RcmdrPlugin Easy Script Templates 1.0.1

Ewoud De Troyer

Contents

1 Introduction

R is a great platform for statisticians to do their analyses and data manipulation. Thanks to the existence of R packages, new code can be easily distributed and executed by other statisticians.
However for other scientists, the barrier to understand the coding language of R, can sometimes be too difficult to breach. This is a shame since it means that all of the implemented methodology in R, is inaccessible for them. One way to bridge this gap is through the use of *Graphical User Interfaces* (GUI). While the R-code is still responsible for the analyses in the background, the user does not need to worry about it since the methods can be accessed through simple point and click.

However making a GUI can take up quite some time. Learning the syntax of creating windows, creating the actual code, etc. Sometimes there is simply no time left to invest in this exercise. This is where the `REST` package (or **R**`cmdrPlugin` **E**`asy` **S**`cript` **T**`emplates`) comes into play. `REST` contains an easy script template to create new dialogs in the form of a plug-in for R Commander (`Rcmdr`). These scripts do not require any knowledge about `tcltk` or `Rcmdr` and are very straightforward to use. They will allow developers to translate the R-code from their packages into a GUI without too much difficulty.
This means that the `REST` is not meant to be used independently. It should be imported or depended upon in your own GUI package.

R Commander was chosen as the starting platform for the GUI's since it will also show the R code (from the package the GUI is based on) going on in the background. For example, if the user would click on a plot button, the original code '`plot(...)`' would appear in the script window. Users can simply decide to ignore this or, even better, use it to start learning the syntax of R while using GUI's.

It should be mentioned that the scripts used in this package are a generalisation of the templates scripts which were used in the `RcmdrPlugin.BiclustGUI` package.

2 R Commander

R Commander, `Rcmdr`(Fox, 2005), is a GUI developed by John Fox from McMaster University, Canada. Originally it was conceived as a basic-statistics graphical user interface for R, but its capabilities have been extended substantially since. The `Rcmdr` package is based on the `tcltk` package (Dalgaard, 2001b) which provides an R interface to the *Tcl/Tk* GUI builder. Since `tcltk` is available on all the operating systems on which R is commonly run, the R Commander GUI will also run on all of these platforms.

The GUI is also very easy to start to use for beginners who do not have any or little experience with R. It will protect beginners from errors as the dialog boxes only have limited options related to the current context. Further, since the users are exposed to the actual R commands through a script and output window, besides analyzing and managing the data in R easily, they can also learn how do it in R without a GUI. Another advantage is that the script will be generated on the fly as the user applies the desired statistics through the point-and-click GUI. This means it can be easily saved at the end of a session which enables the user afterwards to recreate the results by running the R script without going through all the dialogs again. Advanced users can even adapt the created script to do some more detailed analysis. These are the main advantages `Rcmdr` has over other available RGUI packages.

Starting with version 1.3-0, `Rcmdr` also provides the possibility of *plug-in* packages which can augment the R Commander menus. These packages are developed, maintained, distributed and installed indepently of `Rcmdr` and can provide a wide variety of new dialog boxes and statistical functionality. More information on developing such a plug-in can be found in Fox (2007).

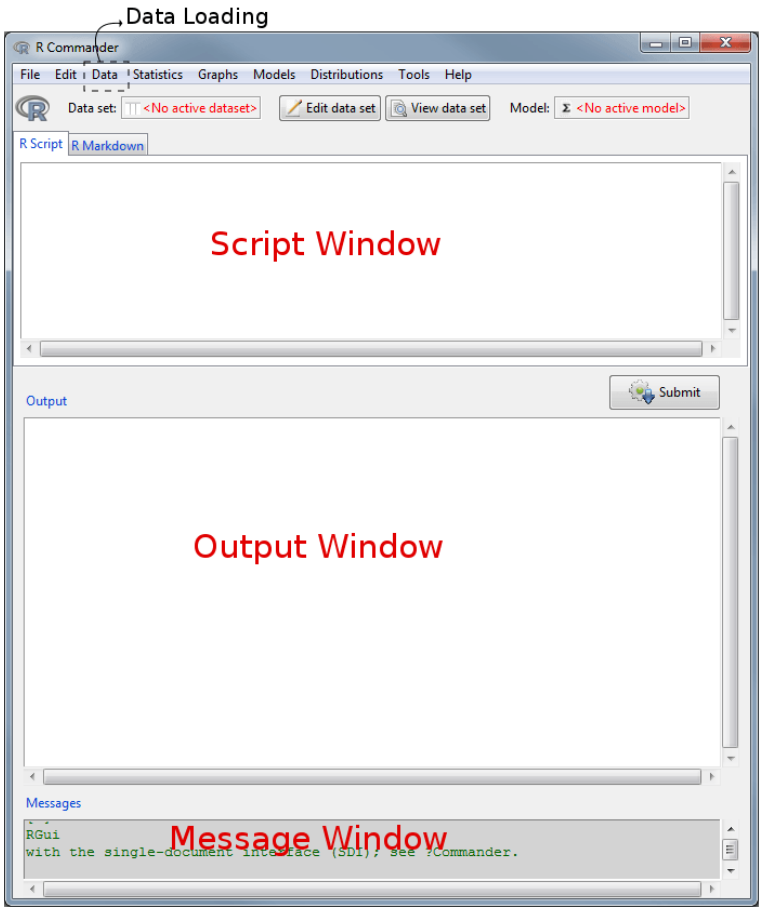

Figure 1: *Default R Commander*

3 Other GUI Creation Packages

There exist several packages in R to help users create GUI's. Some examples are `Rgtk2` (Lawrence and Lang, 2014) and `gWidgets` (Verzani, 2014). Especially the latter already operates at a fairly high level to help users create quick and interactive GUI's without too much fuss. The `REST` package differs in that it works on an even higher level by using a "fill-out" script. This reduces the learning curve even more. The price to pay however is that it is less flexible than than a package like `gWidgets`.
Another difference lies in the fact that `REST` is completely tied to `Rcmdr`, providing all the benefits R Commanders brings in. This includes things like the fully functional system to load in data from several sources, the implemented basic statistics and of course the existence of the script window. The latter might be especially helpfull if you already have an existing package you want to translate into a GUI as it provides a nice teaching tool to learn to use your original package.
So while you lose power in creating your GUI, a lot of the basic things you need to implement when making a GUI are already there, gaining you time instead.

At the end of the day, it will always be the developer who needs to decide which of these packages, fits him/her the most. It will always depend on how much time is availabe and what the end goal is precisely.

4 Creating a GUI for `Rcmdr`

R Commander is already a fully implemented GUI in which basic statistics can be executed. Creating a plug-in comes down to adding new menus and submenus at the top of the window which will lead to newly created dialogs.
Each window you create will simply be an R function. How to create them with the template script will be explained in the next section, but first let's take a look at how you can add these extra menu's.

4.1 Menu file

Before compiling your GUI package, a `menus.txt` file should be added in the `yourpackage/inst/etc/` folder. This text file will contain the information on which and how menu's should be added. The basics will be explained here with the help of an easy example, but more detailed information can be found in Fox (2007).

The text file should contain 7 columns: type, menu/item, operation/parent, label, command/menu, activation and install.

Let's now go through the example of `menus.txt` down below which will result in the menu's in Figure 2. In the first line, *menu* was chosen in the first column to define the *NEWmenu* (second column) menu which should be a part of the *topMenu* (third column), meaning it will appear next to the other big menus in R Commander. In the second line, this new menu will actually be installed by "cascading" the menu under its parent. This is achieved by having *item*, *topMenu* (parent) and *cascade* in the first 3 columns, followed by *NEWmenu* (menu) in the 5th. It is also in this line you can actually give this new menu the label it will have in the GUI by using the 4th column.
Now after we defined a new menu, we can start adding some items. The following 3 lines are all 3 *item*'s in *NEWmenu* (1st and 2nd column). The first two have *command* as operation and a certain label which will appear in the GUI. The 5th column then has the actual command tied to this menu item. These will be your window functions (between double quotes) you have created with the template scripts. The third item has *separator* has the operation. This simply means a line will be added to the menu.
Next, just as we defined a new menu in the *topMenu*, we can also define and install a new submenu, *NEWsubmenu*, in *NEWmenu*. Afterwards, we can again make some new items in this new submenu. All of this is done in just the same way as before.
Finally, you can also add an R function between double quotes in the activation column. These should be functions which give back `TRUE` or `FALSE`. If `FALSE` is given back, the menu item will be grayed out, rendering the user unable to click on it. For example `activeDataSetP()` is an `Rcmdr` function which gives back a boolean value whether there is an active data set or not. In the Figure 2 you can see there is no active data set, which results in 'namewindow4' being inactive.

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

### DEFINE NEW TOP MENU
menu    NEWmenu     topMenu            ""                 ""                  ""                  ""
item    topMenu     cascade            "Name of New Menu" NEWmenu             ""                  ""

### New items
item    NEWmenu     command            "namewindow1"      "window1_function"  ""                  ""
item    NEWmenu     command            "namewindow2"      "window2_function"  ""                  ""
item    NEWmenu     separator          ""                 ""                  ""                  ""

### Submenu
menu    NEWsubmenu  NEWmenu            ""                 ""                  ""                  ""
item    NEWmenu     cascade            "nameofsubmenu"    NEWsubmenu          ""                  ""

