The R-GGobi Interface

Duncan Temple Lang

May 6, 2004

Abstract

We describe some of the facilities provided by the *R–ggobi*interface. We look at the functions relative to the elements of the GGobi hierarchy of figure 1.

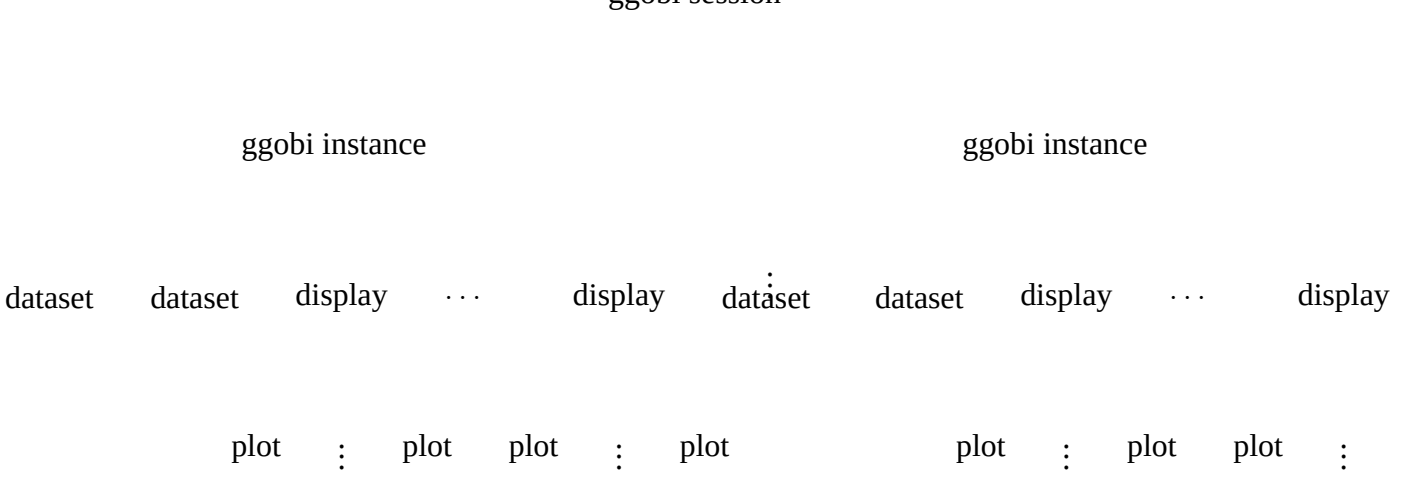

Figure 1: *ggobi* Session Hierarchy. Within a session, there can be multiple independent ggobi instances. Each instance has one or more datasets. A Display contains one or more plots and these plots all relates variables within the same dataset.

1 The Session

1.1 Creating a *ggobi* instance

The function *ggobi()* allows one to create a ggobi instance. One can specify a file name or an *R* data frame to populate the ggobi with one or more datasets, or alternatively create an empty GGobi

and add data later in the session. *ggobi()* creates a GGobi control panel window and an initial scatterplot display if there is any data.

We can use data from a file **flea.xml**

```
g <- ggobi(system.file("data", "flea.xml", pkg="Rggobi")))
```

Alternatively, we can use a data frame.

```
data(mtcars)
g <- ggobi(mtcars)
```

In addition to specifying the source of the data, one can provide other arguments to parameterize how GGobi is created. Specifically one can give "command line arguments" to GGobi as if invoking it as a stand-alone program. For example, we can specify the format of the data and the file in which it is located as

```
g <- ggobi(args=c("-xml", system.file("data", "flea.xml", pkg="Rggobi"))
```

This is equivalent to invoking GGobi on the command line as

```
ggobi -xml <wherever>/data/flea.xml
```

1.1.1 *ggobi* objects

When one calls *ggobi()*, the return value is an S object of class *ggobi*. Generally one never looks at the contents of this object. It is an opaque data type that is used merely to identify the GGobi instance. When there two or more GGobi instances in existence, we can use a *ggobi* object to identify which instance to operate on. Most functions support the *.gobi* argument for this purpose.

When one creates a *ggobi* instance, it becomes the default instance and all functions that operate on a GGobi instance will default to that particular one. This means that one doesn't have to specify the *.gobi* argument for most functions. Indeed, one need not assign the value returned from a call to *ggobi()*. And if one needs to retrieve it, the functions *getGGobi()* and *getDefaultGGobi()* will accomodate.

