
        message=paste(gettextRcmdr("Order of levels must include all integers from 1 to "),
          nvalues, sep=""))
        return()
      }
      levels <- old.levels[order(order)]
      ordered <- if (ordered == "1") ", ordered=TRUE" else ""
      command <- paste("with(", .activeDataSet, ", factor(", variable,
                       ", levels=c(", paste(paste("'", levels, "'", sep=""),
                                            collapse=","), ")",
                       ordered, "))", sep="")
      result <- justDoIt(paste(.activeDataSet, "$", name, " <- ", command, sep=""))
      logger(paste(.activeDataSet,"$", name," <- ", command, sep=""))
      if (class(result)[1] !=  "try-error") activeDataSet(.activeDataSet,
        flushModel=FALSE, flushDialogMemory=FALSE)
    }
    subOKCancelHelp()
    tkgrid(labelRcmdr(subdialog, text=gettextRcmdr("Old Levels"),
             fg=getRcmdr("title.color"), font="RcmdrTitleFont"),
           labelRcmdr(subdialog, text=gettextRcmdr("New order"),
             fg=getRcmdr("title.color"), font="RcmdrTitleFont"), sticky="w")
    for (i in 1:nvalues){
      valVar <- paste("levelOrder", i, sep="")
      assign(valVar, tclVar(i))
      assign(paste("entry", i, sep=""), ttkentry(subdialog, width="2",
                                                 textvariable=get(valVar)))
      tkgrid(labelRcmdr(subdialog, text=old.levels[i]),
        get(paste("entry", i, sep="")), sticky="w")
    }
    tkgrid(subButtonsFrame, sticky="w", columnspan=2)
    dialogSuffix(subdialog, focus=entry1, force.wait=TRUE)
  }
  OKCancelHelp(helpSubject="factor")
  tkgrid(getFrame(variableBox), sticky="nw")
  tkgrid(labelRcmdr(top, text=gettextRcmdr("Name for factor")), sticky="w")
  tkgrid(factorNameField, sticky="w")
  tkgrid(orderedCheckBox, labelRcmdr(orderedFrame,
                           text=gettextRcmdr("Make ordered factor")), sticky="w")
  tkgrid(orderedFrame, sticky="w")
  tkgrid(buttonsFrame, sticky="w")
  dialogSuffix(preventGrabFocus=TRUE)
}
```

FIGURE 4.9: The `reorderFactor()` callback function (part 2, concluded).

returns a vector of level names for the factor to be recoded. These levels are used in the sub-dialog box (at the right of Figure 4.8).

- The sub-dialog is constructed within `onOK()`, in much the same manner as a menu-item callback function:
  - The function `initializeDialog()` is used in the same way as in the main dialog, except that rather than letting the name of the top-level Tk window default to `top`, I specify `subdialog` in the first argument to `initializeDialog()`.
  - The local function `onOKsub()`, defined similarly to `onOK()`, is invoked when the user presses the *OK* button in the sub-dialog.
  - The R Commander utility `subOKCancelHelp()` is used to create the *OK* and *Cancel* buttons in the sub-dialog. Because the `helpSubject` argument is absent, no *Help* button is supplied.
  - The command

    ```
    tkgrid(subButtonsFrame, sticky="w", columnspan=2)
    ```

    inserts the *OK* and *Cancel* buttons into the sub-dialog; and the command

    ```
    dialogSuffix(subdialog, focus=entry1, force.wait=TRUE)
    ```

    places the cursor in the first text-entry field in the *New order* column, and, by `force.wait=TRUE`, causes the main-dialog code to wait until the sub-dialog is closed, allowing the main dialog to compose the command to change the order of the factor levels based on the values that the user enters into the sub-dialog (as illustrated in Figure 4.10).

- The creation of the table in the sub-dialog, showing the *Old levels* of the factor in the first column and the corresponding *New order* of the factor levels in the second column, is a bit tricky; the code for this table is in the loop

```
for (i in 1:nvalues){
  valVar <- paste("levelOrder", i, sep="")
  assign(valVar, tclVar(i))
  assign(paste("entry", i, sep=""), ttkentry(subdialog, width="2",
    textvariable=get(valVar)))
  tkgrid(labelRcmdr(subdialog, text=old.levels[i]),
    get(paste("entry", i, sep="")), sticky="w")
}
```

  - Each time through the loop, `valVar` holds the name of a Tcl variable to be created; e.g., when the loop index `i` is `1`, this variable is named `"levelOrder1"`, and it is assigned the value `1` (more generally `i`) via the call to `tclVar(i)`.
  - Similarly, the variable named `"entry1"` is assigned a themed Tk entry widget for which `"levelOrder1"` is the associated Tcl variable.
  - The call to `tkgrid()` within the loop inserts each line of the table into the sub-dialog box.

```
> Adler$instruction <- with(Adler, factor(instruction, levels=c('GOOD',
+   'SCIENTIFIC','NONE')))
```

FIGURE 4.10: Command generated by the *Reorder Factor Levels* dialog.

- In `onOKsub()`, the loop

```
for (i in 1:nvalues){
  order[i] <- as.numeric(eval(parse(text=
                paste("tclvalue(levelOrder", i, ")", sep=""))))
}
```

  extracts the new order of the factor levels from the Tcl variables `levelOrder1`, `levelOrder2`, etc.

- Because the `Adler` data set is modified, `onOK()` executives the command

```
activeDataSet(.activeDataSet, flushModel=FALSE, flushDialogMemory=FALSE)
```

  to refresh the active data set (where `.activeDataSet` holds the name of the active data set, `"Adler"`). Here, `flushModel=FALSE, flushDialogMemory=FALSE` avoids clearing ("flushing") the active statistical model (if there is one) and the saved states of dialogs, as happens by default when the active data set changes. These operations are unnecessary, and undesirable, when the only change is to the order of the levels of a factor.

4.1.5 *Histogram*: A Dialog That Uses the R Commander *Plot by* Widget

The final illustrative dialog, *Histogram*, shown in Figure 4.11, demonstrates the use of the R Commander *Plot by* button-widget, provided by the `groupsBox()` utility. Figure 4.11 shows an example of the dialog using the `Adler` data set. Because there's only one numeric variable in the data set, `rating`, it's preselected in the variable listbox at the top of the figure, which shows the initial state of the dialog. Pressing the *Plot by groups...* button brings up the sub-dialog in the center of the figure. Selecting `instruction` in the *Groups variable* list and clicking *OK* in the *Groups* sub-dialog changes the button in the main dialog to *Plot by: instruction*, as shown at the bottom of the figure. Clicking *OK* in the main dialog produces the group-wise histograms at the bottom of Figure 4.12.

The code for the callback function `Histogram()` appears in Figure 4.13, which is spread across three pages, but the snippet of code producing the *Plot by* button is very simple, and mostly self-explanatory:

```
groupsBox(Histogram, initialGroup=initial.group,
    initialLabel=if (is.null(initial.group)) gettextRcmdr("Plot by groups")
    else paste(gettextRcmdr("Plot by:"), initial.group), window=dataTab)
```

The first argument, `recall`, set to `Histogram`, causes `Histogram()` to be recalled to reopen the dialog if there's an error. The `groupsBox()` macro assigns the groups-variable selection to the variable `.groups` in the environment of `Histogram()`, setting this variable to `FALSE` if no groups-variable is selected. The variable `.groups` is then used to compose a call to the `Hist()` function, shown at the top of Figure 4.12.

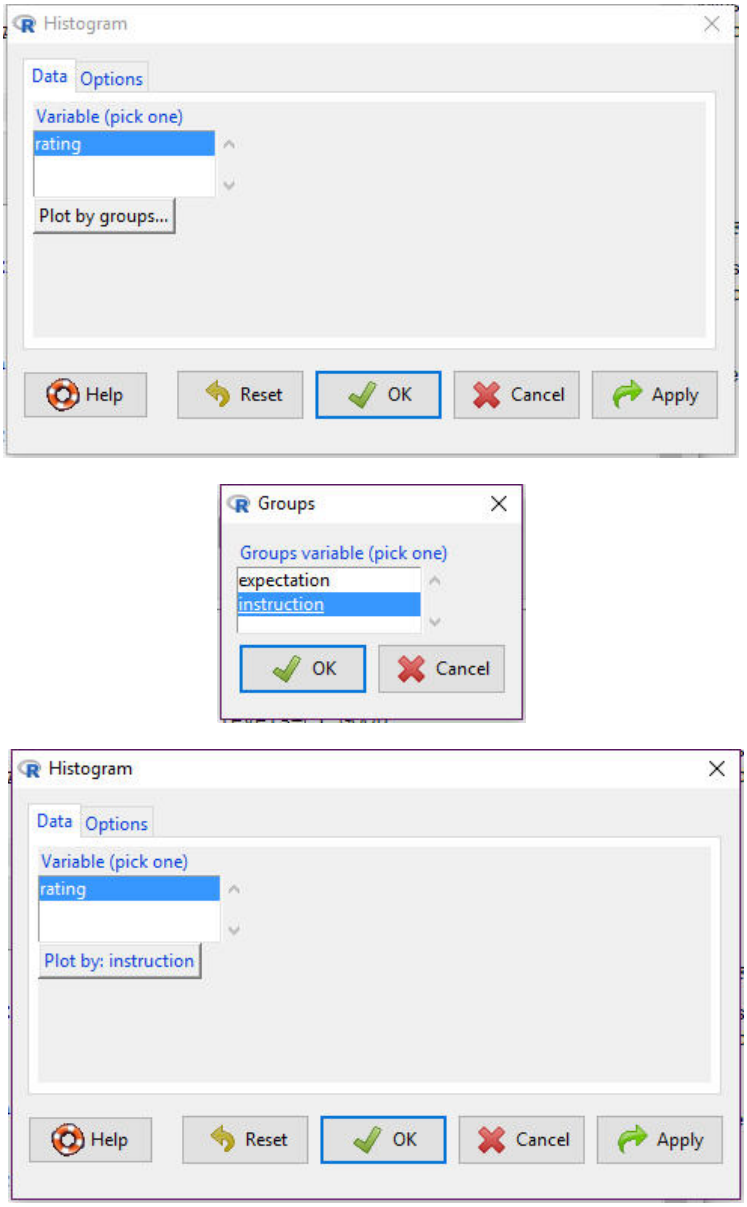

FIGURE 4.11: *Histogram* dialog box: initial state (top); *Groups* sub-dialog (center); after groups selection (botton). Only the *Data* tab in the main dialog is shown.

```
> with(Adler, Hist(rating, groups=instruction, scale="frequency",
+   breaks="Sturges", col="darkgray"))
```

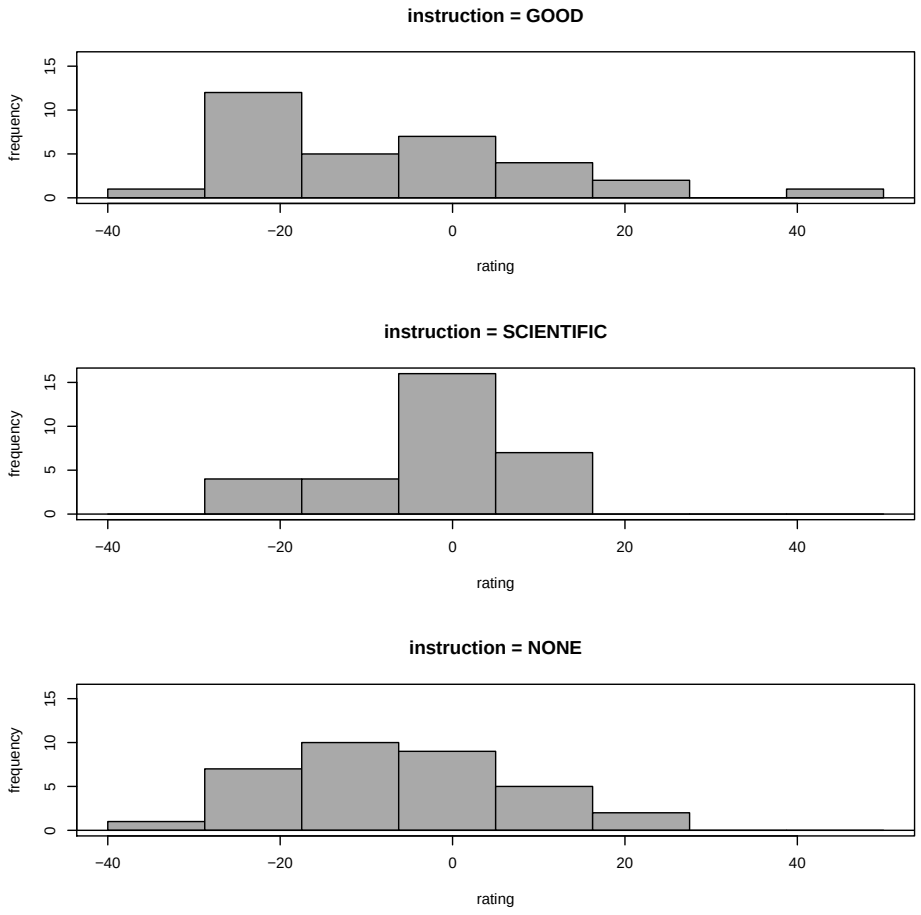

FIGURE 4.12: `Hist()` command (top) and histograms by group (bottom) produced by the *Histogram* dialog.