### New items in submenu
item    NEWsubmenu  command            "namewindow3"      "window3_function"  ""                  ""
item    NEWsubmenu  command            "namewindow4"      "window4_function"  "activeDataSetP()"  ""
```

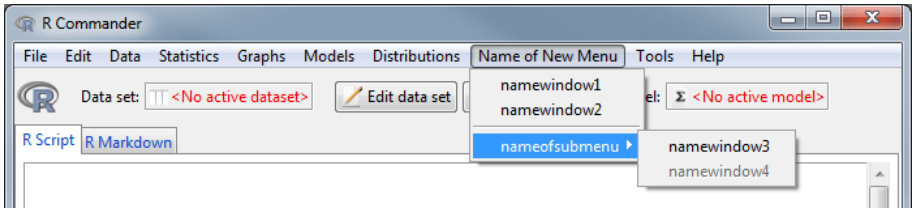

Figure 2: Exame of Menu Creation

4.2 `.onAttach`

In order for your package to be recognised by `Rcmdr` as a plug-in, you will need to add the following `.onAttach` function to your package (see Appendix).

4.3 DESCRIPTION and NAMESPACE File

In order to use templates of the REST package in your own package, you should import both Rcmdr and REST in the DESCRIPTION and NAMESPACE File.
This comes down to adding `Imports: Rcmdr, REST` to the former and `import(Rcmdr,REST)` to the latter.

4.4 Active Data in Rcmdr

There are many ways to load data in R Commander, from the R workspace, from a text file, excel file, SAS, etc. The important thing to know is that the loaded data will become the active dataset in Rcmdr. This active dataset will always be of the dataframe class, so it could be possible you will need to transform it to a matrix if your function requires this.

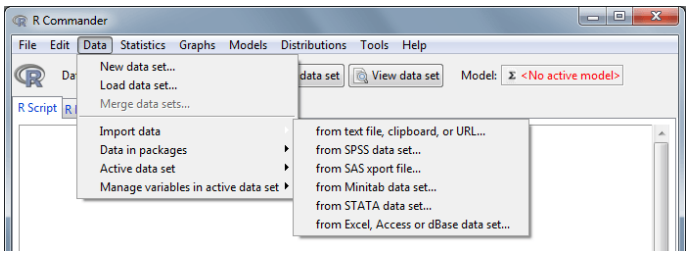

Figure 3: *R Commander - Data Menu*

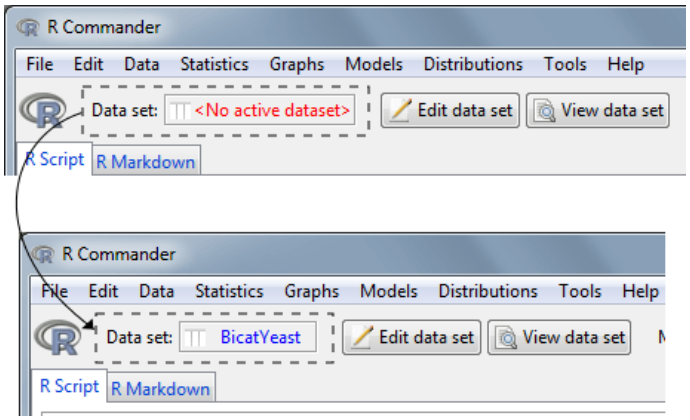

Figure 4: *R Commander - Active Dataset*

Short Summary:

1. DESCRIPTION and NAMESPACE file
2. Add `.onAttach` function
3. Create window functions with *template scripts*
4. Add window functions to `menus.txt`

5 Script Templates Guide

5.1 General Script

We will now start going through `general_script.R` which can be found in the appendix or the `doc` folder of the REST package.

5.1.1 General Window Information

The script starts by making a function which will be called on to create a window. First thing you should do of course is to rename this function to your own liking. Next, some objects (`new.frames`, `grid.config` and `grid.rows`) are initialized that will be used to store information about the window that you are about to create.

```
GUI_WINDOW <- function(list.info=list()){

        ############################
        ## PREAMBLE/INFORMATION ##
        ############################

        dialogtitle <- "This is the title of the window"

        usetabs <- TRUE
        tabnames <- c("Tab 1","Tab 2","Tab 3")
        helppage <- "plot"

        # Do not change these lines
        if(usetabs){ntabs <- length(tabnames)} else {ntabs <- 1}
        new.frames <- .initialize.new.frames(ntabs)
        grid.config <- .initialize.grid.config(ntabs)
        grid.rows <- .initialize.grid.rows(ntabs)
        ### end of "Do not change these lines"

        ###################
        ## GRID BUTTONS ##
        ###################

        make.help.button <- TRUE
        make.setwd.button <- TRUE
        make.resetgws.button <- TRUE
        make.seed.button <- TRUE

        # ... continuation of the script down below (these 2 parts are put here)

}       # Note: The curly bracket is placed here for syntax reasons.
        #       It should be placed after the call of GUI_template.
```

The scripts starts by filling in some information about your new window. A clarifying example follows later in which we make a window of the biclustering plaid method.

- `dialogtitle`: The title of the window which will be shown on top. (This can not be empty!)
- `usetabs`: Logical value determining if tabs should be used.
- `tabnames`: A vector containing names of the tabs if `usetabs` is `TRUE`.
- `helppage`: The name of the helppage the help button should be directed to. (`help(helppage)`) This is only relevant if the help button is created in the grid.

After filling in these variables, you also have the possibility to add some grid buttons. These are some standard buttons which will appear at the bottom of your window or, if you are using multiple tabs, below all the tab windows (Figure 5). While the exit button will always be there, you can add some additional ones by setting the following variables to `TRUE` or `FALSE`.

- `make.help.button`: A help button which leads to the help page defined by `helppage`.
- `make.setwd.button`: A button with which the user can change its working directory.
- `make.resetgws.button`: A button with which the user can reset the global working space.
- `make.seed.button`: Adds an entry field and button to set a certain seed.

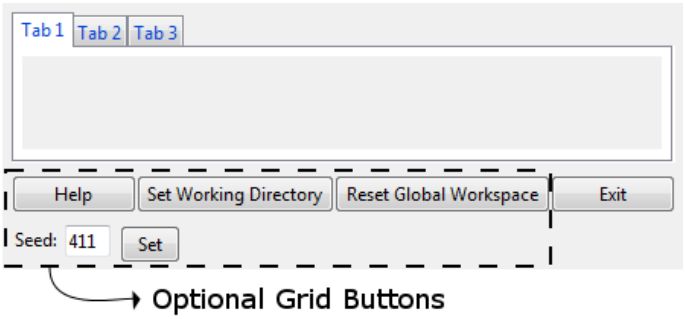

Figure 5: Optional Grid Buttons

Before going on to the next part of the script, a short explanation about `list.info` (parameter of `GUI_WINDOW` function) might be in order. You do not necessarily have to use this, but it can bring a bit more flexibility to your windows. This is especially the case when you are calling a window from another window.
For example, let's say you have created a dialog for a certain graph called `graph_window`. You have not added it to the menu, but the function is linked to a button in another window which will call `method1_window`. You however know that this graphing might also be interesting for method 2 so you also link it to a button in `method2_window`. Your goal now is for the graph window to have a different dialog title, depending from which window it was called from. You can achieve this by storing this information in `list.info`.
You could then use `dialogtitle <- paste("Graph of ",list.info[[1]],sep="")` so that when you call `GRAPH_WINDOW(list(name="Method 1"))` from the button in the method 1 window, the title will reflect this ("Graph of Method 1").
This is of course a very simplistic example, but you can use this for all of the variables in the script, creating very different windows depending on which information is stored in `list.info` (e.g. different frames, different naming, etc.).

5.1.2 Making a Tab

After providing the information about the window we can finally start making it! You can make as many tabs as you want, but they are all created in the same three easy steps as shown in Figure 6:

1. Making the frames
2. Configuring the frames into a grid
3. Combining rows into a box

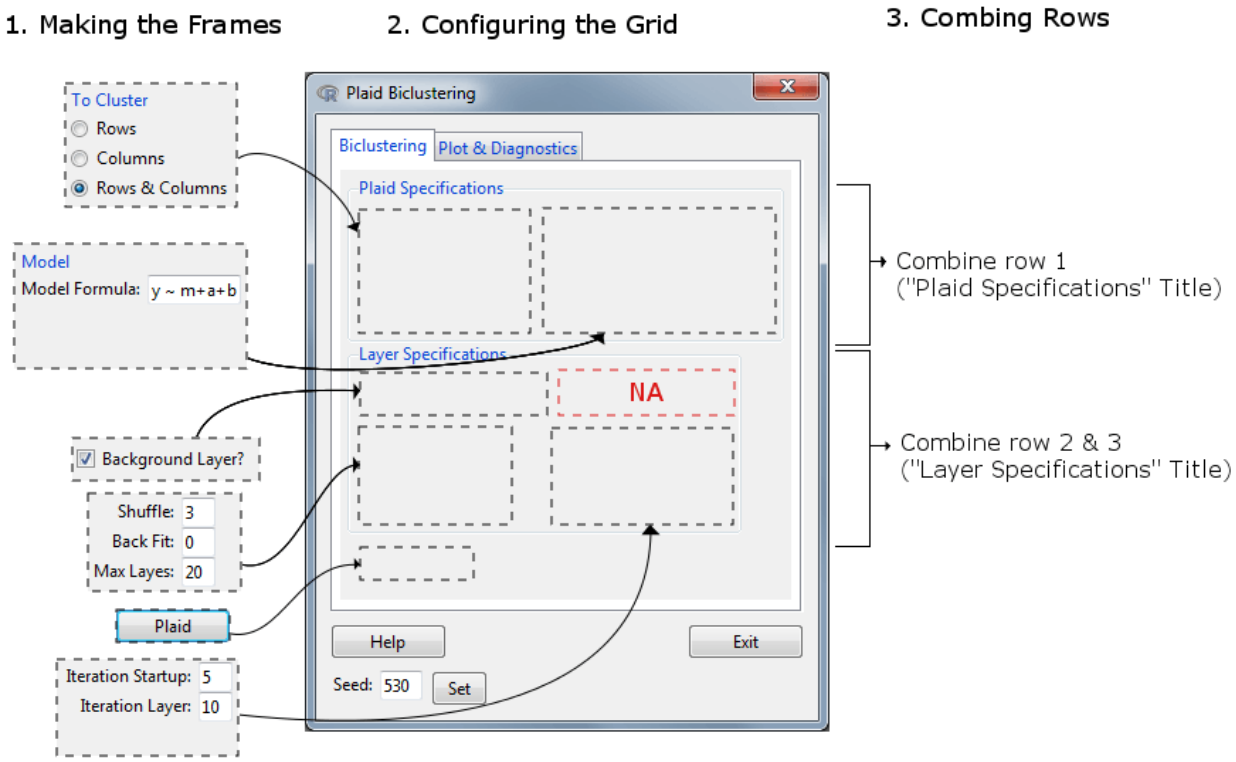

Figure 6: *Making windows in 3 steps*

```
############
## TAB 1 ##
############
Tab <- 1

### 1. ADDING THE FRAMES ###

# Add frames here


### 2. CONFIGURING THE GRID ###
grid.config <- .grid.matrix(Tab=Tab,c("frame1","frame2","frame3",NA),
                byrow=TRUE,nrow=2,ncol=2,grid.config=grid.config)