While one rarely needs to know what the contents of a *ggobi* object contain, here is a short description.

id an integer that indicates the *current* index of this ggobi instance in the list of all existing ggobis. Note that this is not necessarily valid for future use as removing other ggobis from the list will change the position of this one.

ref an *opaque* numeric value that contains the *C*-level address of the ggobi instance. This allows us both to identify the ggobi structure in subsequent calls and also verify that it still exists and is valid.

1.2 Query the session's *ggobi* Instances.

One can find out about the number and nature of the currently open or existing ggobi instances within a session. The function *getNumGGobis()* returns the number of ggobi instances currently active.

```
> getNumGGobis()
[1] 1
```

One can retrieve S objects that refer to the different internal ggobi instances using the function *getGGobi()*. This takes 0 or more integer values identifying the ggobi instances within the internal list. This returns an object of class *ggobi* for each valid index found. By default, it returns a list containing them all. If there is only one GGobi instance, only that is returned.

```
> getGGobi()
$id
[1] 1

$ref
[1] 279051040

attr(,"class")
[1] "ggobi"
```

Users can can look at the main window of a GGobi instance to find out what variables it contains. They can also get this information programmatically. The function *getDescription.ggobi()* returns a brief description of a GGobi instance.

```
> getDescription.ggobi()
$Filename
[1] "../ggobi/data/flea.xml"

$"Data mode"
XML
  3

$"Data dimensions"
[1] 74  7
```

This gives the name of the data source (a file in this case), the format in which the data was read and the number of records and variables in the dataset.

The function *describe.ggobi()* when called with no arguments returns a description of each of the ggobis. The description of each instance gives information about its state, specifically its datasets and displays, and the plots within the displays.

is a list containing

datasets a description of each dataset currently in the ggobi instance, consisting of

name both the name of the dataset and the file name from which it was read. The first name allows multiple datasets within a single XML file.

dims the dimension of the dataset, both number of records and variables.

format the data/file format from which the data were read: XML, ASCII, binary, MySQL, etc.

variables the names of the variables

auxillaryFiles the names of auxillary files that were read in the process of reading the top-level data file. This allows the user to understand more precisely how the data came to be the way they are and how to recreate it, distribute it, etc.

displays a list describing each of the displays in the ggobi instance. Each of these displays is an object of class *ggobiDisplayDescription* and contains the fields

Name the title on the window.

Type the name of the plot type (scatterplot, ash, etc.)

Plots a list containing descriptions of the display's plots:

variables a named integer vector, identifying the variables in this plot by index in their dataset and by name;

dataset the dataset associated with this plot and in which the variables reside;

ggobi the ggobi instance associated with this plot.

The last two of these are duplicates but are part of a general mechanism for describing a plot separately of the containing ggobi instance.

1.3 The Active *ggobi* Instance

Within the R session, there is a concept of an active or default ggobi instance. While all the functions that operate on a ggobi instance allow one to specify to which instance the operation should be applied, if this argument is omitted, the default instance is assumed. One can get a reference to the default ggobi instance using the function *getDefaultGGobi()*.

Similarly, one can set the default ggobi instance via the function *setDefaultGGobi()*.

1.4 Querying a *ggobi* instance

As mentioned above, the *describe.ggobi()* function can be used to create a description of a ggobi instance.

The file names associated with the datasets in a *ggobi* instance can be obtained with a call to the *getFileNames.ggobi()* function. This returns a list with an element for each dataset. By default, each element of the list contains just the name of the primary file from which the data was read. However, if the *auxillary* argument is passed as **T**, then the the names of the auxillary files referenced in the primary one and read during the initialization of the dataset are also returned.

```
> ggobi("../ggobi/data/flea.xml")
> getFileNames.ggobi(T)
[[1]]
[1] "../ggobi/data/flea.xml"        "../ggobi/data/stdColorMap.xml"
```

(Note the current version of **flea.xml** does not reference **stdColorMap.xml**.)

isValid.ggobi() determines whether the an object of class *ggobi* is valid or not. Note that this is useful for those programming the ggobi interaction, e.g. setting views, adding data, etc. Since the user is able to close the *ggobi* instance via a menu action (rather than programmatically), it is often prudent to test whether the ggobi instance still exists when one is addressing it.