```
Histogram <- function () {
    defaults <- list(initial.x = NULL, initial.scale = "frequency",
        initial.bins = gettextRcmdr ("<auto>"), initial.tab=0,
        initial.xlab=gettextRcmdr("<auto>"), initial.ylab=gettextRcmdr("<auto>"),
        initial.main=gettextRcmdr("<auto>"), initial.group = NULL)
    dialog.values <- getDialog("Histogram", defaults)
    initializeDialog(title = gettextRcmdr("Histogram"), use.tabs=TRUE)
    xBox <- variableListBox(dataTab, Numeric(), title = gettextRcmdr("Variable (pick one)"),
        initialSelection = varPosn (dialog.values$initial.x, "numeric"))
    initial.group <- dialog.values$initial.group
    .groups <- if (is.null(initial.group)) FALSE else initial.group
    onOK <- function() {
        tab <- if (as.character(tkselect(notebook)) == dataTab$ID) 0 else 1
        x <- getSelection(xBox)
        closeDialog()
        if (length(x) == 0) {
            errorCondition(recall = Histogram,
              message = gettextRcmdr("You must select a variable"))
            return()
        }
        bins <- tclvalue(binsVariable)
        opts <- options(warn = -1)
        binstext <- if (bins == gettextRcmdr("<auto>"))
            "\"Sturges\""
        else as.numeric(bins)
        options(opts)
        scale <- tclvalue(scaleVariable)
        xlab <- trim.blanks(tclvalue(xlabVar))
        xlab <- if (xlab == gettextRcmdr("<auto>"))
            ""
        else paste(", xlab=\"", xlab, "\"", sep = "")
        ylab <- trim.blanks(tclvalue(ylabVar))
        ylab <- if (ylab == gettextRcmdr("<auto>"))
            ""
        else paste(", ylab=\"", ylab, "\"", sep = "")
        main <- trim.blanks(tclvalue(mainVar))
        main <- if (main == gettextRcmdr("<auto>"))
            ""
        else paste(", main=\"", main, "\"", sep = "")
        putDialog ("Histogram", list (initial.x = x, initial.bins = bins,
             initial.scale = scale,
            initial.tab=tab, initial.xlab=tclvalue(xlabVar),
            initial.ylab = tclvalue(ylabVar),
            initial.main = tclvalue(mainVar),
            initial.group=if (.groups == FALSE) NULL else .groups))
```

FIGURE 4.13: The `Histogram()` callback function (part 1). The function is edited slightly.

```
        if (is.null(.groups) || .groups == FALSE) {
            command <- paste("with(", ActiveDataSet(), ",
                    Hist(", x, ', scale="', scale, '", breaks=',
                binstext, ', col="darkgray"', xlab, ylab, main, "))", sep="")
        }
        else{
            command <- paste("with(", ActiveDataSet(),
                ", Hist(", x, ", groups=", .groups, ', scale="',
                scale, '", breaks=', binstext, ', col="darkgray"',
              xlab, ylab, main, "))", sep="")
        }
        doItAndPrint(command)
        activateMenus()
        tkfocus(CommanderWindow())
    }
    groupsBox(Histogram, initialGroup=initial.group,
        initialLabel=if (is.null(initial.group)) gettextRcmdr("Plot by groups")
        else paste(gettextRcmdr("Plot by:"), initial.group), window=dataTab)
    OKCancelHelp(helpSubject = "Hist", reset = "Histogram", apply="Histogram")
    optionsFrame <- tkframe(optionsTab)
    optFrame <- ttklabelframe(optionsFrame, labelwidget=tklabel(optionsFrame,
        text = gettextRcmdr("Plot Options"),
        font="RcmdrTitleFont", foreground=getRcmdr("title.color")))
    parFrame <- ttklabelframe(optionsFrame, labelwidget=tklabel(optionsFrame,
        text = gettextRcmdr("Plot Labels"),
        font="RcmdrTitleFont", foreground=getRcmdr("title.color")))
    xlabVar <- tclVar(dialog.values$initial.xlab)
    ylabVar <- tclVar(dialog.values$initial.ylab)
    mainVar <- tclVar(dialog.values$initial.main)
    xlabEntry <- ttkentry(parFrame, width = "25", textvariable = xlabVar)
    xlabScroll <- ttkscrollbar(parFrame, orient = "horizontal",
        command = function(...) tkxview(xlabEntry, ...))
    tkconfigure(xlabEntry, xscrollcommand = function(...) tkset(xlabScroll,
        ...))
    tkgrid(labelRcmdr(parFrame, text = gettextRcmdr("x-axis label")),
        xlabEntry, sticky = "ew", padx=6)
    tkgrid(labelRcmdr(parFrame, text =""), xlabScroll, sticky = "ew", padx=6)
    ylabEntry <- ttkentry(parFrame, width = "25", textvariable = ylabVar)
    ylabScroll <- ttkscrollbar(parFrame, orient = "horizontal",
        command = function(...) tkxview(ylabEntry, ...))
    tkconfigure(ylabEntry, xscrollcommand = function(...) tkset(ylabScroll,
        ...))
    tkgrid(labelRcmdr(parFrame, text = gettextRcmdr("y-axis label")),
        ylabEntry, sticky = "ew", padx=6)
    tkgrid(labelRcmdr(parFrame, text =""), ylabScroll, sticky = "ew", padx=6)
```

FIGURE 4.13: The `Histogram()` callback function (part 2).

```
    mainEntry <- ttkentry(parFrame, width = "25", textvariable = mainVar)
    mainScroll <- ttkscrollbar(parFrame, orient = "horizontal",
        command = function(...) tkxview(mainEntry, ...))
    tkconfigure(mainEntry, xscrollcommand = function(...) tkset(mainScroll,
        ...))
    tkgrid(labelRcmdr(parFrame, text = gettextRcmdr("Graph title")),
        mainEntry, sticky = "ew", padx=6)
    tkgrid(labelRcmdr(parFrame, text=""), mainScroll, sticky = "ew", padx=6)
    axisFrame <- tkframe(optFrame)
    radioButtons(axisFrame, name = "scale", buttons = c("frequency", "percent",
        "density"), labels = gettextRcmdr(c("Frequency counts",
            "Percentages", "Densities")), title = gettextRcmdr("Axis Scaling"),
        initialValue = dialog.values$initial.scale)
    binsFrame <- tkframe(optFrame)
    binsVariable <- tclVar(dialog.values$initial.bins)
    binsField <- ttkentry(binsFrame, width = "8", textvariable = binsVariable)
    tkgrid(getFrame(xBox), sticky = "nw")
    tkgrid(groupsFrame, sticky = "w")
    tkgrid(labelRcmdr(binsFrame, text = gettextRcmdr("Number of bins: ")),
        binsField, sticky = "w")
    tkgrid(binsFrame, sticky = "w")
    tkgrid(scaleFrame, sticky = "w")
    tkgrid(axisFrame, sticky = "w")
    tkgrid.configure(binsField, sticky = "e")
    tkgrid(optFrame, parFrame, sticky = "nswe", padx=6, pady=6)
    tkgrid(optionsFrame, sticky = "w")
    dialogSuffix(use.tabs=TRUE, grid.buttons=TRUE)
}
```

FIGURE 4.13: The `Histogram()` callback function (part 3, concluded).

4.2 Saving and Retrieving State Information

The R Commander saves a variety state information in the `.RcmdrEnv` environment, which isn't exported from the **Rcmdr** package. As I'll explain, however, you can interact with this environment through a number of functions that can retrieve information from the `.RcmdrEnv` environment and can place information in it.

The specific contents of `.RcmdrEnv` will vary from session to session. Currently, as I'm writing this chapter, the environment contains the following objects:

```
> objects(envir=Rcmdr:::.RcmdrEnv, all.names=TRUE)
  [1] ".activeDataSet"            ".activeModel"
  [3] ".expected.counts"          "ask.on.exit"
  [5] "ask.to.exit"               "attach.data.set"
  [7] "autoRestart"               "cancelDialogReopen"
  [9] "capabilities"              "command.text.color"
 [11] "commanderWindow"           "commandStack"
 [13] "console.output"            "crisp.dialogs"
 [15] "dataSetLabel"              "dataSetName"
 [17] "default.contrasts"         "default.font.family"
 [19] "default.font.size"         "dialog.values"
 [21] "dialog.values.noreset"     "double.click"
 [23] "editDataset.threshold"     "error.text.color"
 [25] "etc"                       "etcMenus"
 [27] "factors"                   "grab.focus"
 [29] "help_type"                 "Identify3d"
 [31] "installed.packages"        "knitr.editor.open"
 [33] "knitr.output"              "last.message"
 [35] "last.search"               "length.command.stack"
 [37] "length.output.stack"       "log.commands"
 [39] "log.font.family"           "log.font.size"
 [41] "log.height"                "log.text.color"
 [43] "log.width"                 "logFileName"
 [45] "logFont"                   "logWindow"
 [47] "Markdown.editor.open"      "markdown.output"
 [49] "Menus"                     "messageNumber"
 [51] "messages.height"           "messagesWindow"
 [53] "modelClasses"              "modelLabel"
 [55] "modelName"                 "modelNumber"
 [57] "multiple.select.mode"      "ncol"
 [59] "nrow"                      "number.messages"
 [61] "numeric"                   "open.dialog.here"
 [63] "open.showData.windows"     "output.height"
 [65] "output.text.color"         "outputFileName"
 [67] "outputStack"               "outputWindow"
 [69] "prefixes"                  "quit.R.on.close"
 [71] "quotes"                    "RcmdrVersion"
 [73] "reset.model"               "restore.device"
 [75] "restore.help_type"         "restoreTab"
 [77] "retain.messages"           "retain.selections"
 [79] "rgl"                       "rgl.command"
```

```
 [81] "rmd.generated"             "rmd.output.format"
 [83] "rmd.template"              "RmdFileName"
 [85] "RmdWindow"                 "rnw.generated"
 [87] "rnw.template"              "RnwFileName"
 [89] "RnwWindow"                 "RStudio"
 [91] "savedTable"                "saveFileName"
 [93] "saveOptions"               "showData.threshold"
 [95] "sort.names"                "startNewCommandBlock"
 [97] "startNewKnitrCommandBlock" "suppress.icon.images"
 [99] "suppress.menus"            "suppress.X11.warnings"
[101] "tagNumber"                 "theme"
[103] "title.color"               "tkwait.dialog"
[105] "twoLevelFactors"           "use.knitr"
[107] "use.markdown"              "use.rgl"
[109] "variable.list.height"      "variable.list.width"
[111] "variables"                 "warning.text.color"
```

Some of these objects originate in the initialization of the R Commander and are possibly influenced by the user, who can set R Commander options (via the R command `options(Rcmdr=list(`*etc.*`))`). For example,

```
> get("ask.to.exit", envir=Rcmdr:::.RcmdrEnv)
[1] TRUE
```

indicates that when the user selects *File* > *Exit* > *from Commander* or clicks the X at the upper-right of the main R Commander window, an *Exit* dialog will open asking the user to confirm.

A better way to access objects in the `.RcmdrEnv` environment is through the function `getRcmdr()`:

```
> getRcmdr("ask.to.exit")
[1] TRUE
```

Similarly, `putRcmdr()` may be used to save arbitrary objects in the `.RcmdrEnv` environment; for example:

```
> putRcmdr("foo", "bar")
> getRcmdr("foo")
[1] "bar"
```

You should exercise some care to avoid name clashes when you place information in `.RcmdrEnv`, because you don't want to "clobber" (overwrite) an object that's already there:

- You could prefix saved objects with the name of your plug-in package; for example,

  ```
  > putRcmdr("RcmdrPlugin.foo_bar", "baz")
  ```

- You could maintain a list of saved information; for example (assuming that the list `RcmdrPlugin.foo` already exists in `.RcmdrEnv`):

  ```
  > RcmdrPlugin.foo <- getRcmdr("RcmdrPlugin.foo")
  > RcmdrPlugin.foo$bar <- "baz"
  > putRcmdr("RcmdrPlugin.foo")
  ```

- Finally, and perhaps most elegantly, you could maintain an unexported environment in your package, similar to `Rcmdr:::.RcmdrEnv`, for storing state information.

Some of the information stored in `.RcmdrEnv` pertains to the active data set, and is most conveniently retrieved by specialized *accessor* functions, some of which we've already encountered. For example, `Variables()` returns the names of all of the variables in the active data set, `Factors()` returns the names of the factors in the active data set, and `Numeric()` returns the names of the numeric variables in the active data set. Similarly, `ActiveDataSet()` returns the name of the active data set (or `NULL` if there is no active data set).

If they are called with an argument, these functions also serve to modify the state information for the active set. For example, `ActiveDataSet("Prestige")` changes the active data set to `Prestige` (and also resets the information about the variables in the active data set), if the `Prestige` data set resides in memory, and generates an error if it does not.

Saved state information about dialogs is stored in `dialog.values` as a list of lists, one sub-list for each dialog whose state is saved. Currently, for example:

```
> getRcmdr("dialog.values")
$Histogram
$Histogram$initial.x
[1] "rating"

$Histogram$initial.bins
[1] "<auto>"

$Histogram$initial.scale
[1] "frequency"

$Histogram$initial.tab
[1] 0

$Histogram$initial.xlab
[1] "<auto>"

$Histogram$initial.ylab
[1] "<auto>"

$Histogram$initial.main
[1] "<auto>"