### 3. COMBINING THE ROWS ###
grid.rows <- .combine.rows(Tab=Tab,rows=c(1,2),title="A nice box: ",
                border=TRUE,grid.rows=grid.rows,grid.config=grid.config)
```

Looking at the script, you can see it starts with putting the `Tab` to 1. This will make sure everything you are creating and saving now will be done for the first tab.

Step 1:
As already explained earlier, the first step will be to create the frames in which you want to put your function arguments. A variety of frames can be created, but these will be explained in more detail in the following section. To give a quick summary, here is the list of the types of frames which can be generated.

- Check Boxes
- Radio Buttons
- Entry Fields
- Sliders

- Spinboxes
- List Box
- Manual Button

In future updates, there is still the possibility to add even more types if required.

Step 2:
During the creation of the frames in the previous step, you will have given each of them a unique name. Using these framenames, the next step will be to simply order them into a matrix grid, filling in the empty spots with `NA`'s. This is achieved with the `.grid.matrix` function. The function accepts the exact same arguments as the `matrix` function apart from two new ones, namely `Tab` and `grid.config`. The first is to make sure the template function knows we are adding frames in the first tab, while second one is there to ensure that the new information is added to the old `grid.config` object and that old information is not lost.
Further, it is important to know that the inserted frames will *always* be pulled towards the north-west as much as possible. Therefore in a 1-row matrix, something like `c(NA,"frame1")` or `c("frame1",NA)` would give exactly the same result.

Step 3:
The final step will enable you to put one or multiple rows in a seperate box which can serve two different purposes. The first, being the most obvious one, is simply to add some visual distinction between rows with the help of a title. This can be with or without a border around the row(s).
The second purpose is connected to the way frames are added in this *grid*. Sometimes if frames have a large difference in size, other frames might seem to be jumping to the right, trying to fit in one general grid. In general if you see this happening, putting this row(s) in a box will solve this problem and the frames will again be pulled towards the left.
Creating these boxes by combining rows is again very easy, one simply needs to use the `.combine.rows` function which will save the necessary information in the `grid.rows` object. The function only has three arguments you should change: `rows` which is a vector containing the rows you wish to combine, `title` to give the box a title (`""` means no title) and `border` to decide if there should be a border.
Note that in contrast to the grid configuration, you can call this function multiple times until the desired result is obtained.

```
##############
### TAB 2 ###
##############
Tab <- 2

# Repeat the 3 steps of tab 1 for as many tabs as you like...


#####################################################################
## USE ALL THE ARGUMENTS IN THE GENERAL GUI_TEMPLATE FUNCTION    ##
#####################################################################
GUI_template(dialogtitle=dialogtitle,helppage=helppage,make.resetgws.button=
                make.resetgws.button,make.setwd.button=make.setwd.button,
                make.help.button=make.help.button,make.seed.button=make.seed.button,
                usetabs=usetabs,tabnames=tabnames,grid.config=grid.config,grid.rows=grid.rows,
                new.frames=new.frames)
```

Next, you can repeat these 3 steps for as many tabs as you have defined in the beginning. Finally, to end the `GUI_WINDOW` function, the `GUI_template` function is called with all of your defined variables. This is our automated template function we created in the `REST` package in order to complete the window creation. This means that this function will be responsible for actually creating your window.

5.2 Frame Scripts

In this section, the several types of frames which can be used in the `general_script` will be showcased. The idea is that these parts of the R-code (which are also in the Appendix) are copy-pasted into the `general_script` and are adjusted as deemed necessary.
All the frame types have the `title` and `border` option in common. The results of these options can be seen in

Figure 7. Also note that for each frametype the information is saved in one object, namely `new.frames`. Just as the grid and row configuration earlier, new information will keep on getting added to this object, now with the help of the `.add.frame` function. Lastly, at the start of each frame script, a `type` variable will be set to determine the type of frame for this previous mentioned `.add.frame` function.

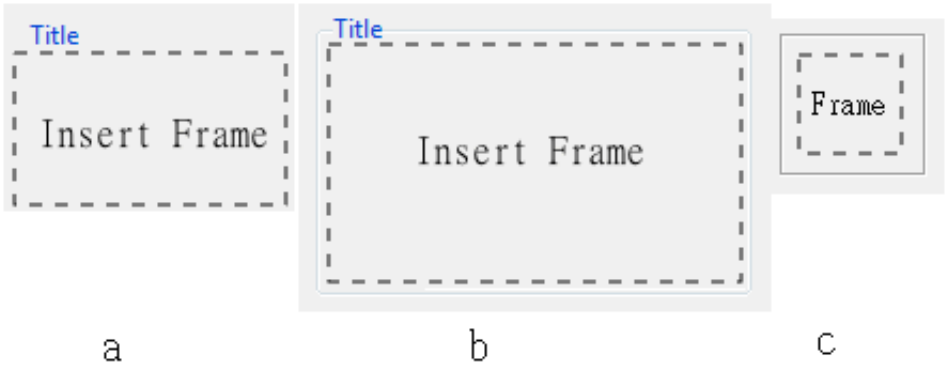

Figure 7: **A.** *Title and No Border* **B.** *Title and Border* **C.** *No Title and Border*

Entry Fields

The first type of frame is the entry fields frame. It can be used for both numerical arguments and character arguments of your function tied to a button. Multiple entries can be added in one frame which will be placed below each other.

```
#### ENTRY FIELDS FRAME ####

type <- "entryfields"

# Change variables accordingly:
frame.name <- "entryframe1"
argument.names <- c("Argument 1","Argument 2","Argument 3")
argument.types <- c("num","num","char")
arguments <- c("arg1","arg2","arg3")
initial.values <- c(1,2,"a")
title <- "A Title"
border <- FALSE
entry.width <- c("2","2","6")

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,type=type
   ,frame.name=frame.name,argument.names=argument.names
   ,arguments=arguments,initial.values=initial.values
   ,title=title,border=border,entry.width=entry.width
   ,argument.types=argument.types  ,new.frames=new.frames)
```

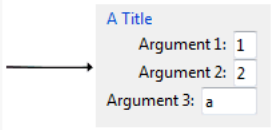

Figure 8: *Entry Fields: Code + Example*

Entry Fields Variables:

- `frame.name`: The unique name of this frame. (Which is used in the grid matrix)
- `argument.names`: The argument names how they will appear in the window.
- `argument.types`: A vector defining if the argument is `"num"` or `"char"`. This basically just means if there should be a `' '` around the value when filling it in in the function. (e.g. In Figure 8 the arguments would be filled in as `,arg1=1,arg2=2,arg3='a'`)
- `arguments`: The actual argument names, used for the function. Note it is not necessary for these to be unique between multiple frames.
- `initial.values`: A vector containing the initial values in the entry fields.
- `title`: Optional title for the frame (`""` means no title)
- `border`: Logical value determining the presence of a border.
- `entry.width`: A vector containing the width of the entry fields (1 width = 1 number/character).

Check Boxes

The second type of frame is the check boxes frame which is used for `TRUE`/`FALSE` arguments. Just like for entry fields, multiple check boxes can be added below each other.

```
#### CHECK BOXES FRAME  ####

type <- "checkboxes"

# Change variables accordingly:
frame.name <-  "checkboxframe1"
argument.names <- c("Check 1","Check 2","Check 3")
arguments <- c("checkarg1","checkarg2","checkarg3")
initial.values <- c(0,1,1)
title <- "title"
border <- FALSE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type
   ,frame.name=frame.name,argument.names=argument.names
   ,arguments=arguments,initial.values=initial.values
   ,title=title,border=border,new.frames=new.frames)
```

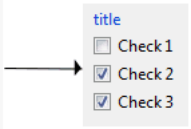

Figure 9: *Check Boxes: Code + Example*

Check Boxes Variables:

- `frame.name`: The unique name of this frame. (Which is used in the grid matrix)
- `argument.names`: The argument names how they will appear in the window.
- `arguments`: The actual argument names, used for the function. Note it is not necessary for these to be unique between multiple frames.
- `initial.values`: A vector containing the initial values in the entry fields. (0 for `FALSE`, 1 for `TRUE`)
- `title`: Optional title for the frame (`""` means no title)
- `border`: Logical value determining the presence of a border.

Radio Buttons

The next type is radio buttons, which is used for only one argument with a finite number of values.

```
#### RADIO BUTTONS FRAME  ####

type <- "radiobuttons"

# Change variables accordingly:
frame.name <- "radioframe1"
argument.names <- c("Button 1","Button 2","Button 3")
arguments <- c("buttonarg")
argument.types <- "char"
argument.values <- c("b1","b2","b3")
initial.values <- "b3"
title <- "Button Options"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type
   ,frame.name=frame.name,argument.names=argument.names
   ,arguments=arguments,argument.values=argument.values
   ,initial.values=initial.values,title=title,border=border
   ,new.frames=new.frames,argument.types=argument.types)
```

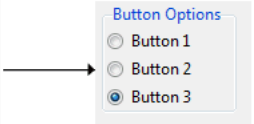

Figure 10: *Radio Buttons: Code + Example*

Radio Buttons Variables:

- `frame.name`: The unique name of this frame. (Which is used in the grid matrix)
- `argument.names`: The names of the buttons how they will appear in the window.

- `arguments:` The actual argument name, used for the function. Note it is not necessary for these to be unique between multiple frames. (Only 1 argument!)
- `argument.types:` Just as for the entry fields, this will determine of the values are filled in with or without `' '`. The two options are again `"num"` and `"char"`, but in contrast with the entry fields it is now only one value and not a vector.
- `argument.values:` The actual values of the radio buttons that correspond to the values passed to the function.
- `initial.values:` The initial value of the radio buttons. It will determine which button is selected on opening the window.
- `title:` Optional title for the frame (`""` means no title)
- `border:` Logical value determining the presence of a border.

Value Sliders
The following type will create value sliders which can only be used for numerical values. Again multiple sliders can be placed under each other. The current value of the slider will always appear on top of it.