1.5 Destroying a *ggobi* instance

To programmatically discard a particular ggobi instance one should use the *close.ggobi()* function. This takes care of releasing the windows and data associated with that instance, and also resets the default *ggobi* instance – *getDefaultGGobi()*.

1.6 Event Handling

Ocassionally it is necessary to force R to handle events in ggobi to ensure its appearance is updated. The function *gdk.flush()* can be called liberally to effect this.

It is not (currently) possible to suspend a single *ggobi* instance. One can supsend them all by removing the R-level event handler.

1.7 Session Constants

There are certain data values that can be considered shared across all ggobis, and others that are also constant. The latter are settings or values that are used internally. They are accessible to the S programmer/user so as to allow them to use names corresponding to internal constants. For example, to specify the glyph for a point, the programmer might use a name, e.g. `"plus"`. However, this is passed to *ggobi* in the internal form that *ggobi* recognizes, as 1. This mapping is made visible and can be done in the S language by (partial) matching the user's human-readable and descriptive name against the list of possible values and returning the internal code that matches.

There are several of these named internal code vectors that can be retrieved from the *ggobi* engine. They are:

getDataModes.ggobi() the list of data format names and their internal codes (an enumerated list of integers);

```
   ASCII   binary R/S data      XML    MySQL  Unknown
       0        1        2        3        4        5
```

getViewTypes.ggobi() the names of the different plot types (and their internal codes);

```
          Scatterplot   Scatterplot Matrix Parallel Coordinates
                    0                    1                    2
```

getGlyphTypes.ggobi() The names of the glyph types and their internal codes[1]

```
plus    x   or   fr   oc   fc    .
   1    2    3    4    5    6    7
```

getModeNames.ggobi `getModeNames.ggobi()`

```
 [1] "1D Plot"          "XYPlot"           "Rotation"         "1D Tou
 [5] "2D Tour"          "Correlation Tour" "Scale"           "Brush"
 [9] "Identify"         "Edit Lines"       "Move Points"     "Scatma
[13] "Parcoords"
```

getGlyphSizes.ggobi()() The admissable values for setting the glyph sizes.

```
[1] 0 1 2 3 4 5 6 7 8
```

As mentioned, these are constants and cannot be changed!

A colormap is a mapping of names or indices to color specifications. The colormap is not a session constant but is particular to each *ggobi* instance. One can obtain the colormap

getColorMap.ggobi() setColorMap.ggobi()

1.8 Functions

getNumGGobis() getGGobi() ggobi()

setDefaultGGobi() getDefaultGGobi()

getViewTypes.ggobi() getModeNames.ggobi() getDataModes.ggobi() getGlyphTypes.ggobi() getGlyphSizes.ggobi()

2 The *ggobi* Instance

2.1 Creating the *ggobi* Instance

We saw how to create a *ggobi* instance via the *ggobi()* function. This can be called multiple times and has the effect of creating additional instances with the specified command line arguments and/or R data sets.

The *ggobi()* function also allows us to specify data not just from a file, but also from within R itself. We can pass a data frame as the first argument to *ggobi()* and that will be added to the resulting ggobi instance as a dataset. For example, to use the dataset *mtcars()* in the R distribution in a ggobi instance, we can issue the commands

```
data(mtcars)
ggobi(mtcars)
```

[1] these are actually 1-based in S, but converted to 0-based in *ggobi*.

Note that we can also specify a file to read on the command, *in addition* to specifying a dataset from within R. This works because *ggobi()* first creates a *ggobi* instance and *then* adds the R data set to it. Thus, the command

```
g <- ggobi(mtcars, args=c("-xml", system.file("data/flea.xml.gz", pkg="R
```

will result in a *ggobi* instance with 2 data sets: flea and mtcars.

We can make further use of this facility to allow us to add new data sets at any time, not just when we create the *ggobi* instance. The function *setData.ggobi()* is used to specify either a file or an *S* data set. (This calls one of the functions *setDataFile.ggobi()* or *setDataFrame.ggobi()*.)

```
ggobi(system.file("data/buckyball"))
df <- data.frame(x=rnorm(100), y=rnorm(100), z=rnorm(100))
setData.ggobi(df)
setData.ggobi(system.file("data/flea.xml"), mode="ASCII")
```

We also saw how one get obtain a description of the *ggobi* instance using the function *describe.ggobi()*.