$Histogram$initial.group
[1] "instruction"
```

Here, the outer list has only one element, for the `Histogram` dialog, reflecting the current R Commander session. More typically, state information would be saved for several dialogs. As we have seen, this information is normally retrieved by `getDialog()` and stored by `putDialog()`; for example:

```
> getDialog("Histogram")
$initial.x
[1] "rating"

$initial.bins
[1] "<auto>"

$initial.scale
```

```
[1] "frequency"

$initial.tab
[1] 0

$initial.xlab
[1] "<auto>"

$initial.ylab
[1] "<auto>"

$initial.main
[1] "<auto>"

$initial.group
[1] "instruction"

> getDialog("twoWayTable") # no state info currently saved
NULL
```

You'll find complete information about the available R Commander accessor functions in Appendix A.

4.3 How the R Commander Interacts With the R Interpreter

As we have seen, R Commander dialogs typically interact with the R interpreter by generating commands in the form of character strings, causing the commands to be executed by calling the function `doItAndPrint()`. Composing commands as character strings can be awkward, but it is a flexible arrangement.

The `doItAndPrint()` function has three arguments:

`command` is a character string containing the command to be executed. If the command is spread over several lines, each line (but the last) should end with a newline chararacter (i.e., `"\n"`).

`log` is a logical value, `TRUE` by default, indicating whether the command should be entered ("logged") into the R Commander *R Script* tab and in the R Commander *Output* pane, along with the printed output that it produces (which appears in any event in the *Output* pane).

`rmd` (for "R Markdown") is a logical value, defaulting to the value of `log`, controlling whether the command is to be entered into the R Commander *R Markdown* and *knitr* tabs (if these tabs exist).

It's normal to use `doItAndPrint()` for most commands generated by R Commander or plug-in dialogs, and it's rare to want to set `log` or `rmd` to `FALSE`. Output produced by the command is directed to the R Commander *Output* pane, and error and warning messages to the *Messages* pane.

In addition to `doItAndPrint()`, your dialog can call the `justDoIt()` or `logger()` functions. As for `doItAndPrint()`, the `command` argument to `justDoIt()` is an R command given as a character string. As its name implies, however, `justDoIt()` executes the command without entering it, or any associated output, into the *R Script* tab, the *R Markdown* tab, the *knitr* tab, or the *Output* pane. In contrast, the `logger()` function prints its `command` argument in the *R Script* tab, the *R Markdown* tab, the *knitr* tab, and the *Output* pane *without* actually executing the command. The `logger()` function has an optional argument, `rmd`, which defaults to `TRUE`, and which has the same effect as the `rmd` argument to `doItAndPrint()`.

5

Handling Statistical Models in R Commander Plug-in Packages

There are special considerations associated with building statistical modeling dialogs for the R Commander, partly to incorporate R model formulas in the dialogs, and partly to work properly with the menu items of the *Models* menu that reference an active statistical model. I'll explain in this chapter how statistical models are handled by the R Commander, primarily using the *Linear Model* dialog to illustrate, but also referencing the *Cox-Regression Model* dialog from the **RcmdrPlugin.survival** package. The latter example demonstrates how a plug-in package can add a class of statistical model objects to the R Commander.

Subsequent sections of the chapter discuss the R Commander *Models* menu and the `RcmdrModels:` field in the plug-in package `DESCRIPTION` file.

5.1 The *Linear Model* Dialog

The R Commander *Linear Model* dialog exemplifies the implementation of a statistical modeling dialog using various R Commander utility functions, including the `modelFormula()` function. The dialog (whose use is described in Section 7.2 and 7.3 of the text) appears in Figure 5.1. The `linearModel()` callback function, invoked by *Statistics* > *Fit models* > *Linear model...*, is shown in Figure 5.2 (which is divided over two pages).

The active data set in the *Linear Model* dialog in Figure 5.1 is the `Prestige` data (introduced in Section 4.2.2 of the text). To build the model formula in the dialog, I double-click on `prestige` in the variable listbox, double-click on `type`, (single-)click on `education` and then on the *natural spline* button, and finally click on `income` and the *natural spline* button. The other selections in the dialog—the name of the model, the spinners for splines and polynomials, and the *Subset expression* and *Weights* text boxes—are left at their defaults.

Many of the elements of the `linearModel()` callback function in Figure 5.2 are familiar, and, as has become my habit, I won't dwell on these, but rather will describe what's new:

- The `linearModel()` function largely bypasses the standard R Commander mechanism for preserving dialog-box state, which uses the functions `getDialog()` and `putDialog()`. The reason for this is historical, and it would be better to use the standard mechanism for a new statistical modeling dialog.

  The first few lines of the function, for example, reuse the formula from the previous statistical model, if it is a linear model. Similarly, the stored value `reset.model` is employed for the *Reset* button in the dialog box. It would currently be simpler to use `getDialog()` and `putDialog()` for these purposes. I retained the existing code in the **Rcmdr** package mainly to honor the advice that "if it ain't broke, don't fix it."

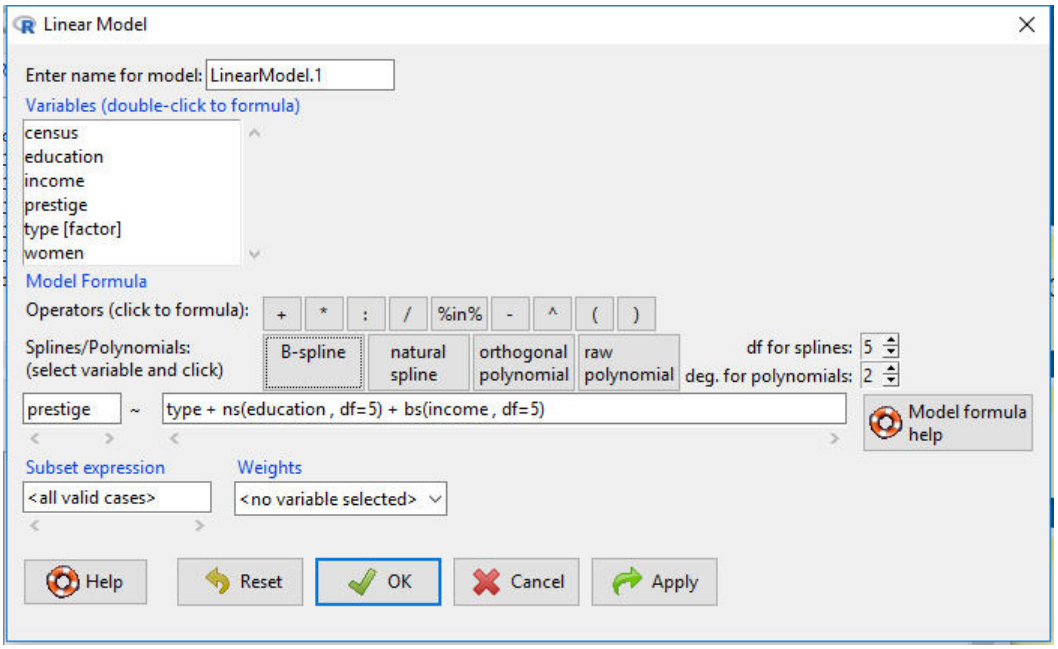

FIGURE 5.1: The R Commander *Linear Model* dialog box, specifying a model for the `Prestige` data.

- The R Commander numbers statistical models serially through a session. Calling `UpdateModelNumber()` increments the current model number by 1, and `getRcmdr("modelNumber")` retrieves the current model number.

  Notice that when `errorCondition()` is called in the local `onOK()` function for the dialog, the argument `model` is set to `TRUE`. This causes the model number to be decremented before the dialog is reopened, preventing numbers from being skipped.

- The key step in constructing the *Linear Model* dialog is the call to the `modelFormula()` macro; because all of the defaults are used in the dialog, `modelFormula()` is called without any arguments.

  `modelFormula()` provides the variable listbox, button bars, and left-hand-side and right-hand-side formula text boxes in the dialog. These elements are placed in the dialog box using `tkgrid()` with `getFrame(xBox)` for the variable listbox, `outerOperatorsFrame` for the button bars, and `formulaFrame` for the model formula.

- Weight-variable selection is implemented with a Tk *combo box* (i.e., a drop-down list, here of numeric variables), produced by the R Commander `variableComboBox()` macro, and the *Subset expression* text box is provided by the R Commander `subsetBox()` macro.

- The call to the `OKCancelHelp()` macro to produce the buttons at the bottom of the dialog is standard, with one exception: The `recall` argument is set to `"resetLinearModel"` rather than to the callback function `"linearModel"` directly. This is necessary to handle the nonstandard manner in which the dialog saves its state and isn't something to emulate.

```
linearModel <- function(){
  initializeDialog(title=gettextRcmdr("Linear Model"))
  defaults <- list(initial.weight = gettextRcmdr("<no variable selected>"))
  dialog.values <- getDialog("linearModel", defaults)
  .activeModel <- ActiveModel()
  currentModel <- if (!is.null(.activeModel))
    class(get(.activeModel, envir=.GlobalEnv))[1] == "lm"
  else FALSE
  if (currentModel) {
    currentFields <- formulaFields(get(.activeModel, envir=.GlobalEnv))
    if (currentFields$data != ActiveDataSet()) currentModel <- FALSE
  }
  if (isTRUE(getRcmdr("reset.model"))) {
    currentModel <- FALSE
    putRcmdr("reset.model", FALSE)
  }
  UpdateModelNumber()
  modelName <- tclVar(paste("LinearModel.", getRcmdr("modelNumber"), sep=""))
  modelFrame <- tkframe(top)
  model <- ttkentry(modelFrame, width="20", textvariable=modelName)
  onOK <- function(){
    modelValue <- trim.blanks(tclvalue(modelName))
    closeDialog()
    if (!is.valid.name(modelValue)){
      errorCondition(recall=linearModel,
        message=sprintf(gettextRcmdr('"%s" is not a valid name.'), modelValue),
        model=TRUE)
      return()
    }
    subset <- tclvalue(subsetVariable)
    if (trim.blanks(subset) ==
        gettextRcmdr("<all valid cases>") || trim.blanks(subset) == ""){
            subset <- ""
            putRcmdr("modelWithSubset", FALSE)
    }
    else{
      subset <- paste(", subset=", subset, sep="")
      putRcmdr("modelWithSubset", TRUE)
    }
    weight.var <- getSelection(weightComboBox)
    putDialog("linearModel", list(initial.weight = weight.var))
    weights <- if (weight.var == gettextRcmdr("<no variable selected>")) ""
        else paste(", weights=", weight.var, sep="")
```

FIGURE 5.2: The `linearModel()` callback function, slightly edited for clarity (part 1)

```
    check.empty <- gsub(" ", "", tclvalue(lhsVariable))
    if ("" == check.empty) {
      errorCondition(recall=linearModel,
        message=gettextRcmdr("Left-hand side of model empty."), model=TRUE)
      return()
    }
    check.empty <- gsub(" ", "", tclvalue(rhsVariable))
    if ("" == check.empty) {
      errorCondition(recall=linearModel,
        message=gettextRcmdr("Right-hand side of model empty."), model=TRUE)
      return()
    }
    if (is.element(modelValue, listLinearModels())) {
      if ("no" == tclvalue(checkReplace(modelValue, type=gettextRcmdr("Model")))){
        UpdateModelNumber(-1)
        linearModel()
        return()
      }
    }
    formula <- paste(tclvalue(lhsVariable), tclvalue(rhsVariable), sep=" ~ ")
    command <- paste("lm(", formula,
        ", data=", ActiveDataSet(), subset, weights, ")", sep="")
    doItAndPrint(paste(modelValue, " <- ", command, sep = ""))
    doItAndPrint(paste("summary(", modelValue, ")", sep=""))
    activeModel(modelValue)
    tkfocus(CommanderWindow())
  }
  OKCancelHelp(helpSubject="linearModel", model=TRUE,
    reset="resetLinearModel", apply="linearModel")
  tkgrid(labelRcmdr(modelFrame,
    text=gettextRcmdr("Enter name for model:")), model, sticky="w")
  tkgrid(modelFrame, sticky="w")
  modelFormula()
  subsetWeightFrame <- tkframe(top)
  subsetBox(window=subsetWeightFrame, model=TRUE)
  weightComboBox <- variableComboBox(subsetWeightFrame,
    variableList=Numeric(), initialSelection=dialog.values$initial.weight,
    title=gettextRcmdr("Weights"))
  tkgrid(getFrame(xBox), sticky="w")
  tkgrid(outerOperatorsFrame, sticky="w")
  tkgrid(formulaFrame, sticky="w")
  tkgrid(subsetFrame, tklabel(subsetWeightFrame, text="   "),
         getFrame(weightComboBox), sticky="nw")
  tkgrid(subsetWeightFrame, sticky="w")
  tkgrid(buttonsFrame, sticky="w")
  dialogSuffix(focus=lhsEntry, preventDoubleClick=TRUE)
}
```

FIGURE 5.2: The `linearModel()` callback function (part 2, concluded)

- In the call to `dialogSuffix()`, `focus=lhsEntry` places the cursor initially in the left-hand-side formula text box, and `preventDoubleClick=TRUE` prevents double-clicking from pressing the *OK* button in the dialog box, because double-clicks are used to transfer variables from the listbox to the model formula.
- The local `onOK` function references Tcl (text) variables created by the `subsetBox()` and `modelFormula()` macros: `subsetVariable` contains the subset expression; `lhsVariable` contains the expression defining the left-hand side of the model formula (i.e., an expression that evaluates to the response variable, and is usually just the name of the response variable); and `rhsVariable` contains the right-hand side of the model formula.

5.2 The *Cox-Regression Model* Dialog in the RcmdrPlugin.survival Package

For a second illustration of a statistical-modeling dialog box, I draw on the *Cox-Regression Model* dialog in the **RcmdrPlugin.survival** package, which is discussed in Section 9.3.1 of the text. Figure 5.3 shows an example of this dialog box in a fresh R Commander session in which the plug-in package is loaded. I use the `Rossi` criminal recidivism data set, also described in the text.

The dialog box includes two tabs: The *Data* tab is shown at the top of Figure 5.3, and the *Model* tab at the bottom. I select `week` as the time variable and `arrest` as the event indicator in the *Data* tab, leaving all other selections in the tab in their default states. In the *Model* tab, I build the right-hand side of the Cox regression model by double-clicking on the various predictors in the *Variables* box. For a Cox model, the left-hand side of the model is formulated from the time and event variables, producing the command