```
#### VALUE SLIDER FRAME  ####

type <- "valuesliders"

# Change variables accordingly:
frame.name <- "sliderframe1"
argument.names <- c("Slider 1  ","Slider 2  ","Slider 3  ")
arguments <- c("sliderarg1","sliderarg2","sliderarg3")
initial.values <- c(1,5,10)
from <- c(1,1,1)
to <- c(5,50,500)
by <- c(1,10,50)
length <- c(50,100,150)
title <- "Title"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
   title=title,border=border,frame.name=frame.name,
   argument.names=argument.names,arguments=arguments,
   initial.values=initial.values,from=from,to=to,by=by,
   length=length,new.frames=new.frames)
```

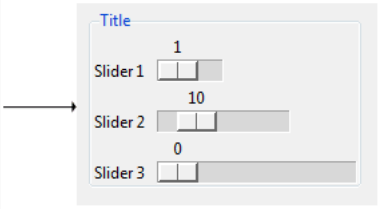

Figure 11: *Value Slider: Code + Example*

Value Sliders Variables:

- `frame.name:` The unique name of this frame. (Which is used in the grid matrix)
- `argument.names:` The argument names how they will appear in the window.
- `arguments:` The actual argument names, used for the function. Note it is not necessary for these to be unique between multiple frames.
- `initial.values:` Vector of initial values of the sliders. Depending on the possible values the slider can take, this value will shift towards it (e.g. Slider 3: 10 as initial value but it was shifted to 0 since this is the closest value the slider represent).
- `from:` Vector of starting points of the slider (note: depending on the `length`, `to` and `by` parameter, this 'from' value could change slightly. It will choose the closest and most fitting value to display the slider properly. e.g. Slider 3 starts from 0 instead of 1).
- `to:` Vector of ending points of the sliders.
- `by:` Vector with the values determining how one movement of the sliders will change the current value.
- `length:` Vector containing the lengths of the sliders.
- `title:` Optional title for the frame (`""` means no title)
- `border:` Logical value determining the presence of a border.

Spin Boxes
This type will create spin boxes which are again solely used for numerical values. Just as for sliders, multiple spin boxes can be placed below each other.

```
#### SPIN BOX FRAME  ####

type <- "spinboxes"

# Change variables accordingly:
frame.name <- "spinboxframe1"
argument.names <- c("Spin Box 1: ","Spin Box 2: ","Spin Box 3: ")
arguments <- c("spinarg1","spingarg2","spingarg3")
initial.values <- c(5,10,20)
from <- c(1,5,10)
to <- c(10,20,30)
by <- c(1,1,1)
entry.width <- "2"
title <- "Spin Box !"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
   frame.name=frame.name,argument.names=argument.names,
   arguments=arguments,initial.values=initial.values,
   from=from,to=to,by=by,entry.width=entry.width,
   title=title,border=border,new.frames=new.frames)
```

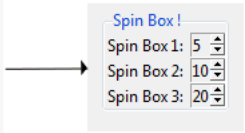

Figure 12: *Spin Boxes: Code + Example*

Spin Boxes Variables:

- `frame.name`: The unique name of this frame. (Which is used in the grid matrix)
- `argument.names`: The argument names how they will appear in the window.
- `arguments`: The actual argument names, used for the function. Note it is not necessary for these to be unique between multiple frames.
- `initial.values`: Vector of initial values of the spin boxes.
- `from`: Vector of starting points of the spin boxes.
- `to`: Vector of ending points of the spin boxes.
- `by`: Vector with the values determining how much one click will change the current value.
- `entry.width`: Width of all the spinboxes (one value which applies to all of them)
- `title`: Optional title for the frame (`""` means no title)
- `border`: Logical value determining the presence of a border.

List Box
The next type is called a list box. This box corresponds with 1 argument for which several values are available. With the list box it is possible to select one or multiple from the box, putting them in a vector.

```
#### LIST BOX FRAME ####

type <- "listbox"

# Change variables accordingly:
frame.name <- "listboxframe1"
arguments <- "listboxarg"
argument.names <- c("Value 1","Value 2","Value 3")
argument.values <- c("value1","value2","value3")
argument.types <- "char"
initial.values <- c("value3")
length <- 4
select.multiple <- FALSE
title <- "A list box:"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
        frame.name=frame.name,argument.names=argument.names,
        arguments=arguments,argument.values=argument.values,
        argument.types=argument.types, initial.values=initial.values,
        length=length,select.multiple=select.multiple,
        title=title,border=border,new.frames=new.frames)
```

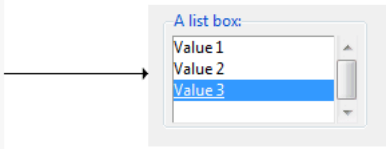

Figure 13: *List Box: Code + Example*

List Box Variables:

- `frame.name`: The unique name of this frame. (Which is used in the grid matrix)
- `arguments`: The actual argument name, used for the function. Note it is not necessary for these to be unique between multiple frames. (Only 1 argument!)
- `argument.names`: The labels of the items how they will appear in the box.
- `argument.values`: The actual values of the list items that correspond to the values passed to the function.
- `argument.types`: Just as for the entry fields, this will determine of the values are filled in with or without `' '`. The two options are again `"num"` and `"char"`, but in contrast with the entry fields it is now only one value and not a vector. (Same as for radio buttons)
- `initial.values`: The initial value of the list box. It will determine which list item is selected on opening the window.
- `length`: A value corresponding with the height of the box. (If no value or `c()` is given, the length of the `argument.names` will be used)
- `select.multiple`: A logical value determining if it should be possible selecting only 1 or multiple list items. For example if the `argument.types` would be `"char"` and `select.multiple` would be `FALSE`, then an example would be simply `'value1'`. If the latter would have been `TRUE`, the result would look like `c('value1','value3')`. (In the case those were selected)
- `title`: Optional title for the frame (`""` means no title)
- `border`: Logical value determining the presence of a border.

Manual Buttons

The last type of frame which can be utilized, is making manual buttons. There are two primary uses for these buttons. The first use is to simple execute a function, based on the arguments of other frames in the window. The second application is to tie the button to another window function to open up more options.

```
#### MANUAL BUTTONS FRAME ####

type <- "buttons"

# Change variables accordingly:
frame.name <- "buttonframe1"
button.name <- "Button 1"
button.function <- "buttonfunction"
button.data <- "d"
button.object <-  "saveobject"
button.width <- "12"
button.data.transf <- "matrix" # only matrix available here !

arg.frames <- c("frame1","frame2")

save <- TRUE
show.save <- TRUE
show <- TRUE
button.otherarg <- ""  # always start with a ,

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,frame.name=frame.name,
        type=type,button.name=button.name,button.width=button.width,
        button.data.transf=button.data.transf,
        button.function=button.function,button.data=button.data,
        button.object=button.object,button.otherarg=button.otherarg,
        arg.frames=arg.frames,save=save,show=show,show.save=show.save,
        new.frames=new.frames)
```

Figure 14: *Manual Button: Code + Example*

Manual Button Variables:

- `frame.name`: The unique name of this frame. (Which is used in the grid matrix)
- `button.name`: The text which will appear on the button.
- `button.function`: A string of the name of the function which should be tied to this button. Another useful practice is to actually make an entire new function for this manual button. This new function could then for example contain a series of functions which would then be carried out all at the same time when clicking on this button.
- `button.data`: The name of the data argument the button function. The data which is loaded in R Commander will then be pasted after this argument. (Simply put `""` when this is not necessary)
- `button.object`: If it is chosen to save what is returned by `button.function`, it will saved in an object with the name given here.
- `button.width`: Character containing the width of the button. (Default = `"12"`)
- `button.data.transf`: Character determining if the data for `button.data` should be transformed. (Only `"matrix"` is possible at this time)
- `button.otherarg`: A string containing extra arguments you do *not* want the user to change. For example if a button was tied to the `sum` function, but you want to always remove the `NA`'s without the user interference. Then `button.otherarg` should be equal to `",na.rm=TRUE"`. This means that for this button this part of the arguments will always be added. (Since these are extra arguments being added, note that a comma should always be used in the beginning. Of course you are also not limited to adding only 1 extra arguments, you can add as many as necessary. (`",arg1=1 ,arg2=10"`) Simply add them here as you would add them inside the function itself.)
- `arg.frames`: A vector containing the names of those frames from which this button function should pull its arguments.
- `save`: Logical value determining if the result of the button function should be saved. For example for a plotting function this is mosty likely not necessary, however for a diagnostic result it is. The difference between a `TRUE` and `FALSE` option is shown in figure 15.
- `show.save`: Logical value determining if the saved result should be printed afterwards as well. (See Figure 15)

- `show:` Logical value determining if the button function should be shown in R Commander. It is good practice to do this for the plotting and diagnostics functions however if is a function to create a new window, it is probably not necessary to show it. (See Section 5.5.1 for another use.)

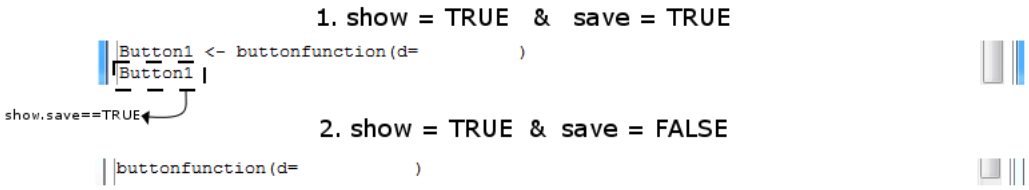

Figure 15: *Manual Buttons - save option*

5.3 Example Script - Plaid Biclustering

In this small section, we will demonstrate some parts of window creation in an example. The example which was chosen was to implement a biclustering method, namely the *Plaid* method. (The full code can be found in the Appendix)

```
plaid_WINDOW <- function(list.info=list()){

    ############################
    ## PREAMBLE/INFORMATION ##
    ############################

    dialogtitle <- "Plaid Biclustering"

    usetabs <- TRUE
    tabnames <- c("Biclustering","Plot & Diagnostics")

    if(usetabs){ntabs <- length(tabnames)} else {ntabs <- 1}
    new.frames <- .initialize.new.frames(ntabs)
    grid.config <- .initialize.grid.config(ntabs)
    grid.rows <- .initialize.grid.rows(ntabs)

    helppage <- "BCPlaid"

    ##################
    ## GRID BUTTONS ##
    ##################

    make.help.button <- TRUE
    make.setwd.button <- FALSE
    make.resetgws.button <- FALSE
    make.seed.button <- TRUE

        #... followed by tabs, frames,...
}
```

First of all the general information is filled out in the script above. This dialog contains two tabs with a help and seed grid button. The code for the second tab has been omitted in the Appendix.
Next in Figure 16, some of the frames are highlighted with their corresponding code. Note also the use of the buttons and how frames are chosen to give the arguments to the function tied to the button (red arrows).
The function used to execute the plaid algorithm is called `biclust` with the argument `method=BCPlaid()`. The latter you can see coming back in the `button.otherarg`. This means the end result would look something like