2.2 Querying the Instance

The dataset is an important element in the interaction with ggobi. Most functions that operate on the *ggobi* instance usually also take a dataset identifier as an argument (usually *.data*). The identifier can be specified in several different forms.

index a number indicating the index of the dataset within the list of datasets. The first dataset is identified by 1. One can determine the number of datasets currently held by a *ggobi* instance with the function *getNumDatasets.ggobi()*. Note that if a dataset is removed, the others are moved and so indeces are not robust ways to identify a dataset.

name A character vector which matches the name of the dataset. The names of the datasets can be obtained from a *ggobi* instance using the function *getDatasetNames.ggobi()* and the *names()* on a *ggobi* instance (i.e. the *names.ggobi()* function). That is,

```
g <- ggobi()
getDatasetNames.ggobi(g) # or
names(g)
```

This is less convenient than using an index, but is robust to adding and removing datasets as the names won't change. In the unusual case that two datasets have the same name, this cannot be used to reference the one later in the list.

ggobiDataset an object of class *ggobiDataset* contains a reference uniquely identifying a dataset, regardless of name or index. From then on, one can use this reference when referring to a dataset. One can be obtain a reference to a dataset by using the function *getDatasetReference.ggobi()* which accepts the dataset identifier as an index or name. Note that there is a convenient alternative to *getDatasetReference.ggobi()*. If one has the *ggobi* instance as a variable, say *g()*, then its subset operator will do the same thing as *getDatasetReference.ggobi()*.

```
g <- ggobi(system.file("data/buckyball"))
d <- g[1]
```

Additionally, the return value of *describe.ggobi()* contains *ggobiDataset* objects for each of the different datasets.

In addition to the convenient syntax for obtaining the dataset reference, we have defined methods for *dim(), dimnames(), nrow(), ncol()*. for *ggobiDataset* objects. Objects of this class are returned from the functions *setData(), setDataFile.ggobi()* and *setDataFrame.ggobi()*. Additionally, we have defined a method for the *[()* operator

```
ggobi()
d <- setData.ggobi("../ggobi/data/flea")
dim(d)
ncol(d)
nrow(d)
dimnames(d)
rownames(d)
colnames(d)
names(d)
```

Currently, the assignment versions of these operations are not defined.

3 Datasets

We can query and set many of the characteristics of the dataset and the associated display information that controls the appearance of the plots involving records from this dataset. These characteristics include

Variable Names *getVariableNames.ggobi()* and the names method for a *ggobiDataset* return a charcter vector off the names of the variables. For example, using the flea example data,

```
> names(g[1])
[1] "tars1" "tars2" "head"  "aede1" "aede2" "aede3"
```

Note also that *dimnames(), dim(), ncol(), nrow(), rownames()*, etc. all work on objects of class *ggobiDataset()*.

rownames *getRowNames.ggobi()* and *setRowNames.ggobi()* query and set the name or label of one or more records.

glyphs both the plotting character/image and its size can be controlled via the functions *getGlyphs.ggobi()* and *setGlyphs.ggobi()*. The available glyphs can be determined via the function *getGlyphsTypes.ggobi()*. Similarly, *getGlyphSizes.ggobi()* gives the permissible values for the sizes.

color The color used to display a given record is stored with the dataset and can be queried and set using the functions *getColors.ggobi()* and *setColors.ggobi()* respectively.

hidden We can control whether a point is to be displayed or not in the different plots by marking it as hidden or not. The functions *getHiddenCases.ggobi()* and *setHiddenCases.ggobi()* both query and set this status on the records in a given dataset.

Row groups *getRowGroups.ggobi()* and *setRowgroups.ggobi()*

One can read the data from a ggobi instance into R via the *getData.ggobi()* function.

```
getData.ggobi(.data=1)
g$getData()
g[1]$getData()
```

3.1 Modifying the Data

We have seen how to query and set many of the attributes of a dataset that control how the values are displayed. We now turn our attention to functions for changing the structure of the dataset: adding or removing a variable and replacing values. We have already seen how

setVariableName() setRowNames()
setVariableValues()

3.2 Removing Variables and Records

removeVariable.ggobi()

3.3 The Displays

We can count the number of displays or plot windows under the control of a *ggobi* instance with the function *getDisplayCount.ggobi()*. We can also get the number of plots within each display with the function *getPlotCount.ggobi()*.