```
CoxModel.1 <- coxph(Surv(week, arrest) ~ age + educ + fin + mar + paro +
  prio + race + wexp, method="efron", data=Rossi)
```

when the *OK* button in the dialog is clicked. The other elements of the *Model* tab are also left in their default states.

The `CoxModel()` callback function, which is invoked by the menu selection *Statistics* > *Fit models* > *Cox regression model...* once the plug-in package is loaded, is displayed in part in Figure 5.4. This function is very long—its listing would spread over five pages if shown in its entirety—reflecting the large number of widgets in the two tabs of the dialog box. Most of these widgets—variable listboxes, sets of radio buttons, text boxes—are by now entirely familiar.

I include the callback function here only to show how to imbed a model formula in a tabbed dialog and how to handle a one-sided formula. Both of these aspects of the callback function are very simple, and are apparent in the first two arguments of the command

```
modelFormula(modelTab, hasLhs=FALSE, rhsExtras=TRUE)
```

The third argument, `rhsExtras=TRUE`, includes the splines and polynomials button bar in the dialog; the default for this argument is `rhsExtras=FALSE` for backwards compatibility with older versions of the **Rcmdr** package, on which some plug-ins may rely. The other lines in Figure 5.2 show how the tabs are handled in the `initializeDialog()` and `dialogSuffix()` commands, how the buttons at the bottom of the dialog are specified, and how the one-sided formula and associated button bars are placed in the dialog box via calls to `tkgrid()`.

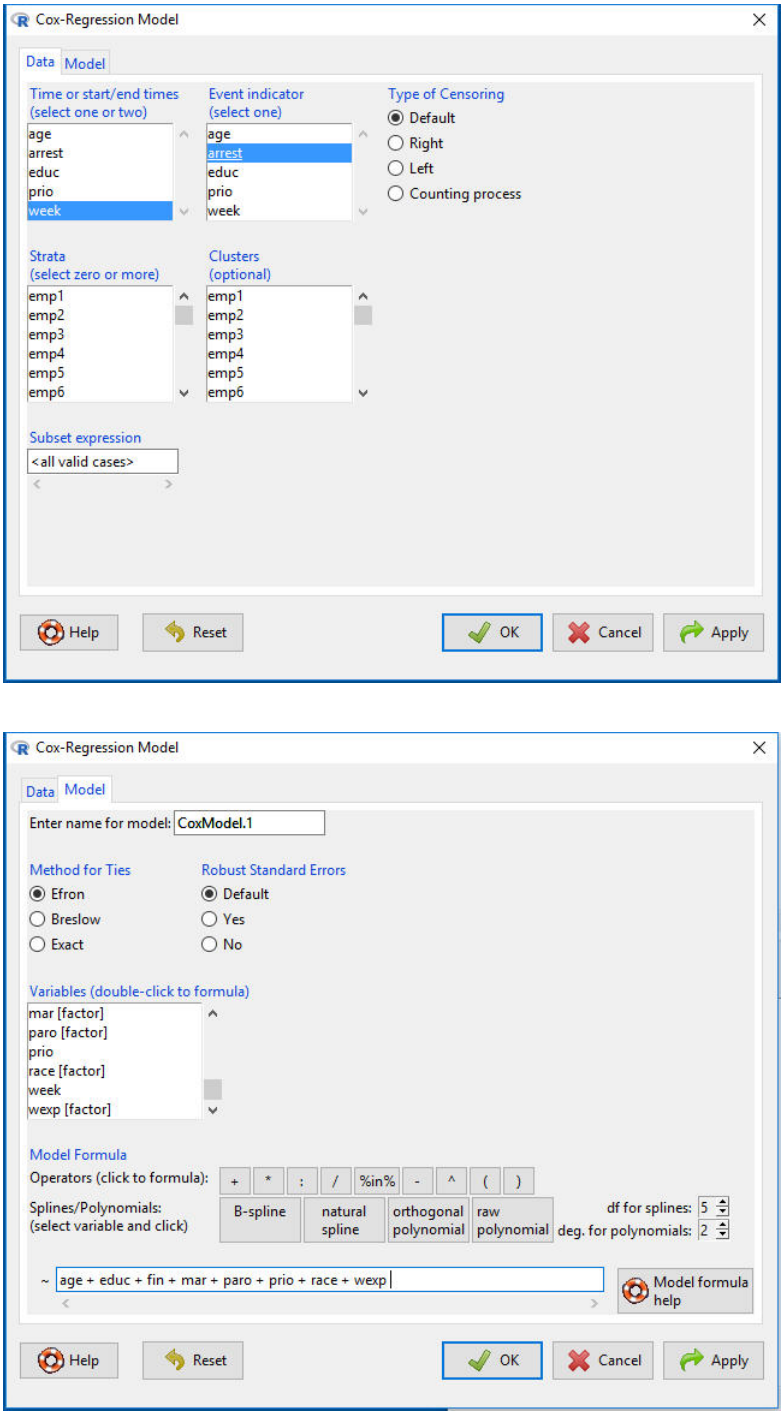

FIGURE 5.3: The R Commander *Cox-Regression Model* dialog box, *Data* tab (top) and *Model* tab (bottom); this example uses the `Rossi` data set.

```
CoxModel <- function(){
    . . .
    initializeDialog(title=gettext("Cox-Regression Model",
            domain="R-RcmdrPlugin.survival"),
        use.tabs=TRUE, tabs=c("dataTab", "modelTab"))
    . . .
    OKCancelHelp(helpSubject="coxph", model=TRUE, reset="CoxModel", apply="CoxModel")
    . . .
    modelFormula(modelTab, hasLhs=FALSE, rhsExtras=TRUE)
    . . .
    tkgrid(labelRcmdr(outerOperatorsFrame, text="         "),
        operatorsFrame, sticky="w")
    tkgrid(outerOperatorsFrame, sticky="ew")
    tkgrid(formulaFrame, sticky="w")
    tkgrid(labelRcmdr(dataTab, text=""))
    . . .
    dialogSuffix(focus=rhsEntry, preventDoubleClick=TRUE, use.tabs=TRUE,
        grid.buttons=TRUE,
        tabs=c("dataTab", "modelTab"),
        tab.names=gettext("Data", "Model", domain="R-RcmdrPlugin.survival"))
}
```

FIGURE 5.4: The `CoxModel()` callback function, with most lines omitted; elided lines are indicated by `. . .` . For the complete `CoxModel()` function, consult the sources for the **RcmdrPlugin.survival** package.

5.3 The R Commander *Models* Menu

An advantage of having your plug-in package define new classes of statistical models is that, once fit, these models can be manipulated via the R Commander *Models* menu. The fully expanded *Models* menu is displayed in Figure 5.5. The corresponding lines in the `Rcmdr-menus.txt` file that define the *Models* menu are shown in Figure 5.6. I've suppressed the final `install?` field in these lines so that they fit on the page; the `install?` field isn't relevant, in any event, to the current discussion.

Writing this section caused me to rethink how the R Commander integrates new classes of statistical models introduced by plug-in packages. The lines in Figure 5.6 from `Rcmdr-menus.txt` are therefore taken from the *development version* 2.4-0 of the **Rcmdr** package (rather than from the current CRAN version, which is 2.3-0 at the time of writing), where I improved the `activation` field for some of the items in the *Models* menu to make them play better with plug-in packages.[1] I think that there's further room for improvement, and I may introduce additional changes—and not just in the `Rcmdr-menus.txt` file. If I do so, I'll endeavor to make the changes backwards-compatible.

Let's focus our attention first on the `activation` field for the *Models* menu lines in the `Rcmdr-menus.txt` file:

- Many of the menu lines have `"lmP()"` or `"lmP() || glmP()"` in the `activation` field,

[1] For an explanation of how the R Commander menus are defined in the `Rcmdr-menus.txt` file, see Chapter 3.

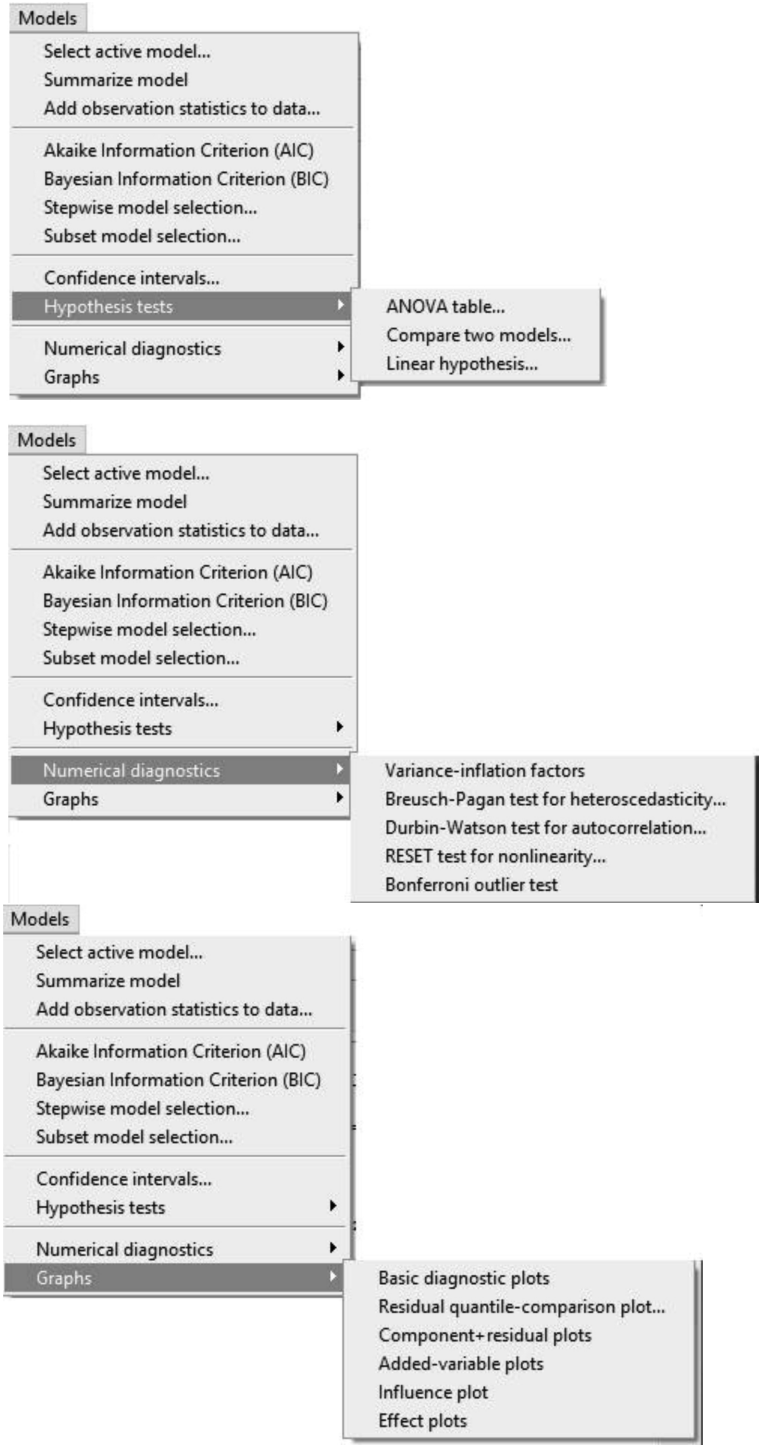

FIGURE 5.5: The R Commander *Models* menu, expanded.