```
PlaidResult <- biclust(x=data,method=BCPlaid(),background=TRUE,
                                shuffle=3,back.fit=0,max.layers=20,...)
```

with of course the extra addition of the other parameters of other frames (the frames defined in `arg.frames`).

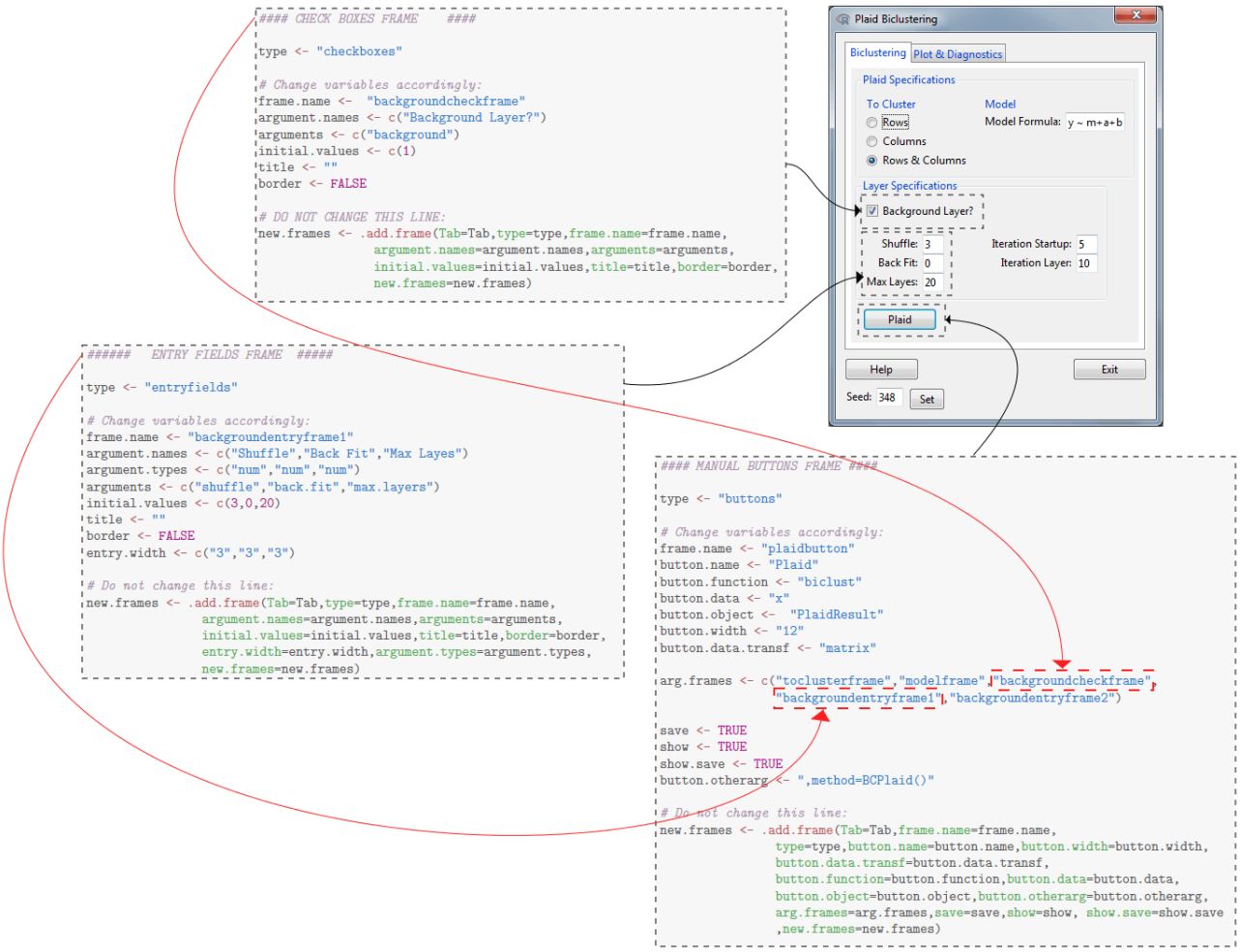

Figure 16: Building the Plaid Window

Following the rest of the frame creations is of course the grid configuring and the row combining of the first tab. In this extract of the script, one can see the frames from Figure 16 being placed in the last three of the four rows of the matrix after which the second and third row are made into a box with border and *Layer Specifications* title.

```
### 2. CONFIGURING THE GRID ###
grid.config <- .grid.matrix(Tab=Tab,c("toclusterframe","modelframe","backgroundcheckframe"
               ,NA,"backgroundentryframe1","backgroundentryframe2","plaidbutton",NA),
               byrow=TRUE,nrow=4,ncol=2,grid.config=grid.config)


### 3. COMBINING THE ROWS ###
grid.rows <- .combine.rows(Tab=Tab,rows=c(1),title="Plaid Specifications",border=TRUE,
             grid.rows=grid.rows,grid.config=grid.config)
grid.rows <- .combine.rows(Tab=Tab,rows=c(2,3),title="Layer Specifications",border=TRUE,
             grid.rows=grid.rows,grid.config=grid.config)


#### Plot & Diagnostics Tab has been omitted ###


#########################################################################
## USE ALL THE ARGUMENTS IN THE GENERAL GUI_TEMPLATE FUNCTION    ##
#########################################################################
GUI_template(dialogtitle=dialogtitle,helppage=helppage,make.resetgws.button=
       make.resetgws.button,make.setwd.button=make.setwd.button,make.help.button=
       make.help.button,make.seed.button=make.seed.button,usetabs=usetabs,tabnames=
```

```
        tabnames,grid.config=grid.config,grid.rows=grid.rows,new.frames=new.frames)
```

Finally, the `GUI_template` function is used to combine all the gathered information and create the actual window.

5.4 Testing your windows

To test your created window function, first load in the `REST` package which will launch the basic R Commander interface. Now open up an empty script and paste your window script there. Going back to the example in the previous section, this would simply mean pasting the entire `plaid_WINDOW` function here. Now run this function in R and your new window will appear on top of the R Commander Interface.
If necessary, load in some data first before you start testing out the buttons and checking if the right functions are appearing in the R Commander window.

5.5 Advanced Techniques

5.5.1 Using `doItAndPrint()` and `justDoIt()`

We have seen before that for *manual buttons*, there is a variable called `show` which will prevent the function from being printed in R Commander. This is helpfull if this function is simply to open up a new window. There is however another way you can use this option (`show==FALSE`) in combination with `save==FALSE`.
In this case, nothing is shown and nothing is saved when you click the button. The function tied to the button is simply being executed. What will be shown now is how to let several functions appear in the R-Commander screen instead of only one.
This is done through the help of the following two `Rcmdr` functions. In these functions the `command` argument is simply an R-expression in character format. (e.g. `"a <- 10+9"` or `"b <- mean(c(1,2,3,4))"` or `"a+b"`)

- `doItAndPrint(command)`: This function will print the command to the script window and execute.
- `justDoIt(command)`: This fucntion will only execute the command in the output window, but not print it in the script window.

So for example what you could do is make a function containing these `doItAndPrint` and/or `justDoIt`, and then link it to a button. Like this you can send multiple commands (containing functions, expressions,...) to the R Commander windows by clicking only a single button in your GUI.

Tips:

1. Use `paste()` or `paste0()` for the creation of your commands. By using these functions you can let your commands be created by the arguments of the function linked to the button.
2. If you need to use characters in a command, use `' '` or `\" \"`.
   (e.g. `command <- paste0("names <- c('one','two')")`
   or `command <- paste0("names <- c(\"one\",\"two\")")`)
3. `ActiveDataSet()` will give back the name of the current active dataset in R Commander.

Example:
In this example `CenterColumns` is linked to a button and will compute either the medians or the means of the columns of the active dataset (which is a data frame) and then plot these. Note that the function has an argument which it will receive to either use the median or mean.

```
CenterColumns <- function(type=c("mean","median")){
        command <- paste0("center.vector <- apply(",ActiveDataSet(),
                                    ",MARGIN=2,FUN=",type,")")
        doItAndPrint(command)
        doItAndPrint("center.vector")