It is often convenient to be able to refer to a *ggobi* display directly. Just as we can refer to *ggobi* and *ggobiDataset* objects, we can obtain references to *ggobiDisplay* objects. We use the function *getDisplays.ggobi()* with the argument *describe* being **F**.

We can get a list of objects describing the actual displays of a *ggobi* instance by calling *getDisplays.ggobi()* with the *describe* argument as **T**. This returns a list with an element for each display. Each of these elements is of class *ggobiDisplayDescription* and its elements are described above (see 1.2 above).

3.4 State of the Instance

We can determine which plot is active using the function *getActivePlot.ggobi()*.

setActivePlot.ggobi().

We can determine the state of the brush region in a variety of different ways.

getBrushGlyph.ggobi() getBrushLocation.ggobi() getBrushSize.ggobi()

At any point, the *ggobi* instance is in a particular mode. These modes include things such as brushing, point identification, editing lines, active scatterplot matrix, active 1D plot, active 2D tour and so on. The full list of the names of these modes can be obtained via the function *getModeNames.ggobi()*.

```
> getModeNames.ggobi()
 [1] "1D Plot"          "XYPlot"           "Rotation"         "1D Tour"
 [5] "2D Tour"          "Correlation Tour" "Scale"            "Brush"
 [9] "Identify"         "Edit Lines"       "Move Points"      "Scatmat"
[13] "Parcoords"
```

One can query the current mode in which the ggobi state is using the function *getMode.ggobi()*. Similarly, one can set the mode for the *ggobi* instance via the corresponding *setMode.ggobi()* function. This expects an argument which is one of the names of the modes returned from *getModeNames.ggobi()*. Note that not all modes make sense from a programmatic stand point. Some are descriptive and some are settable. Brushing, point identification, moving points are two that are most useful.

setBrushColor.ggobi() setBrushGlyph.ggobi()

```
> g <- ggobi("../ggobi/data/flea.xml")
> g$setMode("Brush")
[1] "XYPlot"
> g$setBrushColor(12)     # returns the old value - an un-named colour

0
```

setBrushLocation.ggobi() setBrushSize.ggobi()

Now we programmatically place the brush at a particular position on the display.

```
 g$setBrushLocation(50, 50)
 g$setBrushSize(100, 50)
```

Note that these are in pixel coordinates, and not the units of the plot.

We can determine which records are selected *getSelectedIndices.ggobi()*

```
> getSelectedIndices.ggobi()
Heptapot. Heptapot. Heptapot. Heptapot.
       21        35        36        40
```

3.5 Functions

setDataFile.ggobi() setDataFrame.ggobi()

getDatasetNames.ggobi() names(g)()

4 Controlling Plots

We now move onto the next level of the hierarchy, the displays. Each *ggobi* instance supports multiple simultaneous and linked displays. Each display in turn supports multiple plots within it. We can interrogate the list of displays and also the plots within a display. *()*

The *getDisplayOptions.ggobi()* returns the settings that control the way in which plots are displayed. The default values are given as

```
getDisplayOptions.ggobi()
          Points   Directed edges Undirected edges            Edges
            TRUE             FALSE             FALSE             TRUE
  Missing Values              Grid              Axes      Center Axes
            TRUE             FALSE             TRUE              TRUE
   Double Buffer              Link
            TRUE              TRUE
```

These are explained as

Points

Directed edges

Undirected edges

Edges

Missing Values

Grid

Axes

Center Axes

Double Buffer

Link

One can change some or all of these settings via the function *setDisplayOptions.ggobi()*. Modifications to these do not apply to existing plots. Instead, they are applied to plots that are created after they have been set.

setPlotRange()

setPlotVariables()

4.1 The `.ggobi` suffix

You may have noticed that almost of the functions in the ggobi interface end with the extension `.ggobi`. This makes things slightly akward. The primary reason for this naming scheme is to avoid name conflicts. (Namespaces might make it slightly simpler, but not significantly.) However, there is a simpler syntax than appending the `.ggobi` suffix, and it is one that I both prefer and encourage.