```
#type menu/item    operation/parent  label                                          command/menu              activation
menu modelsMenu       topMenu     ""                                             ""                        ""
item modelsMenu       command     "Select active model..."                       selectActiveModel         "modelsP()"
item modelsMenu       command     "Summarize model"                              summarizeModel            "activeModelP()"
item modelsMenu       command     "Compare model coefficients..."                compareCoefficients       "modelsP(2)"
item modelsMenu       command     "Add observation statistics to data..."        addObservationStatistics  "activeModelP()"
item modelsMenu       separator   ""                                             ""                         ""
item modelsMenu       command     "Akaike Information Criterion (AIC)"           aic                        "logLikP()"
item modelsMenu       command     "Bayesian Information Criterion (BIC)"         bic                        "logLikP()"
item modelsMenu       command     "Stepwise model selection..."                  stepwiseRegression         "logLikP()"
item modelsMenu       command     "Subset model selection..."                    subsetRegression           "lmP()"
item modelsMenu       separator   ""                                             ""                         ""
item modelsMenu       command     "Confidence intervals..."                      confidenceIntervals        "activeModelP()"
item modelsMenu       command     "Bootstrap confidence intervals..."            Bootstrap                  "activeModelP()"
item modelsMenu       command     "Delta method confidence interval..."          DeltaMethodConfInt         "activeModelP() && !multinomP()"
menu hypothesisMenu   modelsMenu  ""                                             ""                         ""
item modelsMenu       cascade     "Hypothesis tests"                             hypothesisMenu             ""
item hypothesisMenu   command     "ANOVA table..."                               anovaTable                 "activeModelP()"
item hypothesisMenu   command     "Compare two models..."                        compareModels              "modelsP(2)"
item hypothesisMenu   command     "Linear hypothesis..."                         testLinearHypothesis       "activeModelP()"
item modelsMenu       separator   ""                                             ""                         ""
menu diagnosticsMenu  modelsMenu  ""                                             ""                         ""
item modelsMenu       cascade     "Numerical diagnostics"                        diagnosticsMenu            ""
item diagnosticsMenu  command     "Variance-inflation factors"                   VIF                        "lmP() || glmP()"
item diagnosticsMenu  command     "Breusch-Pagan test for heteroscedasticity..." BreuschPaganTest           "lmP()"
item diagnosticsMenu  command     "Durbin-Watson test for autocorrelation..."    DurbinWatsonTest           "lmP()"
item diagnosticsMenu  command     "RESET test for nonlinearity..."               RESETtest                  "lmP()"
item diagnosticsMenu  command     "Bonferroni outlier test"                      OutlierTest                "lmP() || glmP()"
menu modelsGraphsMenu modelsMenu  ""                                             ""                         ""
item modelsMenu       cascade     "Graphs"                                       modelsGraphsMenu           ""
item modelsGraphsMenu command     "Basic diagnostic plots"                       plotModel                  "lmP() || glmP()"
item modelsGraphsMenu command     "Residual quantile-comparison plot..."         residualQQPlot             "lmP()"
item modelsGraphsMenu command     "Component+residual plots"                     CRPlots                    "lmP() || glmP()"
item modelsGraphsMenu command     "Added-variable plots"                         AVPlots                    "lmP() || glmP()"
item modelsGraphsMenu command     "Influence plot"                               InfluencePlot              "lmP() || glmP()"
item modelsGraphsMenu command     "Effect plots"                                 effectPlots                "EffectP()"
item topMenu          cascade     "Models"                                       modelsMenu                 ""
```

FIGURE 5.6: The lines (slightly edited, and with the `install?` field suppressed) in the `Rcmdr-menus.txt` file that define the R Commander *Models* menu, its sub-menus, and menu items. These lines are taken from the development version 2.4-0 of the **Rcmdr** package.

restricting these menu items to linear models or to linear and generalized linear models. The functions `lmP()` and `glmP()` are *predicate functions*,[2] which return `TRUE` or `FALSE` depending upon whether a condition holds. R Commander-supplied predicates typically—but not necessarily—are called with no arguments. Here, `lmP()` returns `TRUE` if there's an active statistical model and if it's of (primary) class `"lm"` or `"aov"`; similarly, `glmP()` returns `TRUE` if there's an active statistical model of (primary) class `"glm"`.

- Some of the predicate functions are capable of testing whether there are *at least* a certain number of objects of a particular type. For example, `modelsP(2)`, used twice in Figure 5.6, returns `TRUE` if there are 2 or more statistical models (of any model class recognized by the R Commander) in memory and `FALSE` otherwise.

- Some of the menu lines simply require that there's an active model: `activeModelP()` returns `TRUE` in this circumstance. The implicit assumption is that the callback function for the corresponding menu item can handle a model object of *any* class. This assumption may be problematic: For example, the `summarizeModel()` callback function *assumes* that there's an appropriate `summary()` method for the class of the current statistical model. In this case, if no `summary()` method exists, the `summary.default()` method will be invoked, likely producing uninformative output but not an error. There probably *should be* a `summary()` method for the class of statistical models in question, however, and your plug-in package could provide it—`summary.foo()` for models of class `"foo"`.

  In other cases, an error might result if a method is absent and there's no default method, or if the default method makes an assumption about the contents of a model object that doesn't hold. A safe solution is to provide the necessary method, even if it just prints an informative error or warning message, which would then appear in the R Commander *Messages* pane: For example,

[2] R Commander predicates are discussed more systematically in Section A.4 of Appendix A.

```
vcov.foo <- function(object, ...){
  stop('There is no vcov() method for models of class "foo".'
}
```

I'll likely add more effective error-checking to the *Models*-menu callback functions, and may try to generalize some menu items that are currently restricted to `"lm"` and `"glm"` objects.

Table 5.1 shows the assumptions made by callback functions invoked by menu items in the *Models* menu.

- The `addObservationStatistics()` callback function requires additional explanation: The dialog created by this function provides potentially for adding fitted values, residuals, studentized residuals, hat-values, Cook's distances, and observation indices to the data set. Whether or not these statistics (with the exception of observation indices, which can always be added to the data set) are available for the current model depends upon whether there are applicable methods for the `fitted()`, `residuals()`, `rstudent()`, `hatvalues()`, and `cooks.distance()` generics. For fitted values and residuals, the default methods will be invoked, possibly producing errors. In these cases, it would be better, as explained above, if you could either introduce methods that return proper fitted values and residuals, or, if that's not sensible, print informative error messages.

In addition to referencing existing menu items in the *Models* menu, your plug-in can define new menu items and sub-menus specific to a model class introduced by the plug-in. An example is provided by the **RcmdrPlugin.survival** package: See Figures 3.5 and 3.6 (on pages 20–21).

Callback Function	Activation	Assumptions
`summarizeModel()`	`activeModelP()`	`summary()` method
`compareCoefs()`	`modelsP(2)`	`coef()` and `vcov()` methods for models selected
`aic()`, `cbic()`, `stepwiseRegression()`	`logLikP()`	`logLik()` method returning `"df"` and `"nobs"` as attributes
`addObservationStatistics()`	`activeModelP()`	none (see discussion)
`confidenceIntervals()`	`activeModelP()`	`confint()` method
`Bootstrap`	`activeModelP()`	`coef()` method
`DeltaMethodConfint`	`activeModelP() && !multinomP()`	`coef()` and `vcov()` methods
`anovaTable`	`activeModelP()`	`Anova()` method, possibly `Anova.default()`
`compareModels()`	`modelsP(2)`	`anova()` method supporting model comparison
`testLinearHypothesis()`	`activeModelP()`	`coef()` and `vcov()` methods
`effectPlots`	`EffectP()`	non-default `Effect()` method

TABLE 5.1: Assumptions made by callback functions in the R Commander *Models* menu (see Figure 5.6)

5.4 The RcmdrModels: Field in the Plug-in Package DESCRIPTION File

The last piece of the models puzzle is the `RcmdrModels:` field in the plug-in package's `DESCRIPTION` file, which simply lists the classes of new statistical models to be recognized by the R Commander. By way of illustration, the `DESCRIPTION` file for the **RcmdrPlugin.survival** package is shown in Figure 5.7.[3] Thus, when the plug-in package is loaded, the R Commander subsequently recognizes objects of classes `"coxph"` (Cox regression models), `"survreg"` (parametric survival regression models), and `"cox.penal"` (Cox models with "frailty" terms) as statistical model objects. The remainder of the package `DESCRIPTION` file is standard.

[3]This `DESCRIPTION` file was also discussed in Section 2.4.

```
Package: RcmdrPlugin.survival
Type: Package
Title: R Commander Plug-in for the 'survival' Package
Version: 1.1-1
Date: 2016-08-15
Author: John Fox
Maintainer: John Fox <jfox@mcmaster.ca>
Depends: survival, date, stats
Imports: Rcmdr (>= 2.2-1)
Description: An R Commander plug-in for the survival
  package, with dialogs for Cox models, parametric survival regression models,
  estimation of survival curves, and testing for differences in survival
  curves, along with data-management facilities and a variety of tests,
  diagnostics and graphs.
License: GPL (>= 2)
LazyLoad: yes
LazyData: yes
RcmdrModels: coxph, survreg, coxph.penal
```

FIGURE 5.7: `DESCRIPTION` file for the **RcmdrPlugin.survival** package (shown previously in Figure 2.5), illustrating the `RcmdrModels:` field.

6

Debugging R Commander Plug-in Packages

Debugging programs can be a difficult process in the best of circumstances. For several reasons, debugging R Commander dialog-box callback functions isn't the best of circumstances:

- It is often necessary to *use* (i.e., to interact with) the dialog to discover the source of an error.
- The macro-like utility functions employed to construct R Commander dialogs complicate the debugging process.
- The R Commander intercepts error and warning messages to print them in the *Messages* pane, at times rendering the messages invisible when there is a dialog-disabling bug.
- You normally need to restart R, reinstall your package, and reload the **Rcmdr** package and your plug-in, and then input a data set, to test a dialog (but see below).

I use the RStudio IDE to develop and debug R programs and packages (including the **Rcmdr** package), and, partly for this reason, I've taken pains to insure the the R Commander can be used inside RStudio. In it's default configuration, when the **Rcmdr** package is loaded in RStudio, it directs output and messages to the R console, and the main R Commander window therefore appears without *Output* and *Messages* panes. In addition, R help pages invoked from the R Commander appear in the RStudio *Help* tab. Because of potential incompatibilities with the RStudio graphics device, however, graphs are displayed in the R *Windows* graphics device under Windows, the *Quartz* graphics device under Mac OS X, and the *X11* graphics device under Linux/Unix.

6.1 Debugging Callback Functions

You can debug a dialog-box callback function in your *installed* plug-in using the usual R and RStudio debugging tools. For example, to debug the `Survfit()` callback function having loaded the **RcmdrPlugin.survival** package, you can issue the commands

```
debugonce(RcmdrPlugin.survival::Survfit)
Survfit()
```

at the command prompt in the R console. These commands work because the **RcmdrPlugin.survival** package exports its callback functions (as explained in Section 2.4).[1] I suggest that you try the commands to see what happens:

[1] Because the **Rcmdr** package *doesn't* export its callback functions, you would need, e.g., `debugonce(Rcmdr:::numericalSummaries)` and the command `Rcmdr:::numericalSummaries()` to debug a standard R Commander callback function. The only reason to do so, however, would likely be to see how the function works.

- Notice how macro calls to functions like `OKCancelHelp()` and `radioButtons()` are expanded, stepping through each line of the macro. You can use the *RStudio* debugging tools here to skip to the end of a macro, for example by selecting *Debug > Finish function or loop* from the RStudio menus, by pressing the equivalent key combination (*Shift-F6*), or by pressing the corresponding tool button near the top of the *Console*.

- Debugging a callback function can be useful to detect problems in constructing the corresponding dialog box but won't help you locate an error in building an R command in your dialog. Once the callback function exits, debug mode terminates, typically leaving an open modal Tk dialog box. If there's a problem in the local `onOK()` function for the dialog, then you won't discover that until you press the *OK* button in the dialog.

- If you want to debug the `onOK()` function for the dialog, you can enter the command `debug(onOK)` while you're stepping through your callback function in debugging mode, after this local function is defined in the callback function. Then, when you subsequently press *OK* in the displayed dialog, you'll enter `onOK()` in debugging mode, allowing you to use the usual debugging tools to step through the function, to examine local variables like the command that's constructed in `onOK()`, etc.

Alternatively, and often more conveniently, you could insert a call or calls to `browser()` in a callback function or inside the local `onOK()` to stop execution at the corresponding lines, entering debugging mode at these points.

To illustrate, Figure 6.1 shows a "screenshot" of RStudio in the process of debugging the local `onOK()` function defined by the `Survfit()` callback function. To arrive at this point, having loaded the **RcmdrPlugin.survival** plugin and read the `Rossi` data set from the package:

- I entered the command `debugonce(RcmdrPlugin.survival::Survfit)`;

- called the `Survfit()` function at the R command prompt;

- stepped through `SurvFit()` until `onOK()` was defined;

- entered the command `debug(onOK)` in debugging mode at the browser command prompt;

- continued through `Survfit()` until the callback function exited, bringing up the *Survival Function* dialog box (shown in Figure 6.2);

- clicked the *OK* button in the dialog to execute `onOK()`, entering debugging mode in `onOK()`.

To debug callback functions for your plug-in in this manner implies reinstalling and restarting the plug-in each time your make a change to a callback function. That's time-consuming, and so I've built some features into the R Commander to facilitate more convenient debugging. To use these features, you'll have to unpack the source-code tree for the **Rcmdr** package in a convenient location on your computer. Then proceed as follows:

- Comment-out the following two lines in the `Commander()` function, located in the file `commander.R`:

```
assignInMyNamespace("RcmdrEnv", RcmdrEnv)
assignInMyNamespace(".RcmdrEnv", NULL)
```

  In version 2.3-0 of the **Rcmdr** package, these are lines 28 and 29 in `commander.R`, and are preceded by the comment

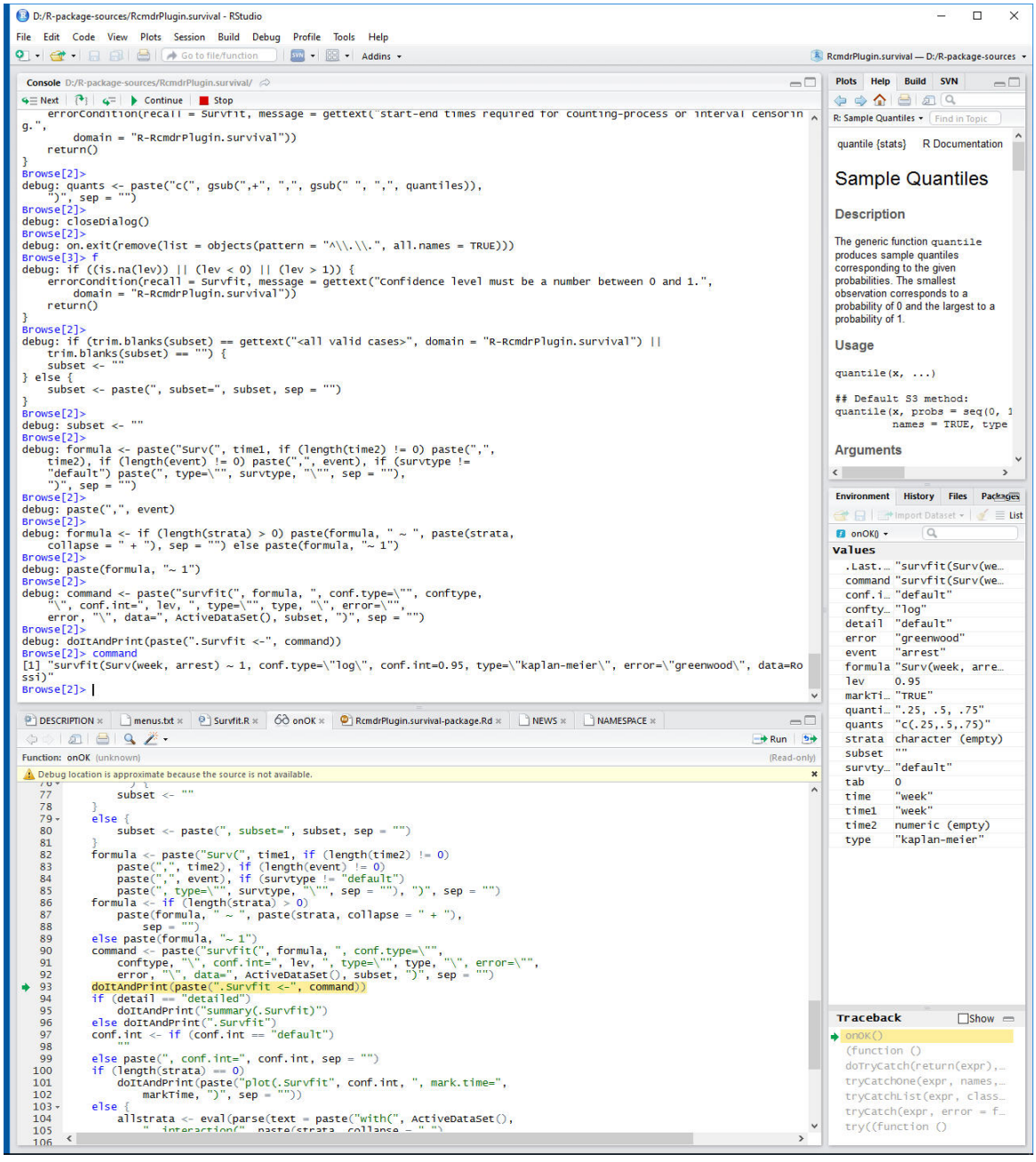

FIGURE 6.1: RStudio session debugging the local `onOK()` function defined by the `Survfit()` callback function.

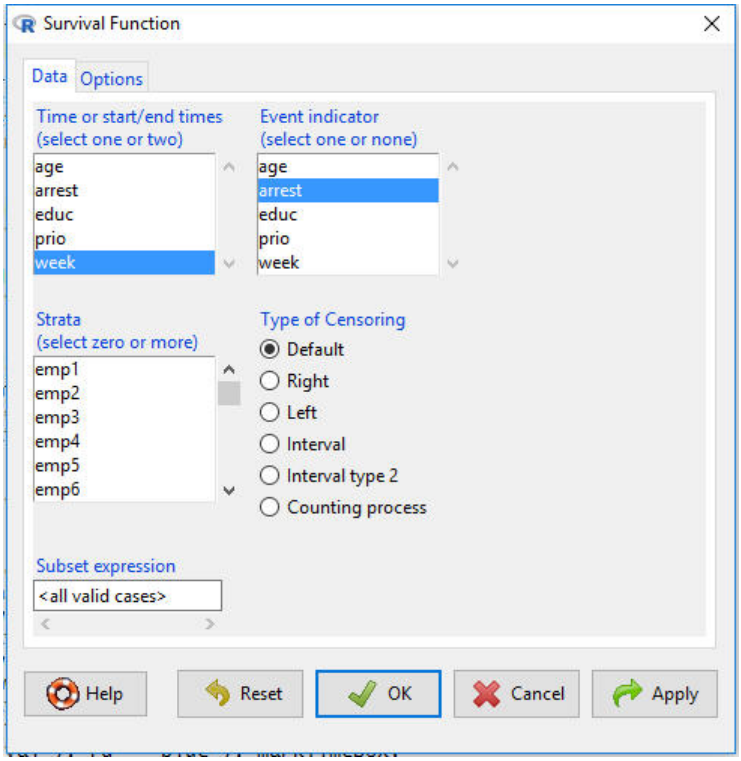

FIGURE 6.2: *Survival Function* dialog constructed by the `Survfit()` callback function in the **RcmdrPlugin.survival** package. Clicking the *OK* button in the dialog produces the debugging session shown in Figure 6.1.

```
# 1. load the Rcmdr package via library(Rcmdr)
#        or possibly library(RcmdrPlugin.<your plugin>)
# 2. close the Commander window (but don't exit from R)
# 3. source this file
# 4. re-open the Commander window via Commander()
# 5. source the file for the dialog you're working on
# 6. open the dialog by entering your_dialog_function() in the R Console

path <-  "<location of the Rcmdr sources on your computer>/R" # adjust for your system
files <- list.files(path, pattern=".*\\.R")
files <- paste(path, files, sep="/")
for (file in files) source(file)
library(tcltk)
library(tcltk2)
options(Rcmdr=list(RcmdrEnv.on.path=TRUE, suppress.X11.warnings=TRUE,
        use.markdown=TRUE))
```

FIGURE 6.3: The file `debug-Rcmdr.R`.

```
# the following two lines to be commented-out for debugging:
```

Do not reinstall the **Rcmdr** package after making these changes to `commander.R`.

- Load the **Rcmdr** package and your plug-in package in the normal manner, and then immediately close the R Commander window without exiting from R.
- Source the file `debug-Rcmdr.R`, which is shown in Figure 6.3. To use this file you first must edit it to reflect the location of the **Rcmdr** package sources on your system. You can download a copy of the file at http://socserv.mcmaster.ca/jfox/Books/RCommander/debug-Rcmdr.R.
- Then source the `.R` file for the callback function that you're working on—say the function `Survfit()`, defined in the file `Survfit.R` in the **RcmdrPlugin.survival** sources.
- Restart the R Commander interface by entering `Commander()` at the R command prompt in the *Console*.
- Finally, invoke the callback function directly at the R command prompt, for example, by the command `Survfit()`. You can debug the callback function in the usual manner, by setting breakpoints, calling `debugonce(Survfit)`, etc. Normally, you'd read a data set and perform whatever operations are required *before* invoking the callback function. If the data set is already in memory from a previous debugging iteration, then you can simply make it the active data set via the *Data set:* button in the R Commander toolbar.

Unless you end up crashing R or the R Commander, you should be able edit your callback function, source it, and debug it multiple times without restarting R or reloading the **Rcmdr** package or your plug-in.

Appendix A

A Guide to the R Commander Utility Functions

This appendix covers the utility functions exported by the **Rcmdr** package that are most useful in writing plug-in packages. I've omitted functions that, although exported by the **Rcmdr** package, are unlikely to be used in a plug-in, such as functions for managing the R Markdown document created during an R Commander session. *All* of the exported utilities are briefly described in the help file `?Rcmdr.Utilities`.

The functions described in the appendix are divided into sections based on their purpose. The arguments for each function are presented and explained. Macro-like functions are marked as such. Macro-like functions are occasionally required to deal with scoping issues that arise in interfacing R with Tcl/Tk via the **tcltk** package.

A.1 Building Dialogs

`initializeDialog()` (macro)

Creates a top-level Tk window for an R Commander dialog box and performs a variety of housekeeping operations. You should typically call `initializeDialog()` near the beginning of a callback function that creates a dialog box.

```
initializeDialog(window=top, title="", offset=10, preventCrisp,
    use.tabs=FALSE, notebook=notebook, tabs=c("dataTab", "optionsTab"),
    suppress.window.resize.buttons=TRUE)
```

Arguments:

`window` The name of the top-level window to be created. I typically use the default `top` for a main dialog and `subdialog` for a sub-dialog created inside the `onOK()` function of a main dialog.

`title` The text to appear in the title-bar of the window.

`offset` Offset of the top-left corner of the window in pixels from the top-left corner of the main R Commander window.

`preventCrisp` Ignored (but retained for backwards compatibility).

`use.tabs` Create a tabbed dialog?

`notebook` Name for the Tk notebook widget created to contain tabs (if `use.tabs=TRUE`).

`tabs` Names for the tabs—use as many names as there are tabs to be created.

`suppress.window.resize.buttons` Should be `TRUE` for a dialog that isn't resizable.

`dialogSuffix()` (macro)

Performs house-keeping tasks to finalize a dialog box. You would typically call `dialogSuffix()` at the end of your callback function.

```
dialogSuffix(window=top, onOK=onOK, onCancel=onCancel, rows, columns,
    focus=top, bindReturn=TRUE, preventGrabFocus=FALSE,
    preventDoubleClick=FALSE, preventCrisp, use.tabs=FALSE,
    notebook=notebook, tabs=c("dataTab", "optionsTab"),
    tab.names=c("Data", "Options"), grid.buttons=FALSE,
    resizable=FALSE, force.wait=FALSE)
```

Arguments:

`window` The name of the top-level window to be finalized.

`onOK` The name of the local function defined within the callback function to be called when the *OK* button in the dialog box is pressed. I typically use the default `onOK` for a main dialog and `onOKsub` for a sub-dialog.

`onCancel` The name of the local function to be called when the *Cancel* button is pressed. The `OKCancelHelp()` macro provides a standard `onCancel()` function, so this argument can almost always be left at its default.

`rows`, `columns`, `preventCrisp` Ignored (but retained for backwards compatibility).

`focus` Tk window to get the focus. By default `top`, which is the usual name for the top-level window for the dialog, but may be, e.g., a text widget within the dialog.

`bindReturn` Bind the *Return* or *Enter* key to the `onOK` function.

`preventGradFocus` Prevent the dialog box from grabbing the focus.

`preventDoubleClick` Prevent double-clicking from pressing the *OK* button, even when the `double.click` option is set; necessary for statistical modelling dialogs, which use double-clicking to build the model formula.

`use.tabs` Finalize a tabbed dialog?

`notebook` Name of the Tk notebook widget containing tabs in a tabbed dialog.

`tab.names` Text labels for the tabs in a tabbed dialog.

`grid.buttons` Insert a call to `tkgrid()` for the frame containing the *OK*, *Cancel*, etc., buttons; use `TRUE` for tabbed dialogs and optionally for other dialogs.

resizable Is the dialog resizable?

`force.wait` Call `tkwait.window()` so that processing is suspended until the dialog is closed; overrides the R Commander `tkwait.dialog` option if the latter is set to `FALSE` (its default). The `force.wait` argument should normally be set to `TRUE` for sub-dialogs when the main dialog is still open.

`OKCancelHelp()`, `subOKCancelHelp()` (macros)

Creates *OK* and *Cancel* buttons, and, optionally, *Help*, *Reset*, and *Apply* buttons; `subOKCancelHelp()` is for sub-dialogs and creates only *OK*, *Cancel*, and (optionally) *Help* buttons.

```
OKCancelHelp(window=top, helpSubject=NULL, model=FALSE,
    reset=NULL, apply=NULL, helpPackage=NULL)

subOKCancelHelp(window=subdialog, helpSubject=NULL)
```

Arguments:

`window` The name of the top-level window to which the buttons will be added.

`helpSubject` Quoted character string to be used in a call to `help()` when the *Help* button is pressed; if `NULL` there will be no *Help* button.

`model` Set to `TRUE` for a statistical modeling dialog.

`reset` Name of the function to be called, typically the callback function itself, when the *Reset* button is pressed; if `NULL` there will be no *Reset* button.

`apply` Name of the function to be called, typically the callback function itself, when the *Apply* button is pressed; if `NULL` there will be no *Apply* button.

`helpPackage` Quoted name of the package in which the specified help subject resides; not usually necessary but may be needed to resolve ambiguity when the same help subject appears in more than one attached package.

Unless you specify `grid.buttons=TRUE` in the call to `dialogSuffix()`, it's necessary to include a command like `tkgrid(buttonsFrame, sticky="w")` above the call to `dialogSuffix()` to place the *OK*, *Cancel*, etc., buttons at the bottom of the dialog box.

`checkBoxes()` (macro)

Builds a set of check boxes.