        doItAndPrint(paste0("plot(center.vector,main='Column Centers',xlab='Columns',
                            ylab='Value')"))
}
```

Clicking this button with mean as an argument would then result in Figure 17, accompanied by the plot in a graphics device.

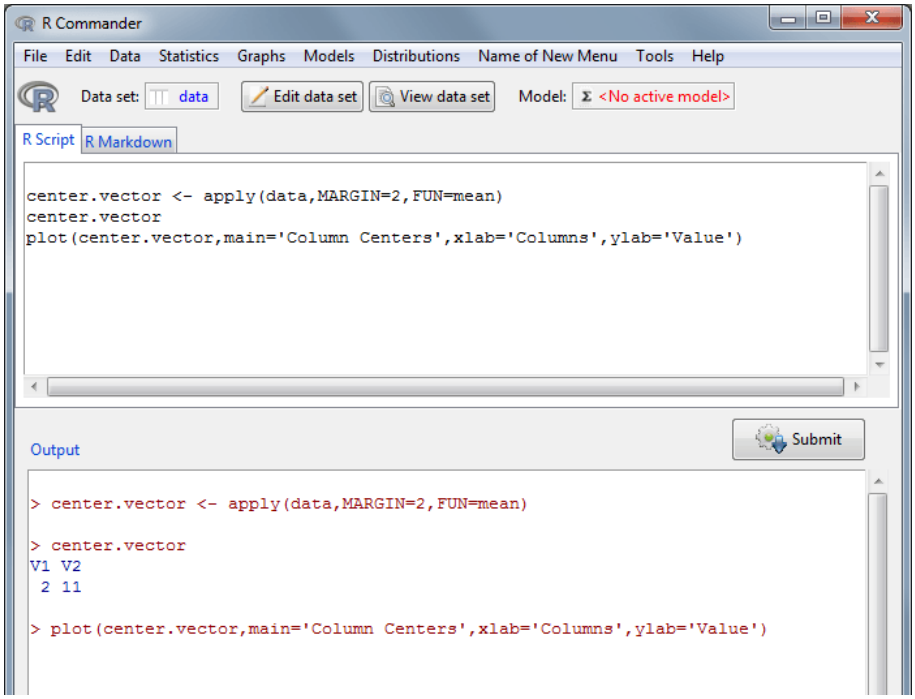

Figure 17: Example - CenterColumns

5.5.2 Window Environments

When a window is being created, all the variables (such as `new.frames`) are being saved in a certain environment. You could see this as a box in which all the parameters and variables connected with this window reside. By using the `GetWindowsENVIR()` function, you will get back a `list` object of which each element corresponds to a window which has been used. The name of such a list element is the `dialogtitle` and the element itself is the created environment. Also remember that each time you reopen a window, the environment will change to a new one and if you call the `list` object again, this change will be visible here too.
Now you might wonder why knowing this environment could be useful. By having access to these variables of another window, you can change these at any time, giving your GUI even more flexibility.
An easy example could be that you have a button called *'Presets'* on your main window. Pressing this button would then open up a new window in which the user can select with a radio button which one of the presets it would like (Preset 1, Preset 2,...). In this new window the user could then click on an *OK* button. The function tied to this button would then, with the above described process, change all the inputs in the main window (to this preset) before closing down the second window.
While this can be done manually, to facilitate this process, some functions were created for the most used application, namely changing what the user can enter in the GUI. Also an example will be given for clarification.

Functions

`ChangeWindow(dialogtitle, tab=1, framename, argument, new.value)`

Description:
This function will change the value the user can choose/enter in the frames. The behaviour is a bit different for the types of frames. See `new.value` for more details.

- `dialogtitle:` The title of window which is going to be altered. This should be the same as `dialogtitle` in your window script.
- `tab:` The number of the tab in which you want something changed (if no tabs are used, choose number 1).
- `framename:` The name of the frame in which something is going to be changed (same name as in the window script).
- `argument:` The name of the argument in which something should be changed (same as in the window script).

- `new.value`: The new value which should be entered for this argument in this particular frame. `new.value` should almost always be a character (string), unless you are using it to change something in a list box.
  - ▷ *Entry Fields:* `new.value` should be a character/string (can be anything) which will then be pasted in entry field of the chosen argument. (e.g. `"5"`, `"c(1,2,3)"` or `"c('one','two')"`)
  - ▷ *Check Boxes:* `new.value` should be either `"0"` or `"1"` which corresponds with unchecked and checked.
  - ▷ *Radio Buttons:* `new.values` should be that one of the `argument.values` you wish to be selected
  - ▷ *Value Sliders:* `new.values` should be a character containing the numerical value you want the slider to be on (e.g. `"10"`). Note that this value will be rounded to the closest possible position on the slider.
  - ▷ *Spin Boxes:* `new.values` should be a character containing the numerical value you want to spinbox to have (e.g. `"5"`). Note that this value will be rounded to one of the possible values the spin box can take.
  - ▷ *List Box:* Instead of simply changing which items are selected in the box, `new.values` will instead change the items which should be available in the box. In this case, `new.values` should be a data frame consisting out of 2 columns. The first column should contain the new `argument.names` and the second the new `argument.values`.

```
CancelWindow(dialogtitle)
```

Description:
This function will close down the chosen window.

- `dialogtitle`: The title of the window which should be destroyed. This should be the same `dialogtitle` as in the window script.

Example

In this example (Figure 18) we have a main window called *'Example'* which has initially an empty entry field after 'Values?'. Clicking the *Choose* button opens up a new window called *'Choosing'*. In this new window the user can select one or multiple values in a list box and after pressing *OK*, they will appear in vector format in the earlier mentioned empty entry field before the new *Choosing* window closes down.
The code for the relevant frames is given in Figure 18 together with the function tied to the *OK* button. Note that to save some space, the `.add.frame` line was omitted in these frame scripts.

First, you can already see that the function tied to the *Choose* button is simply the `choose_WINDOW` function without any arguments (`arg.frames <- c()`). The function tied to the *OK* button, `setentry_example`, has 1 argument (`values`) which comes from `"listboxframe1"`. This is again defined by the `arg.frames` for the *OK* button.
The first thing `setentry_example` does is converting `values` to a character format in `new.value`. For example in this example `c("value1","value3")` becomes `"c('value1','value3')"`. Next, this `new.value` is entered in the entry field by using the `ChangeWindow` function on the `"Example"` window, first tab, `"entryframe1"` frame and `"arg1"` argument.
Lastly the *'Choosing'* window is closed down by using the `CancelWindow` function.

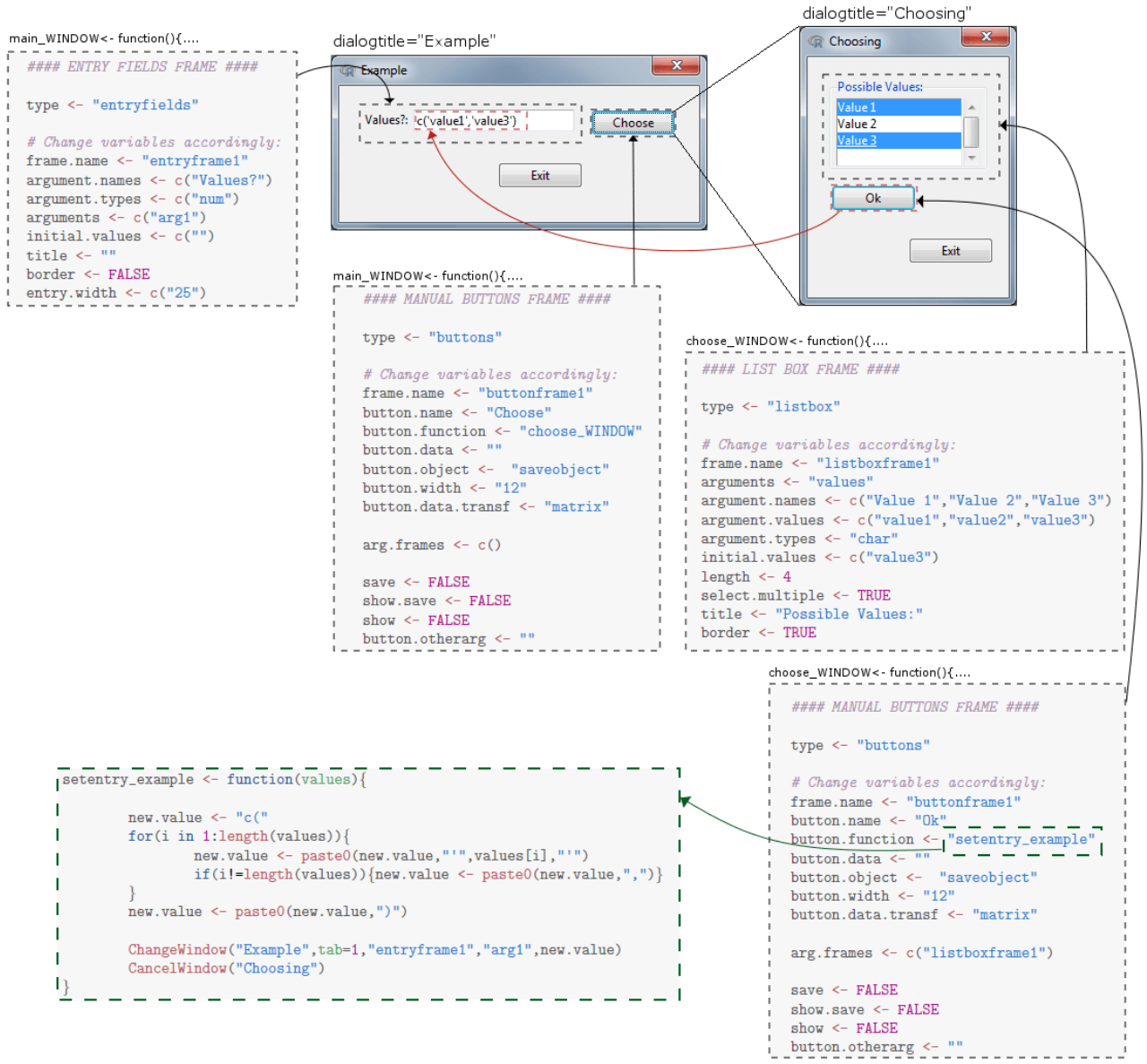

Figure 18: Window Environments - Example (Code + Window)

5.6 Extra Functions

In this section, you will be able to find some extra functions meant to be linked to a button or to be used inside functions described in Section 5.5.1.

5.6.1 Save Function

```
SaveGUI(object.names, init.name="result")
```

This function will open up a *Save* window (Figure 19), saving the chosen `object.names` (vector of the names of the objects to be saved) in an `.RData` file. The `init.name` variable simply decide the standard save name which should appear in the save window.

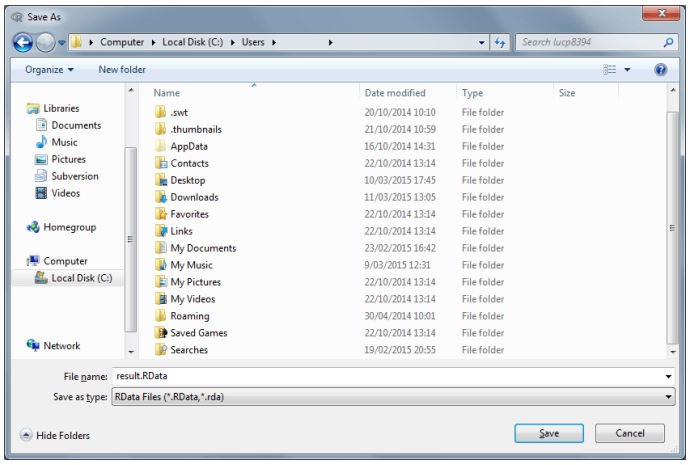

Figure 19: Save Window

5.6.2 Load Function