The functions defining the interface are global and almost all take a *ggobi* object on which to operate. This is the *S* style. The *C++* and *Java*^{TM} styles are more classically object oriented and would have find the appropriate function *within* the *ggobi* object, and not globally. So, we can use the same mechanism in R; namely, invoke a function within the *ggobi* instance.

An example of how this style works shows the simpler interface. We start by creating a *ggobi* instance, but this time we make certain to assign the return value. (We can also retrieve it using *getDefaultGGobi()*.)

```
g <- ggobi(system.file("../ggobi/data/flea.xml"))
```

Now, suppose we want to invoke the *getDisplayCount.ggobi()*. We can do this as

```
g$getDisplayCount()
```

and this returns the same result as calling *getDisplayCount.ggobi()* directly.

Any of the functions named *x.ggobi()* that take an argument named *.gobi* can be called in this manner. For example,

```
data(mtcars)
g$setData(mtcars)
```

Note that most of the functions that work on a dataset within a ggobi instance have a default value for the dataset – the first one. If this is a suitable default, then one can use the ggobi instance in this way to invoke the call. For example, suppose we want to get the glyph information for the only dataset in the flea example.

```
g <- ggobi(system.file("../ggobi/data/flea.xml"))
g$getGlyphs()
```

We are, in fact, dispatching the call to the dataset, not the ggobi instance. But this works because of the default argument for the *.data* argument in *getGlyphs.ggobi()*. A better programming practice using the same style would have us do the following

```
g[1]$getGlyphs()
```

This would then turn into the equivalent of

```
d <- g[1]  # d is now a ggobiDataset
getGlyphs.ggobi(.data=d, .gobi=d[["ggobi"]])
```

Of course, this means that we do not have to limit ourselves to the default dataset, but instead can identify it when applying the [operator to the *ggobi* object.

One of the benefits of overriding the $ operator is that it makes it less likely (i.e. slightly more difficult) for users to dereference its fields such as –*"red ref()"*. Since these are really hidden/opaque values that cannot be interpreted at the S level, making it harder to access them is a good thing.

Note that this mechanism use the naming conventions of the calls and is not guaranteed to work in general. It is more a convenient syntax that allows those of us who think in the *C++/Java*^{TM} style to be a little more at home. Since I think this is a more natural way to think about mutable operations, objects and foreign refferences, I think it is useful.

5 Programmatic Display Generation

The user can interactively create new plots of different types using the Display menu on the main panel of the *ggobi* instance. Similarly, we can programmatically generate the same types of plots via R commands. We can create instances of the standard plot types using the functions *scatmat.ggobi()*, *scatterplot.ggobi()* and *parcoords.ggobi()*, described below. Each of these takes

- the variables to plot,
- the dataset in which these are to be found, and
- the ggobi instance in which the dataset is to be resolved.

As with most of the functions, one can specify a dataset instance *ggobiDataset* as the value of *.data* argument and this will also specify the *.gobi* argument. Alternatively, one can specify the index or name of the dataset and it will be resolved against by looking in the ggobi instance, which may be the default one.

All that remains to create one of these displays is to specify the variables that are to be displayed. These can be specified by name or by index. The former is more readable and invariant to adding or removing variables. The names can be retrieved via the *getVariableNames.ggobi()* function or the *names()* method for the *ggobiDataset* class.

```
getVariableNames.ggobi(.data = 1, .gobi=g)
names(g[1])
```

If one specifies the values as indices, recall that the first variable is given by 1 and the last by

```
ncol(g[1])
```

Now, here are brief descriptions of each of the plots and some examples for creating them.

scatmat.ggobi()

```
g$scatmat()
g$scatmat(c("tars1", "tars2"))
  # A non-symmetric layout with
```

```
  # x[1] versus y[1], x[2] versus y[1]
  # x[1] versus y[2], x[2] versus y[2]
g$scatmat(c("tars1", "tars2"),c("head","aede1"))
```

scatterplot.ggobi()

parcoords.ggobi()

```
g$parcoords("tars1", "tars2", "aede1")
parcoords.ggobi(1,2,4)
g[1]$parcoords(1:3)
```

5.1 Programmed, Non-standard Layouts

These types of plots are more than adequate for most uses. However, they do place different plot types in different windows. For instance, scatterplots and parallel coordinate plots are put in different displays and the user must arrange these windows manually to best show the effects of the linked plots. What is more desirable is to be able to put a collection of plots into a single window and control how they are arranged. In this section, we describe how to do this using R commands. Providing controls to create arbitrary displays in *ggobi*'s graphical interface would complicate it unecessarily. But providing programmatic control in R is another example of the advantages of this style of interface.