```
checkBoxes(window=top, frame=stop("frame not supplied"),
    boxes=stop("boxes not supplied"),
    initialValues=NULL, labels=stop("labels not supplied"),
    title=NULL, ttk=FALSE)
```

Arguments:

`window` Name of a previously created parent window to contain the check boxes.

`frame` Quoted name of a Tk frame widget to be created to contain the check boxes.

`boxes` Character vector of names for the check boxes, one for each box to be created. A Tcl variable is created in the environment of the calling function for each check box, recording the current state of the check box. If, e.g., `boxes=c("a", "b")`, these variables are named `aVariable` and `bVariable`. You should make sure that these names are unique within the dialog-box function.

`initialValues` A vector of `0`s and `1`s indicating whether each box is (respectively) initially unchecked or checked; the values can be quoted numerals (character values) or numeric.

`labels` A vector of character strings to label the check boxes.

`title` An optional title (character string) to label the set of check boxes

`ttk` If `TRUE` (the default is `FALSE`), a border (box) is drawn around the set of check boxes.

`radioButtons()` (macro)

Constructs a related set of radio buttons.

```
radioButtons(window=top, name=stop("name not supplied"),
    buttons=stop("buttons not supplied"),
    values=NULL, initialValue=..values[1],
    labels=stop("labels not supplied"),
    title="", title.color=getRcmdr("title.color"),
    right.buttons=FALSE, command=function(){})
```

Arguments:

`window` Name of a previously created parent window to contain the radio buttons.

`name` Quoted name to be used to construct the Tcl variable recording the state of the radio buttons and for the Tk frame widget containing the buttons. For example, if `name="buttons"` then the variable is `buttonsVariable` and the frame `buttonsFrame`. These names should be unique within the dialog-box function.

`buttons` Character vector of names for the buttons, to be used for creating Tk buttons; for example, if `buttons=c("one", "two")` then the buttons are named `oneButton` and `twoButton`. These names also should be unique within the dialog-box function.

`values` A vector of values, one per button, to be returned when the corresponding button is pressed; if `NULL` (the default) this is set to the `buttons` argument.

`initialValue` The value of the initially pressed button; by default, this is the first button.

`labels` A vector of text labels to be printed next to the buttons.

`title` An optional title for the set of radio buttons.

`title.color` The color for the title, defaulting to the standard R Commander title color.

`right.buttons` If `TRUE` (the default is `FALSE`) the button labels are printed to the right of the buttons.

`command` A function to be executed each time a button is pressed (and before the user presses *OK*), allowing you to modify the dialog depending upon the currently pressed button. Rarely used.

`variableListBox()`, `variableComboBox()` (macros)

These functions construct list widgets, returned as objects respectively of class `"listbox"` and `"combobox"`, from which the user can select one or (optionally) more items. The lists are usually, but not necessarily in the case of `variableListBox()`[1], the variables in the current data set or a subset of these variables (such as the factors in the current data set. `variableListBox()` creates a scrollable list, `variableComboBox()` a drop-down list.

```
variableListBox(parentWindow, variableList=Variables(), bg="white",
    selectmode="single", export="FALSE", initialSelection=NULL,
    listHeight=getRcmdr("variable.list.height"), title)

variableComboBox(parentWindow, variableList=Variables(),
    export="FALSE", state="readonly",
    initialSelection=gettextRcmdr("<no variable selected>"),
    title="")
```

Arguments:

`parentWindow` The Tk widget containing the listbox—the top-level window for the dialog, a tab, or a Tk frame.

`variableList` A character vector of items composing the list to be displayed, most often a list of variables from the current data set. The default, returned by `Variables()` is all of the variables in the current data set; other common values are `Factors()` (all factors in the current data set) and `Numeric` (all numeric variables). See Section A.3 for more on `Variables()`, `Factors()`, and `Numeric()`.

`bg` The color of the background of the variable list.

`selectmode` If `"single"` (the default), the user can select only one item in the list; specify `selectmode="multiple"` to allow the user to select more than one item. Note that you cannot otherwise restrict the *number* of items selected, so, e.g., if you require the user to select exactly two items, you'll have to check the number selected.

`export` Sets the `exportSelection` for the Tk `listbox` widget. You can almost always leave this at the default value, `FALSE`.

`initialSelection` For `variableListBox()`, the index or indexes of the item or items in the list that are initially selected, using 0-based indexing. You can use `varPosn()` (see immediately below) to translate variable names into indexes. If `NULL` (the default), then no items are initially selected.

`listHeight` The number of items to display at one time. The default is taken from the R Commander `variable.list.height` option.

`title` Character string giving the title for the listbox; optional for `variableComboBox()`.

`state` `"readonly"` if the user can't modify the entries in a combo box. Other possibilities are `"normal"` and `"disabled"`, but these are rarely useful.

[1] Because `variableComboBox()` attaches `"<no variable selected>"` to the top of the displayed list, it's only really suitable for lists of variables.

varPosn()

Returns the 0-based position of one or more names, typically variable names, in a vector of names.

```
varPosn(variables,
    type=c("all", "factor", "numeric", "nonfactor", "twoLevelFactor"),
    vars=NULL)
```

Arguments:

`variables` A character vector of names.

`type` The type of variables in the active data set within which the names in `variables` will be matched.

`vars` An optional arbitrary vector of names within which the names in `variables` will be matched; if given, `type` is ignored.

getFrame(), getSelection()

These generic functions are for working with `"listbox"` and `"combobox"` objects created by `variableListBox()` and `variableComboBox()`. `getFrame()` returns the Tk frame widget containing the listbox or combo box, and can be used, e.g., with `grid()` to place the box in the dialog. `getSelection()` returns a vector of names of the currently selected items in a listbox (possibly one or none—in the latter event the result is length 0), or the name of the selected item in a combo box.

```
getFrame(object)

getSelection(object)
```

Argument:

`object` An object of class `"listbox"` or `"combobox"`.

`groupsBox()` (macro)

Creates a "by-group" button and associated sub-dialog, for use, e.g., in plotting by groups.

```
groupsBox(recall=NULL, label=gettextRcmdr("Plot by:"),
    initialLabel=gettextRcmdr("Plot by groups"),
    errorText=gettextRcmdr("There are no factors in the active data set."),
    variables=Factors(),
    plotLinesByGroup=FALSE, positionLegend=FALSE,
    plotLinesByGroupsText=gettextRcmdr("Plot lines by group"),
    initialGroup=NULL, initialLinesByGroup=1, window=top)
```

Arguments:

`recall` The function to be called in the event that there are no applicable groups variables in the active data set; normally the dialog-box function that calls `groupsBox()`.

`label` The text for the "by-group" button when a groups variable is selected (to be followed by the name of the selected variable).

`initialLabel` The text for the "by-group" button when no groups variable is selected.

`errorText` The error message to be printed if the button is pressed when no suitable groups variables are in the active data set.

`variables` Candidates for groups variables, defaults to all factors in the active data set.

`plotLinesByGroup` Include a check-box in the groups sub-dialog? `FALSE` by default.

`positionLegend` Includes the message "Position legend with mouse click" in the sub-dialog; no longer used in any standard R Commander dialog; retained for backwards compatibility.

`plotLinesByGroupsText` The text to appear to the left of the (optional) check-box in the sub-dialog.

`initialGroup` Quoted name of the variable initially selected in the sub-dialog. If `NULL` no variable is initially selected.

`initialLinesByGroup` If `1` (the default), the (optional) check-box in the sub-dialog is initially checked; if `0` it is unchecked.

`window` The unquoted name of the Tk windows containing the "by-group" button.

The `groupsBox()` macro creates several variables in the environment of the calling dialog-box function: `groupsFrame` is the Tk frame widget containing the "by-group" button and can be placed in the dialog with, e.g., `tkgrid(groupsFrame, sticky="w")`; `.groups` contains the quoted name of the selected groups variable or is `FALSE` if no groups variable is selected; `.linesByGroup` is `TRUE` or `FALSE` depending upon the status of the (optional) check-box.

`subsetBox()` (macro)

Creates a text box for a subsetting expression.

```
subsetBox(window = top, subset.expression = NULL, model = FALSE)
```

Arguments:

`window` The Tk window to contain the subset box.

`subset.expression` If non-`NULL`, a character string that evaluates to an R subsetting expression—e.g., returning `TRUE` or `FALSE` for each case. If `NULL` and `model=TRUE`, the subset expression saved by a previous statistical model if one was saved. Otherwise if `NULL`, the string `"<all valid cases>"` is used.

`model` `TRUE` for a statistical-modeling dialog (such as the standard R Commander *Linear Model* dialog) that handles the subset expression.

The `subsetBox()` macro creates the variables `subsetFrame` (with the Tk frame widget containing the subset box) and `subsetVariable` (the Tcl variable containing the contents of the box) in the environment of the calling dialog-box function.

`modelFormula()` (macro)

Constructs an R model-formula widget, usually, but not necessarily, for a statistical-modeling dialog.

```
modelFormula(frame=top, hasLhs=TRUE, rhsExtras=NULL,
    formulaLabel=gettextRcmdr("Model Formula"))
```

Arguments:

`frame` The Tk window within which the formula widget resides.

`hasLhs` If `FALSE` (the default is `TRUE`), the formula has only a right-hand side.

`rhsExtras` If `TRUE`, the toolbar containing buttons for generating polynomials and regression splines is included in the formula widget. The default is `TRUE` for models that have a left-hand side and `FALSE` for those that don't.

`formulaLabel` The text label to appear to the left of the formula box.

The `modelFormula()` macro creates several variables in the environment of the calling dialog-box function, including `formulaFrame` (the Tk frame containing the formula widget); and `lhsVariable` and `rhsVariable` (Tcl variables with the left-hand and right-hand sides of the formula).

`formulaFields()`

Extracts information from a statistical-model object.

```
formulaFields(model, hasLhs=TRUE, glm=FALSE)
```

Arguments:

`model` The model object

`hasLhs` Whether the formula in the model object has a left-hand side.

`glm` Whether the model is a `"glm"` object.

Returns a list with the following elements:

`lhs` A character string with the left-hand side of the model, or `NULL` if `hasLhs=FALSE`.

`rhs` A character string with the right-hand side of the model.

`data` A character string with the name of the data set to which the model was fit.

`family` For a GLM, a character string with the name of the family for the model; otherwise `NULL`.

`link` For a GLM, a character string with the name of the link for the model; otherwise `NULL`.

`UpdateModelNumber()`

The R Commander numbers models serially during a session; the numbers can be used to create unique model names, and the current number can be retrieved via `getRcmdr("modelNumber")`.

```
UpdateModelNumber(increment=1)
```

Argument:

`increment` Self-explanatory, defaults to `1`. If negative, the model number is decremented.

You normally call `UpdateModelNumber()` near the beginning of a statistical modeling dialog, before constructing a (default) model name. In the event of an error, you can call `UpdateModelNumber(-1)` to avoid skipping numbers.

putDialog(), getDialog()

These functions are for managing state information for dialogs.

```
putDialog(dialog, values=NULL, resettable=TRUE)
```

```
getDialog(dialog, defaults=NULL)
```

Arguments:

`dialog` Character string giving the name under which state information is stored; typically the name of the dialog-box function.

`values` A list containing the state information, with elements of the form *`name = value`*.

`resettable` Whether the dialog has a *Reset* button, default `TRUE`.

`defaults` A list of default values containing items with the same names as `values`, to be used if no state information is stored for `dialog`.

putRcmdr(), getRcmdr()

These functions are for storing and retrieving arbitrary information in the `.RcmdrEnv` environment maintained by the **Rcmdr** package. Be careful not to use an existing name, so as not to clobber standard R Commander state information. To check the initial contents of `.RcmdrEnv`, load the **Rcmdr** package and enter the command `ls(envir=Rcmdr:::.RcmdrEnv, all.names=TRUE)` at the R command prompt. For additional discussion of this point, see Section 4.2.

```
putRcmdr(x, value)
```

```
getRcmdr(x, mode="any", fail=TRUE)
```

Arguments:

`x` A character string giving the name under which the information is stored.

`value` Any R object to be stored.

`mode` The R mode of the object to be retrieved; it's generally safe to let this default to `"any"`.

`fail` If `TRUE`, the default, and the named object doesn't exist in `.RcmdrEnv`, then an error results; if, under these circumstances `fail=FALSE`, then `NULL` is returned, without an error.

titleLabel()

Creates a ttk label widget with the R Commander title font and color.

```
titleLabel(...)
```

Argument:

`...` Arguments to be passed to `ttklabel()`.

A.2 Utilities Useful for `onOK()` Button-Callback Functions

`closeDialog()` (macro)

As its name suggests, closes the dialog box (and performs some housekeeping). `closeDialog()` should normally be called somewhere in `onOK()`.

```
closeDialog(window=top, release=TRUE)
```

Arguments:

`window` Tk window to close.

`release` If `TRUE`, call `tkgrab.release()`; it's best to leave this alone.

`doItAndPrint()`, `justDoIt()`, `logger()`

These functions are for processing R commands composed as character strings. By far the most commonly used of these commands is `doItAndPrint()`.