```
LoadGUI()
```

This function opens up a *Load* window (Figure 20) in which saved `.RData` objects can be loaded.

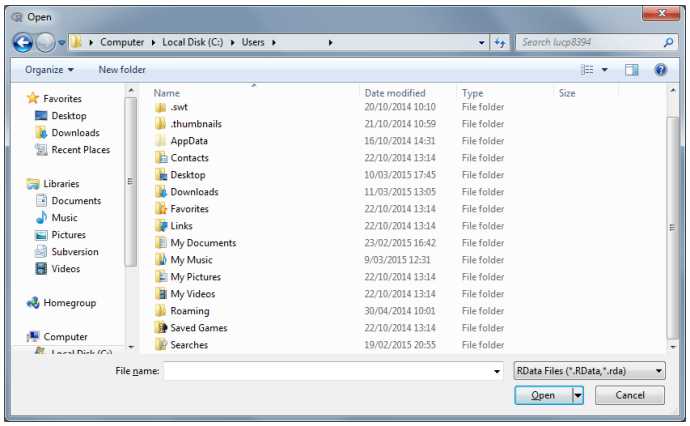

Figure 20: Load Window

References

Dalgaard, P. (2001a), "A Primer on the R-Tcl/Tk Package," *R-News*, 1, 27–31.

— (2001b), "The R-Tcl/Tk interface," in *DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, Vienna, Austria.

— (2002), "Changes to the R-Tcl/Tk package," *R-News*, 2, 25–27.

Fox, J. (2005), "The R Commander: A basic-statistics graphical user interface to R," *Journal of Statistical*, 14, 1–42.

— (2007), "Extending the R Commander by "Plug-in" Packages," *R-News*, 7, 46–52.

Fox, J. and Bouchet-Valat, M. (2013), *Getting Started With the R Commander*.

Lawrence, M. and Lang, D. T. (2014), *Package 'RGtk2' : R bindings for Gtk 2.8.0 and above*, package Version 2.20.31.

R Core Team (2014), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria.

Verzani, J. (2014), *Package 'gWidgets': gWidgets API for building toolkit-independent, interactive GUIs*, package Version 0.0-54.

6 Appendix

All the code in the appendix can also be found in R scripts, located in the `doc` folder of the package.

6.1 `.onAttach`-function

```
.onAttach <- function(libname, pkgname){
    if (!interactive()) return()
    putRcmdr("slider.env", new.env())
    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()
        }
    }
}
```

6.2 General Script

```
GUI_WINDOW <- function(list.info=list()){

    ############################
    ## PREAMBLE/INFORMATION ##
    ############################

    dialogtitle <- "This is the title of the window"

    usetabs <- TRUE
    tabnames <- c("Tab 1","Tab 2","Tab 3")
    helppage <- "plot"

    # Do not change these lines
    if(usetabs){ntabs <- length(tabnames)} else {ntabs <- 1}
    new.frames <- .initialize.new.frames(ntabs)
    grid.config <- .initialize.grid.config(ntabs)
    grid.rows <- .initialize.grid.rows(ntabs)
    ###

    ###################
    ## GRID BUTTONS ##
    ###################

    make.help.button <- TRUE
    make.setwd.button <- TRUE
    make.resetgws.button <- TRUE
    make.seed.button <- TRUE

    ############
    ## TAB 1 ##
    ############
    Tab <- 1
```

```
    ### 1. ADDING THE FRAMES ###

    # Add frames here


    ### 2. CONFIGURING THE GRID ###
    grid.config <- .grid.matrix(Tab = Tab, c("frame1","frame2","frame3",NA),
      byrow=TRUE, nrow=2, ncol=2, grid.config=grid.config)


    ### 3. COMBINING THE ROWS ###
    grid.rows <- .combine.rows(Tab = Tab, rows = c(1,2),title = "A nice box: ",
      border = TRUE, grid.rows=grid.rows, grid.config = grid.config)

    ##############
    ### TAB 2 ###
    ##############
    Tab <- 2

    # Repeat what you did for tab 1 for as many tabs as you like...


    ######################################################################
    ## USE ALL THE ARGUMENTS IN THE GENERAL GUI_TEMPLATE FUNCTION   ##
    ######################################################################
    GUI_template(dialogtitle = dialogtitle, helppage = helppage, make.resetgws.button =
        make.resetgws.button, make.setwd.button = make.setwd.button,
        make.help.button = make.help.button, make.seed.button = make.seed.button,
        usetabs = usetabs, tabnames = tabnames, grid.config = grid.config, grid.rows =
        grid.rows, new.frames = new.frames)

}
```

6.3 Frame Scripts

```
#### ENTRY FIELDS FRAME ####

type <- "entryfields"

# Change variables accordingly:
frame.name <- "entryframe1"
argument.names <- c("Argument 1","Argument 2","Argument 3")
argument.types <- c("num","num","char")
arguments <- c("arg1","arg2","arg3")
initial.values <- c(1,2,"a")
title <- "A Title"
border <- FALSE
entry.width <- c("2","2","6")

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,type=type
        ,frame.name=frame.name,argument.names=argument.names
        ,arguments=arguments,initial.values=initial.values
        ,title=title,border=border,entry.width=entry.width
        ,argument.types=argument.types  ,new.frames=new.frames)




#### RADIO BUTTONS FRAME  ####
```

```
type <- "radiobuttons"

# Change variables accordingly:
frame.name <- "radioframe1"
argument.names <- c("Button 1","Button 2","Button 3")
arguments <- c("buttonarg")
argument.types <- "char"
argument.values <- c("b1","b2","b3")
initial.values <- "b3"
title <- "Button Options"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type
        ,frame.name=frame.name,argument.names=argument.names
        ,arguments=arguments,argument.values=argument.values
        ,initial.values=initial.values,title=title,border=border
        ,new.frames=new.frames,argument.types=argument.types)


#### CHECK BOXES FRAME  ####

type <- "checkboxes"

# Change variables accordingly:
frame.name <-  "checkboxframe1"
argument.names <- c("Check 1","Check 2","Check 3")
arguments <- c("checkarg1","checkarg2","checkarg3")
initial.values <- c(0,1,1)
title <- "title"
border <- FALSE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type
        ,frame.name=frame.name,argument.names=argument.names
        ,arguments=arguments,initial.values=initial.values
        ,title=title,border=border,new.frames=new.frames)


#### VALUE SLIDER FRAME  ####

type <- "valuesliders"

# Change variables accordingly:
frame.name <- "sliderframe1"
argument.names <- c("Slider 1  ","Slider 2  ","Slider 3  ")
arguments <- c("sliderarg1","sliderarg2","sliderarg3")
initial.values <- c(1,5,10)
from <- c(1,1,1)
to <- c(5,50,500)
by <- c(1,10,50)
length <- c(50,100,150)
title <- "Title"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
        title=title,border=border,frame.name=frame.name,
        argument.names=argument.names,arguments=arguments,
        initial.values=initial.values,from=from,to=to,by=by,
```

```
        length=length,new.frames=new.frames)


#### SPIN BOX FRAME  ####

type <- "spinboxes"

# Change variables accordingly:
frame.name <- "spinboxframe1"
argument.names <- c("Spin Box 1: ","Spin Box 2: ","Spin Box 3: ")
arguments <- c("spinarg1","spingarg2","spingarg3")
initial.values <- c(5,10,20)
from <- c(1,5,10)
to <- c(10,20,30)
by <- c(1,1,1)
entry.width <- "2"
title <- "Spin Box !"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
        frame.name=frame.name,argument.names=argument.names,
        arguments=arguments,initial.values=initial.values,
        from=from,to=to,by=by,entry.width=entry.width,
        title=title,border=border,new.frames=new.frames)

#### LIST BOX FRAME ####

type <- "listbox"

# Change variables accordingly:
frame.name <- "listboxframe1"
arguments <- "listboxarg"
argument.names <- c("Value 1","Value 2","Value 3")
argument.values <- c("value1","value2","value3")
argument.types <- "char"
initial.values <- c("value3")
length <- 4
select.multiple <- FALSE
title <- "A list box:"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
    frame.name=frame.name,argument.names=argument.names,
    arguments=arguments,argument.values=argument.values,
    argument.types=argument.types, initial.values=initial.values,
    length=length,select.multiple=select.multiple,
    title=title,border=border,new.frames=new.frames)


#### MANUAL BUTTONS FRAME ####

type <- "buttons"

# Change variables accordingly:
frame.name <- "buttonframe1"
button.name <- "Button 1"
button.function <- "buttonfunction"
button.data <- "d"
button.object <-  "saveobject"
```

```
button.width <- "12"
button.data.transf <- "matrix" # only matrix available here !

arg.frames <- c("frame1","frame2")

save <- TRUE
show.save <- TRUE
show <- TRUE
button.otherarg <- ""  # always start with a ,

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,frame.name=frame.name,
        type=type,button.name=button.name,button.width=button.width,
        button.data.transf=button.data.transf,
        button.function=button.function,button.data=button.data,
        button.object=button.object,button.otherarg=button.otherarg,
        arg.frames=arg.frames,save=save,show=show,show.save=show.save,
        new.frames=new.frames)
```

6.4 Example Script

```
plaid_WINDOW <- function(list.info=list()){

##########################
## PREAMBLE/INFORMATION ##
##########################

dialogtitle <- "Plaid Biclustering"


usetabs <- TRUE

tabnames <- c("Biclustering","Plot & Diagnostics")

if(usetabs){ntabs <- length(tabnames)} else {ntabs <- 1}
new.frames <- .initialize.new.frames(ntabs)
grid.config <- .initialize.grid.config(ntabs)
grid.rows <- .initialize.grid.rows(ntabs)


helppage <- "BCPlaid"


##################
## GRID BUTTONS ##
##################

make.help.button <- TRUE
make.setwd.button <- FALSE
make.resetgws.button <- FALSE
make.seed.button <- TRUE

############
## TAB 1 ##
############
Tab <- 1

### 1. ADDING THE FRAMES ###

#### RADIO BUTTONS FRAME   ####
```

```
#                                #

type <- "radiobuttons"

# Change variables accordingly:
frame.name <- "toclusterframe"
argument.names <- c("Rows","Columns","Rows & Columns")
arguments <- c("cluster")
argument.values <- c("r","c","b")
argument.types <- "char"
initial.values <- "b"
title <- "To Cluster"
border <- FALSE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,frame.name=frame.name,
      argument.names=argument.names,arguments=arguments,argument.values=
      argument.values,initial.values=initial.values,title=title,border=border,
      new.frames=new.frames,argument.types=argument.types)


######   ENTRY FIELDS FRAME  #####
#        #

type <- "entryfields"

# Change variables accordingly:
frame.name <- "modelframe"
argument.names <- c("Model Formula")
argument.types <- c("num")
arguments <- c("fit.model")
initial.values <- c("y ~ m+a+b")
title <- "Model"
border <- FALSE
entry.width <- c("10")

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,type=type,frame.name=frame.name,argument.names=
      argument.names,arguments=arguments,initial.values=initial.values,title=title,
      border=border,entry.width=entry.width,argument.types=argument.types
      ,new.frames=new.frames)

#### CHECK BOXES FRAME    ####
#                                 #

type <- "checkboxes"