The steps for creating a non-standard display are the following:

- create plot *descriptions*, each defining the type of plot and the variables
- create a layout specification indicating the cells of a grid that each plot is to occupy
- combine the descriptions and layout information in a call to *plotLayout()*.

The different plot descriptions are created with the functions *ashDescription()*, *parallelCoordDescription()*, *scatterplotDescription()* and *scatmatrixDescription()*. Each of these takes an arbitrary number of variable identifiers specifying which variables are to be displayed in this plot. Obviously a scatterplot should only be given two variables. These identifiers can be given as variable names or indices. Note that they are not resolved at this point. Instead, they are held in the description and only resolved when the description is converted to an actual plot. This allows the same description to be used with different datasets, effectively making it a template. One can optionally specify the dataset and ggobi instance when creating the description and hence narrow or specialize the description to be used only with that dataset. The concept of a template is to allow one to effectively "program" a plot without actually writing an S expression or function.

The next piece to specify when creating a non-standard plot is the locations of the different plots within the display. This is reasonably simple and is based on a grid and is common in different GUI toolkits. One decomposes the display window (container) into an $r \times c$ grid. One positions a plot by giving the left and right, and top and bottom cells which it is to occupy.

The function *gtkCells()* is used for generating the basic specification for an $r \times c$ grid. It returns the positions in which each plot occupies one cell. One can modify this object to create different

layouts. For example, suppose we want to have 3 plots, with the first two being of equal size and occupying the top row, and the third occupying the entire last row. To do this, we create a 2×2 grid and postition the first two plots in cell $(1, 1)$ and $(1, 2)$. The third plot goes in the two cells of the bottom row $(2, 1), (2, 2)$.

This can be specified with the following commands.

```
l <- gtkCells(2,2)[-4,]
l[3, "right"] <- 3
```

We drop the last cell and then modify the 3^{rd} entry to span both columns by making the right-hand column it borders be 3 so that it occupies cells 1 and 2.

We can then couple the descriptions and the layout and the descriptions and create the display using *plotLayout()*. This takes an arbitrary number of description objects and the layout via the *cells* argument. It resolves the variables in the descriptions using either

a the description's own dataset and ggobi specification, or

b the values specified in the call to *plotLayout()*.

If the layout is very simple and puts one plot in each cell, one need only give the dimensions of the grid via the *mfrow* argument.

An example may help. (These examples and screen shots can be found at `http://www.ggobi.org/RS`

```
plot1 <- ashDescription("tars2")
plot2 <- scatmatrixDescription("tars1", "tars2", "head")
plot3 <- parallelCoordDescription("tars1", "tars2", "head")
```

Now, we will use the layout that we generated earlier.

```
g <- ggobi(system.file("data/flea"))
plotLayout(plot1, plot2, plot3, mfrow=c(2,2), cells = l, .gobi = g)
```

We can create a different layout and reuse the plot descriptions. This time, we will create a zig-zag layout, with a plot in the top-left, middle-right and bottom-left cells. We do this by extracting only the first, fourth and fifth cells.

```
l <- gtkCells(3,2)[c(1,4,5),]
plotLayout(plot1, plot2, plot3, mfrow=c(3,2), cells = l, .gobi = g)
```

The function *resolvePlotDescription()* is responsible for instantiating the individual plots from their descriptions.

We can organize the descriptions into a collection of plots via the function *plotList()*. This is not currently used but we will define methods for the class *ggobiPlotlist* in the future.

5.2 Functions

plotLayout()
ashDescription(), parallelCoordDescription(), scatmatrixDescription() scatterplotDescription()
gtkCells()
resolvePlotDescription().

6 Using R Functions in *ggobi*

So far, we have managed to provide a way for R users to integrate the visualization of data with their other operations on that and related data. The communication between R and *ggobi* has been controlled entirely by the R user. We have, however, not made direct use of *ggobi* being embedded in R to implement functionality that *ggobi* requires by borrowing it from R. More specifically, *ggobi* has not directly accessed any of the functionality of R for its computations. For the most part, this has not been necessary. However, consider the following example.