```
doItAndPrint(command, log=TRUE, rmd=log)

justDoIt(command)

logger(command, rmd=TRUE)
```

Arguments:

`command` A character string representing a single complete command; the command can be spread over several lines separated by new-line characters, `"\n"`. Note that a long character string will be automatically split to fit in the *R Script* tab, and so you generally don't have to include new-lines unless these improve the clarity of the command.

`log` Echo the command to the *R Script* tab as well as executing it and printing its output. This should almost always be `TRUE` (the default).

`rmd` Include the command in the R Markdown and knitr LaTeX documents built by the R Commander; the default is the value of `log`—i.e., almost always `TRUE`. Should be set to `FALSE` if the command requires direct user intervention (e.g., interactive point-identification in a graph).

`doItAndPrint()` causes the command to be executed, entered into the R Commander *R Script* tab, optionally entered into the *R Markdown* tab, and entered, along with any printed output generated into the *Output* pane. `justDoIt()` causes the command to be executed without any of the other effects. `logger()` "logs" the command into the *R Script* tab, *R Markdown* tab, and *Output* pane without executing the command.

RcmdrTkmessageBox()

Creates a customized Tk message box and returns the user's response (the name of the button pressed).

```
RcmdrTkmessageBox(message, icon=c("info", "question", "warning",
    "error"), type=c("okcancel", "yesno", "ok"), default, title="")
```

Arguments:

`message` Character string with the text message to display.

`icon` One of four standard icons defined by the R Commander.

`type` Determines the buttons displayed.

`default` The button that's pressed by default, which depends on the `type`. If unspecified, the default button is `"ok"` for `"okcancel"`; `"yes"` for `"yesno"`; `"ok"` for `"ok"`.

`title` Character string giving the title for the message box.

errorCondition() (macro)

Closes the dialog box, prints an error message in the R Commander *Messages* pane, and, typically, reopens the dialog in its previous state.

```
errorCondition(window=top, recall=NULL, message, model=FALSE)
```

Arguments:

`window` Tk window for the dialog, to be closed.

`recall` Text string giving the name of the callback function for the dialog, to be reopened.

`message` Text string giving error message.

`model` If `TRUE` the model number will be decremented by 1, preventing model numbers from being skipped when a model-producing dialog is closed as a consequence of an error.

Message()

Print a message in the R Commander *Messages* pane.

```
Message(message, type=c("note", "error", "warning"))
```

`message` Text string giving the message to be printed.

`type` The type of message, flagged as such in the *Messages* pane.

checkReplace()

Used to check whether an object of a given name already exists, to avoid clobbering an existing object when a user provides an object name in a dialog. `checkReplace()` uses `RcmdrTkmessageBox()` to create a yes/no message box that returns either `"no"` (the default) or `"yes"`. You can then use the answser to decide whether to replace the existing object.

```
checkReplace(name, type=gettextRcmdr("Variable"))
```

Arguments:

`name` The quoted object name to check.

`type` A character string used to customize the message in the box, which is of the form, "*type name* already exists. Overwrite *type*" (with the second appearance of *type* converted to lower-case).

activateMenus()

Causes the activation status of menus and menu-items to be updated. It's rarely necessary to call this functions directly—for example, it's called whenever the active data set changes.

```
activateMenus()
```

setBusyCursor(), setIdleCursor()

If a computation is expected to be time-consuming, calling `setBusyCursor()` draws this fact to the user's attention. The result is system-dependent, but is typically something like an "hour-glass" cursor. Calling `setIdleCursor()` after the computation restores the cursor to its usual state.

```
setBusyCursor()
setIdleCursor()
```

trim.blanks()

Removes one or more initial and trailing blanks (`" "`) from a character string.

```
trim.blanks(text)
```

Argument:

`text` Character string to be trimmed.

popCommand(), popOutput()

The R Commander maintains stacks (last-in, first-out queues) of commands and text output. These commands return the last item in each stack, by default removing it from the stack.

```
popCommand(keep=FALSE)
popOutput(keep=FALSE)
```

Argument:

`keep` If `TRUE`, leave the last item on the stack rather than removing it.

A.3 Working With the Active Data Set and Active Statistical Model

`activeDataSet()`, `ActiveDataSet()`

These functions reset the active data set or retrieve its name. They are partly redundant; both are retained for backwards compatibility. After doing some housekeeping, `activeDataSet()` calls `ActiveDataSet()`.

```
activeDataSet(dsname, flushModel=TRUE, flushDialogMemory=TRUE)
ActiveDataSet(name)
```

Arguments:

`dsname` A character string giving the name of a data frame (or an object coercible to a data frame) that is to become the active data set. If missing, the name of the active data set is returned, or, if there's no active data set, an error message is printed.

`flushModel`, `flushDialogMemory` If a new active data set is specified, normally a record of the active statistical model and any information about dialog states are removed, because such information typically pertains to the previously active data set. Sometimes, however—for example, when a variable is added to the active data set—it would be better to retain this information, in which case you can set one or both of these arguments to `FALSE`.

`name` A character string giving the name of a data frame to become the active data set; if missing, the name of the active data set is returned, or, if there's no active data set, `NULL` is returned.

`Variables()`, `Numeric()`, `Factors()`, `TwoLevelFactors()`

These functions return the saved names or save the names of various classes of variables in the active data set. It's rarely necessary to save variable names explicitly because this is typically done automatically when a data set becomes the active data set or when the active data set is modified. Character and logical variables in the active data set are treated by the R Commander as factors and are included in the names returned by `Factors()`.

Arguments:

`names` A character vector of variable names to be stored. If missing, the names of the variables (or variables of a particular class) in the active data set set are returned—which is typically how these functions are used.

`listVariables()`, `listNumeric()`, `listFactors()`, `listTwoLevelFactors()`

These functions return the names of various classes of variables in a data set. Logical variables and character variables are treated as if they are factors.

```
listVariables(dataSet=ActiveDataSet())
listNumeric(dataSet=ActiveDataSet())
listFactors(dataSet=ActiveDataSet())
listTwoLevelFactors(dataSet=ActiveDataSet())
```

Argument:

`dataSet` A character string giving the name of a data set. If missing, then the saved names of variables for the active data set are returned.

`listDataSets()`

Lists the names of all data frames, by default currently residing in the global environment.

```
listDataSets(envir=.GlobalEnv, ...)
```

Arguments:

`envir` The environment in which to look.

`...` Ignored.

`activeModel()`, `ActiveModel()`

These functions set or retrieve the name of the active statistical model. They are largely redundant; both are retained for backwards compatibility.

```
activeModel(model)
ActiveModel(name)
```

Arguments:

`model` A character string giving the name of the new active statistical model. If missing, the name of the current active statistical model is returned, or, if there is no active model, an error message is printed.

`name` A character string giving the name of the new active statistical model. If missing, the name of the current active statistical model is returned, or, if there is no active model, `NULL` is returned.

`listAllModels()`, `listLinearModels()`, `listAOVModels()`, `listGeneralizedLinearModels()`, `listMultinomialLogitModels()`, `listProportionalOddsModels()`

These functions list the names of all models in classes recognized by the R Commander, or of models in a particular class, by default residing in the global environment.

```
listAllModels(envir=.GlobalEnv, ...)
listLinearModels(envir=.GlobalEnv, ...)
listAOVModels(envir=.GlobalEnv, ...)
listGeneralizedLinearModels (envir=.GlobalEnv, ...)
listMultinomialLogitModels(envir=.GlobalEnv, ...)
listProportionalOddsModels(envir=.GlobalEnv, ...)
```

Arguments:

`envir` The environment in which to look.

`...` Optional arguments, to be passed to the `ls()` function.

A.4 Predicate Functions

A predicate function returns `TRUE` or `FALSE` depending upon whether some condition obtains. Many R Commander predicate functions end with `"P"` (e.g., `activeDataSetP()`, `factorsP()`), and either take no arguments or have only an optional argument. Some R Commander predicate functions have names beginning with `"check"` (e.g., `checkFactors()`).

Predicate functions can be useful for determining whether a menu item should be installed, or, if installed, should be activated under current circumstances. They may also be useful for checking various conditions in dialog callback functions.

A.4.1 Predicates Associated With Data Sets

`activeDataSetP()`	Is there an active data set?
`variablesP(n=1)`	Are there at least `n` variables in the active data set?
`checkVariables(n=1)`	Are there at least `n` variables in the active data set? Additionally print an error message if there are not.
`numericP(n=1)`	Are there at least `n` numeric variables in the active data set?
`checkNumeric(n=1)`	Are there at least `n` numeric variables in the active data set? Additionally print an error message if there are not.
`factorsP(n=1)`	Are there at least `n` factors in the active data set?
`checkFactors(n=1)`	Are there at least `n` factors in the active data set? Additionally print an error message if there are not.
`twoLevelFactorsP(n=1)`	Are there at least `n` two-level factors in the active data set?
`checkTwoLevelFactors(n=1)`	Are there at least `n` two-level factors in the active data set? Additionally print an error message if there are not.
`dataSetsP(n=1)`	Are there at least `n` data sets in memory?

A.4.2 Predicates Associated With Statistical Models

`modelsP(n=1)`	Are there at least `n` statistical models recognized by the R Commander in memory?
`activeModelP()`	Is there an active statistical model?
`lmP()`	Is the current statistical model an `"lm"` or `"aov"` object? `FALSE` if there is no active model.
`glmP()`	Is the current statistical model a `"glm"` object? `FALSE` if there is no active model.
`multinomP()`	Is the current statistical model a `"multinom"` object? `FALSE` if there is no active model.
`polrP()`	Is the current statistical model a `"polr"` object? `FALSE` if there is no active model.

A.4.3 Predicates Associated With Operating Systems

`WindowsP()`	Is the R Commander running under Windows?
`X11P()`	Is the R Commander running under X-Windows?
`RappP()`	Is the R Commander running under Rapp on Mac OS X?
`MacOSXP(release)`	Is the R Commander running under Mac OS X? `release` is an optional version string; if specified returns `TRUE` if the Mac OS X version $\geq$ `release`.

A.4.4 Other Predicates

`checkClass(object, class, message=NULL)` (macro)

Returns `TRUE` if the *primary* S3 class of an object matches.

`object` An R object.

`class` Character string naming an S3 class.

`message` Optional character string giving error message to print if the class doesn't match; if `NULL` a generic error message is composed.

`exists.method(generic, object, default=TRUE, strict=FALSE)`

Returns `TRUE` if an appropriate S3 method is located.

`generic` The quoted name of an R S3 generic function.

`object` An R object.

`default` Is it OK to use the default method for the given generic?

`strict` Is it *not* OK to inherit a method from another class?

`checkMethod(generic, object, message=NULL, default=FALSE, strict=FALSE, reportError=TRUE)` (macro)

Returns `TRUE` if an appropriate S3 method is located; calls `exists.method()`.

`generic` The quoted name of an R S3 generic function, typically (but not necessarily) applicable to statistical models.

`object` An R statistical-model (or other) object.

`message` Optional character string giving error message to print if the class doesn't match; if `NULL` a generic error message making reference to a statistical-model class is composed.

`default` Is it OK to use the default method for the given generic?

`strict` Is it *not* OK to inherit a method from another class?

`reportError` Print an error message if a match isn't found.

`is.valid.name(x)`

Returns `TRUE` if the character string `x` is a valid R object name.

`is.valid.number(string)`

Returns `TRUE` if the character string `string` can be coerced to a valid (non-missing) numeric object (e.g., a single number or a numeric vector).

`packageAvailable(name)`

Is the package `name` (given as a character string) available in an accessible library?

References

P. Dalgaard. A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31, 2001. http://cran.r-project.org/doc/Rnews/Rnews_2001-3.pdf.

P. Dalgaard. Changes to the R-Tcl/Tk package. *R News*, 2(3):25–27, 2002. https://cran.r-project.org/doc/Rnews/Rnews_2002-3.pdf.

J. Fox. Extending the R Commander by "plug-in" packages. *R News*, 7(3):46–52, 2007.

J. Fox. *Using the R Commander: A Point-and-Click Interface for R*. Chapman & Hall/CRC Press, Boca Raton FL, 2017.

J. Fox and M. Carvalho. The **RcmdrPlugin.survival** package: Extending the R Commander interface to survival analysis. *Journal of Statistical Software*, 49(1):1–32, 2012. URL http://www.jstatsoft.org/index.php/jss/article/view/v049i07.

M. E. Lawrence and J. Verzani. *Programming Graphical User Interfaces in R*. Chapman and Hall/CRC Press, Boca Raton FL, 2012.

T. Lumley. Programmer's niche: Macros in R. *R News*, 1(3):11–13, 2001.

J. K. Ousterhout and K. Jones. *Tcl and the Tk Toolkit*. Addison-Wesley, Upper Saddle River NJ, second edition, 2010.

R Core Team. *Writing R Extensions*, 2016. Version 3.3.1.

B. D. Ripley. Internationalization features of R 2.1.0. *R News*, 5(1):2–7, 2005.

G. Snow. ***TeachingDemos****: Demonstrations for Teaching and Learning*, 2016. URL https://CRAN.R-project.org/package=TeachingDemos. R package version 2.10.

T. M. Therneau. *A Package for Survival Analysis in S*, 2015. URL http://CRAN.R-project.org/package=survival. version 2.38.

T. M. Therneau and P. M. Grambsch. *Modeling Survival Data: Extending the Cox Model*. Springer, New York, 2000.

H. Wickham. *R Packages: Organize, Test, Document, and Share Your Code*. O'Reilly, Sebastopol CA, 2015.

Author Index

Subject Index