# Change variables accordingly:
frame.name <-  "backgroundcheckframe"
argument.names <- c("Background Layer?")
arguments <- c("background")
initial.values <- c(1)
title <- ""
border <- FALSE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,frame.name=frame.name,argument.names=
      argument.names,arguments=arguments,initial.values=initial.values,title=title,
      border=border,new.frames=new.frames)
```

```
######   ENTRY FIELDS FRAME  #####
#        #

type <- "entryfields"

# Change variables accordingly:
frame.name <- "backgroundentryframe1"
argument.names <- c("Shuffle","Back Fit","Max Layes")
argument.types <- c("num","num","num")
arguments <- c("shuffle","back.fit","max.layers")
initial.values <- c(3,0,20)
title <- ""
border <- FALSE
entry.width <- c("3","3","3")

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,type=type,frame.name=frame.name,argument.names=
      argument.names,arguments=arguments,initial.values=initial.values,title=title,
      border=border,entry.width=entry.width,argument.types=argument.types
      ,new.frames=new.frames)


######   ENTRY FIELDS FRAME  #####
#        #

type <- "entryfields"

# Change variables accordingly:
frame.name <- "backgroundentryframe2"
argument.names <- c("Iteration Startup","Iteration Layer")
argument.types <- c("num","num")
arguments <- c("iter.startup","iter.layer")
initial.values <- c(5,10)
title <- ""
border <- FALSE
entry.width <- c("3","3")

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,type=type,frame.name=frame.name,
      argument.names=argument.names,arguments=arguments,initial.values=
      initial.values,title=title,border=border,entry.width=entry.width,
      argument.types=argument.types  ,new.frames=new.frames)

#### MANUAL BUTTONS FRAME ####

type <- "buttons"

# Change variables accordingly:
frame.name <- "plaidbutton"
button.name <- "Plaid"
button.function <- "biclust"
button.data <- "x"
button.object <-  "PlaidResult"
button.width <- "12"
button.data.transf <- "matrix" # only matrix available here !

arg.frames <- c("toclusterframe","modelframe","backgroundcheckframe",
      "backgroundentryframe1","backgroundentryframe2")

save <- TRUE
show <- TRUE
```

```
show.save <- TRUE
button.otherarg <- ",method=BCPlaid()"

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,frame.name=frame.name,
type=type,button.name=button.name,button.width=button.width,
button.data.transf=button.data.transf,
button.function=button.function,button.data=button.data,
button.object=button.object,button.otherarg=button.otherarg,
arg.frames=arg.frames,save=save,show=show,new.frames=new.frames,
            show.save=show.save)

### 2. CONFIGURING THE GRID ###
grid.config <- .grid.matrix(Tab=Tab,c("toclusterframe","modelframe",
       "backgroundcheckframe",NA,"backgroundentryframe1","backgroundentryframe2",
       "plaidbutton",NA),byrow=TRUE,nrow=4,ncol=2,grid.config=grid.config)


### 3. COMBINING THE ROWS ###
grid.rows <- .combine.rows(Tab=Tab,rows=c(1),title="Plaid Specifications",
      border=TRUE,grid.rows=grid.rows,grid.config=grid.config)
grid.rows <- .combine.rows(Tab=Tab,rows=c(2,3),title="Layer Specifications",
      border=TRUE,grid.rows=grid.rows,grid.config=grid.config)

#############
### TAB 2 ###
#############
Tab <- 2

# Repeat what you did for tab 1 for as many tabs as you like...


#########################################################################
## USE ALL THE ARGUMENTS IN THE GENERAL GUI_TEMPLATE FUNCTION    ##
#########################################################################
GUI_template(dialogtitle=dialogtitle,helppage=helppage,make.resetgws.button=
      make.resetgws.button,make.setwd.button=make.setwd.button,make.help.button=
      make.help.button,make.seed.button=make.seed.button,usetabs=usetabs,tabnames=
      tabnames,grid.config=grid.config,grid.rows=grid.rows,new.frames=new.frames)

}
```

6.5 Window Environment Example

```
main_WINDOW <- function(list.info=list()){

############################
## PREAMBLE/INFORMATION ##
############################

dialogtitle <- "Example"

usetabs <- FALSE

tabnames <- c("Biclustering","Plot & Diagnostics")

if(usetabs){ntabs <- length(tabnames)} else {ntabs <- 1}
new.frames <- .initialize.new.frames(ntabs)
grid.config <- .initialize.grid.config(ntabs)
grid.rows <- .initialize.grid.rows(ntabs)
```

```
helppage <- ""

##################
## GRID BUTTONS ##
##################

make.help.button <- FALSE
make.setwd.button <- FALSE
make.resetgws.button <- FALSE
make.seed.button <- FALSE

############
## TAB 1 ##
############
Tab <- 1

### 1. ADDING THE FRAMES ###

#### ENTRY FIELDS FRAME ####

type <- "entryfields"

# Change variables accordingly:
frame.name <- "entryframe1"
argument.names <- c("Values?")
argument.types <- c("num")
arguments <- c("arg1")
initial.values <- c("")
title <- ""
border <- FALSE
entry.width <- c("25")

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,type=type
,frame.name=frame.name,argument.names=argument.names
,arguments=arguments,initial.values=initial.values
,title=title,border=border,entry.width=entry.width
,argument.types=argument.types  ,new.frames=new.frames)


#### MANUAL BUTTONS FRAME ####

type <- "buttons"

# Change variables accordingly:
frame.name <- "buttonframe1"
button.name <- "Choose"
button.function <- "choose_WINDOW"
button.data <- ""
button.object <-  "saveobject"
button.width <- "12"
button.data.transf <- "matrix"

arg.frames <- c()

save <- FALSE
show.save <- FALSE
show <- FALSE
button.otherarg <- ""  # always start with a ,

# Do not change this line:
```

```
new.frames <- .add.frame(Tab=Tab,frame.name=frame.name,
type=type,button.name=button.name,button.width=button.width,
button.data.transf=button.data.transf,
button.function=button.function,button.data=button.data,
button.object=button.object,button.otherarg=button.otherarg,
arg.frames=arg.frames,save=save,show=show,show.save=show.save,
new.frames=new.frames)

### 2. CONFIGURING THE GRID ###
grid.config <- .grid.matrix(Tab=Tab,c("entryframe1","buttonframe1"),
    byrow=TRUE,nrow=1,ncol=2,grid.config=grid.config)

### 3. COMBINING THE ROWS ###

#####################################################################
## USE ALL THE ARGUMENTS IN THE GENERAL GUI_TEMPLATE FUNCTION    ##
#####################################################################
GUI_template(dialogtitle=dialogtitle,helppage=helppage,make.resetgws.button=
    make.resetgws.button,make.setwd.button=make.setwd.button,make.help.button=
    make.help.button,make.seed.button=make.seed.button,usetabs=usetabs,tabnames=
    tabnames,grid.config=grid.config,grid.rows=grid.rows,new.frames=new.frames)

}

choose_WINDOW <- function(){

############################
## PREAMBLE/INFORMATION ##
############################

dialogtitle <- "Choosing"

usetabs <- FALSE

tabnames <- c("Biclustering","Plot & Diagnostics")

if(usetabs){ntabs <- length(tabnames)} else {ntabs <- 1}
new.frames <- .initialize.new.frames(ntabs)
grid.config <- .initialize.grid.config(ntabs)
grid.rows <- .initialize.grid.rows(ntabs)

helppage <- ""

###################
## GRID BUTTONS ##
###################

make.help.button <- FALSE
make.setwd.button <- FALSE
make.resetgws.button <- FALSE
make.seed.button <- FALSE

############
## TAB 1 ##
############
Tab <- 1

### 1. ADDING THE FRAMES ###

#### LIST BOX FRAME ####
```

```
type <- "listbox"

# Change variables accordingly:
frame.name <- "listboxframe1"
arguments <- "values"   # should only be 1
argument.names <- c("Value 1","Value 2","Value 3")
argument.values <- c("value1","value2","value3")
argument.types <- "char"      # should be only 1
initial.values <- c("value3")     # Can be 1 or multiple
length <- 4  #no character , if not given, will take length of names
select.multiple <- TRUE
title <- "Possible Values:"
border <- TRUE

# DO NOT CHANGE THIS LINE:
new.frames <- .add.frame(Tab=Tab,type=type,
frame.name=frame.name,argument.names=argument.names,
arguments=arguments,argument.values=argument.values,
argument.types=argument.types, initial.values=initial.values,
length=length,select.multiple=select.multiple,
title=title,border=border,new.frames=new.frames)

#### MANUAL BUTTONS FRAME ####

type <- "buttons"

# Change variables accordingly:
frame.name <- "buttonframe1"
button.name <- "Ok"
button.function <- "setentry_example"
button.data <- ""
button.object <-  "saveobject"
button.width <- "12"
button.data.transf <- "matrix"

arg.frames <- c("listboxframe1")

save <- FALSE
show.save <- FALSE
show <- FALSE
button.otherarg <- ""  # always start with a ,

# Do not change this line:
new.frames <- .add.frame(Tab=Tab,frame.name=frame.name,
type=type,button.name=button.name,button.width=button.width,
button.data.transf=button.data.transf,
button.function=button.function,button.data=button.data,
button.object=button.object,button.otherarg=button.otherarg,
arg.frames=arg.frames,save=save,show=show,show.save=show.save,
new.frames=new.frames)


### 2. CONFIGURING THE GRID ###
grid.config <- .grid.matrix(Tab=Tab,c("listboxframe1","buttonframe1"),
    byrow=TRUE,nrow=2,ncol=1,grid.config=grid.config)


### 3. COMBINING THE ROWS ###

#####################################################################
## USE ALL THE ARGUMENTS IN THE GENERAL GUI_TEMPLATE FUNCTION   ##
```

```
#####################################################################
GUI_template(dialogtitle=dialogtitle,helppage=helppage,make.resetgws.button=
    make.resetgws.button,make.setwd.button=make.setwd.button,make.help.button=
    make.help.button,make.seed.button=make.seed.button,usetabs=usetabs,tabnames=
    tabnames,grid.config=grid.config,grid.rows=grid.rows,new.frames=new.frames)

}

setentry_example <- function(values){

new.value <- "c("
for(i in 1:length(values)){
new.value <- paste0(new.value,"'",values[i],"'")
if(i!=length(values)){new.value <- paste0(new.value,",")}
}
new.value <- paste0(new.value,")")

ChangeWindow("Example",tab=1,"entryframe1","arg1",new.value)
CancelWindow("Choosing")
}
```