Suppose we have a new methodology for smoothing and an algorithm implemented in the S-language. We want to be able to use this in place of *ggobi*'s own smoothing functionality. What we want to have happen is that our S function is called each time the user moves the slider to select a different bandwidth for the smoothing. Then, our function is given the values for the x and y variables involved in the smoothing and the new bandwidth value. It performs its computations and returns the smoothed y values (predicted for the given x values). We can write a simple R function to do this. Suppose we use *loess()* to perform the smoothing. Our function is then defined as follows:

```
library(modreg)  # to ensure the loess function is available.
function(x, y, w) {
  predict(loess(y ~ x, data.frame(x=x, y = y), span = w))
}
```

Now, we have to arrange to have it called by ggobi when the value of the slider controlling the bandwidth is changed. We do this with the S function *setSmootherFunction.ggobi()*. This registers the function with ggobi and is invoked at the appropriate times.

(Note that I cannot currently find ggobi's smoothing slider given the recent changes to the interface.)

We can also find the definition of the current S smoothing function registered with *ggobi* using the function *getSmootherFunction.ggobi()*. This returns the function object itself.

6.1 Identifying Points

A second form of having ggobi interact with R is when we are using *ggobi*'s *Identify* mode which allows the user to move around a plot and have the label of a point be displayed when the mouse moves over it. Rather than simply displaying the label, an R user might want to use this "browsing" in a more programmatic way. For example, we might use it as a more advanced form of the *locator()* function, and store which points were identified. Similarly, as each point is identified, we might highlight the corresponding row in a spreadsheet, or update an R plot appropriately. In other words, we can link actions in a *ggobi*plot not only to other *ggobi* plots, but to other objects accessible from S and all using *ggobi*'s notification mechanism.

We first illustrate how to program the simplest of these interactions, namely, just printing the index of the selected/identified observation each time the user moves over a point in the plot. The following R function does exactly what we want.

```
f <- function(id, display) {
```

```
    print(id)
}
```

ggobi calls it with two arguments: the index of the observation being identified and also an object of class *ggobiDisplay* which identifies which display the identification is being performed and indirectly (via the display) in which dataset the observation resides.

We can now register this with ggobi

```
setIdentifyHandler.ggobi(f)
```

We can switch to Identify mode programmatically via the expression

```
setMode.ggobi("Identify")
```

and now to test this, move your mouse around the currently active plot and watch what is printed out on the R console.

One can easily imagine doing something more interesting, such as highlighting a row in a spread-sheet, etc. Our example would not change except for replacing the call to *print()* with a call to something that modified the view of the spread-sheet. The key thing about this example is that we can perform our response solely in terms of the information in this call.

Now we will try to illustrate how to create an R function that accumulates a list of the indices of the identified points. Each time the user identifies a point, *ggobi* will call our R function and we will add it to the list. Since there will be multiple calls to this function – one each time a point is identified – we must have a way of storing the indices of the points across these different calls. We use an R closure for this purpose. We define a function which returns two functions. When either of these two functions are called, they can see the same shared variables that were defined within the top-level function in which they themselves were defined.

```
identifyGen <-
function() {
    identified <- integer(0)

    identify <- function(id, display) {
       # append the new id onto the
     identified <<- c(identified, id)
    }

    list(identify = idenfity, indices = function() unique(identified))
}
```

So now we can invoke the function *identifyGen()* and register the function `identify` function it returns as the R-ggobi identify handler.

```
i <- identifyGen()
setIdentifyHandler.ggobi(i)
setMode.ggobi("Identify")
```

Once again, move the mouse around the active plot. When you have selected several points, return to the R prompt and issue the command

```
i$indices()
```

You should see the indices of the points that you selected.

As with the *getSmootherFunction.ggobi()*, one can query the currently registered identify handler using *getIdentifyHandler.ggobi()*.

We chose to implement the simple example here to avoid having to depend on the existence of other R packages and software, etc. The idea should be clear and it should be reasonable clear how to program other style of interaction.

The smoothing and identify examples are presented here and implemented in the R-ggobi interface not because they are themselves the most compelling choices for allowing ggobi to call R functions. Rather, they are examples to illustrate that this style of interaction and extensible, programmatic customization of ggobi is possible. We, the developers of *ggobi*, can add hooks so that callbacks can be registered for different *ggobi* events. We are very open to suggestions as to what are useful events and what information should be supplied to callbacks.