Using the R-GGobi Link

Andreas Buja
Di Cook
Deborah Swayne
Duncan Temple Lang

May 7, 2004

Abstract

In this document, we describe some of the features of the R-GGobi link. We illustrate some of the *common* uses and the different functions to both query and modify the state of the GGobi. This serves as a form of a tutorial.

More sophisticated things can be done with the link, including programming customized GUIs containing linked GGobi plots, writing new linking rules in S, and responding to GGobi events, create GGobi plugins written in R.

1 Getting Started

The first thing to do is create a ggobi instance. Each ggobi instance is initially associated with a single dataset. There are two ways to create this: one specifies the name of the file to read and the other takes a data frame or matrix in R. The first takes a string, exactly as it would be passed to GGobi on the command line, such as

```
> ggobi("../data/flea")
```

The second form passes the data frame as the first argument:

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

Both of these create the basic ggobi GUI in the same manner as the stand-alone application via the command line. One can control how the GGobi instance is created by passing a vector of strings as arguments which are processed as command line arguments. When specifying the name of a data file, one can also specify the mode of the data, e.g. whether this is XML, ASCII, binary, etc.

We can create a GGobi instance with no data and then subsequently add one or more datasets to it.

```
> ggobi()
> setData.ggobi(data.frame(a=rnorm(100), b=runif(100))
```

We can add datasets to any GGobi instance, regardless of whether we started it with or without a dataset.

The opposite of creating a ggobi instance is closing its displays and removing it from the session. This is done via `close.ggobi`. As usual, the particular instance to operate on is identified by the *.gobi* argument and defaults to the default instance.

I have seen this crash, but can't yet see why.

At any point, the dataset can be augmented by adding new variables, one at a time:

```
addVariable.ggobi(1:32, name="foo")
```

These will then appear in the GGobi main window and can be used as a regular variables in plots, etc.

We should also note that we can have any number of GGobi instances in existance simultaneously, with the same or different datasets.

```
g1 = ggobi(mtcars)
g2 = ggobi(ggobi.find.file(c("data", "flea.xml")))
```

Almost all of the functions in the `Rggobi` package take a GGobi instance as an argument. (The exceptions are one that operate on the global level.) By default, the last GGobi instance created is used as the target of the operation and one does not have to specify it. If one wants to address a different GGobi, one can specify it via the *.gobi* parameter. We have introduced a more object-oriented syntax for invoking these `foo.ggobi` functions. If we have a GGobi instance of as an S variable, say *g()*, then we can invoke the function `foo.ggobi` as g$foo(). This takes care of expanding to `foo.ggobi` and passing *g()* as the *.gobi* argument. For example,

```
g1$addVariable(1:32, name = ""foo")
```

is equivalent to the command

```
addVariable.ggobi(1:32, name="foo", .gobi = g1)
```

The idea is that we are changing the state of the underlying GGobi instance.

2 System Values

The representation of points or observations within a plot is controlled by its *glyph*. Each observation has its own glyph setting. This controls the type and size of glyph. The possible values for the glyph type and size are fixed for a ggobi session. This set of possible values can be queried via the functions `getGlyphTypes.ggobi` and `getGlyphSizes.ggobi`

```
> getGlyphTypes.ggobi()
 +  x or fr oc fc  .
 1  2  3  4  5  6  7
> getGlyphSizes.ggobi()
[1] 0 1 2 3 4 5 6 7 8
```

Each ggobi instance has its own color map. The color of each observation is specified as a row index into this colormap. Each row of this map or table is an RGB color value.

```
> getColorMap.ggobi()
         Red        Green      Blue
1  1.0000000 0.0008697642 0.0000000
2  1.0000000 0.2699931334 0.0000000
3  1.0000000 0.5499961852 0.0000000
4  1.0000000 0.8399938964 0.0000000
5  1.0000000 1.0000000000 0.0000000
6  0.0000000 0.7499961852 1.0000000
7  0.5099870 0.4399938964 1.0000000
8  0.6000000 0.8000000000 0.2000000
9  0.0000000 0.9799954223 0.6000000
10 0.7299916 0.3299916075 0.8299992
```

One can set the color map with the function `setColorMap.ggobi`. To change, for example, the second row to the values $.8, .2.3$, we can use the following steps

```
mp <- getColorMap.ggobi()
mp[2,] <- c(.8,.2,.3)
setColorMap.ggobi(mp)
```

3 Brushing

One can determine which observations are contained within the current brushing region with the function `getSelectedIndices`. This returns a vector containing the (unordered) indices of the observations in the dataset that have been selected, together with their labels.

The function `isObservationSelected.ggobi` returns a logical vector with as many elements as there are observations in the dataset. Each element indicates whether the observation is within the current brushing region.

4 Programmatically Configuring a View

One can (will be able to) set the brush location and size directly via a function call. Additionally, one can control the glyph type, size and color for the points under the brush.

One can use the programmatic interface to capture views and replay them. For example, we might want to display the effect of identifying different groups of observations.

```
views <- list(c(20,20, 100,100), c(400,300), c(300,150,40,40))
setMode.ggobi("Brush")
for(i in views) {
  setBrushLocation.ggobi(i)
  print(getSelectedIndices.ggobi())
  prompt()
}
```

5 Creating New Windows and Plots

See **RSCreatePlot.c** for some details.

The graphical user interface for GGobi allows one to create new windows and groups or collections of plots. These collections are defined by the menu or button that is used to generate them and offers little in the way of allowing the user to specify what the type or content of the individual elements of the collection are. In R, we have a programming language and so we can use it to allow the user create these groups of plots and to add them to a particular window.

The current mechanism for creating plots from R is a two step process.

1. one creates descriptions of the individual plots that one wants to create. These are very simple, involving just the type of plot and identifiers for the variables it is to contain.

2. pass these plots to the function `plotLayout` to create the plots and lay them out within a new (or existing) window.

5.1 Plot Descriptions

There are 5 types of plot descriptions that one can create. The first 4 are the basic plots offered by GGobi. These are

- parallel coordinate plots;
- 1D ASH plot;
- scatterplot; and
- scatter plot matrix.

The 5^{th} type is a general container that is used to house any number of these basic types[1].

To create a description of a plot, one decides the desired plot type and calls the corresponding function Each of these takes the variables involved in the plot as their first arguments (or argument for the ASH plot). These can be

[1] At present, we don not allow containers within containers.

parallel coordinate plots	`parallelCoordDescription`
1D ASH plot	`ashDescription`
scatterplot	`scatterplotDescription`
scatter plot matrix	`scatmatrixDescription`

Table 1: Plot Description Functions

specified either as the names or indices (starting from 1) of the variables within a dataset managed by a ggobi instance. All of the description functions allow one to associate a particular ggobi instance and dataset with the description. These default to being unspecified and can be provided when the plot is being created. By specifying these, one indicates that the description is to be used with a particular ggobi and/or dataset.

We recommend that people use names to identify variables and also datasets.

```
plot4 <- scatterplotDescription("tars1", "tars2")
plot5 <- scatmatrixDescription("tars1", "tars2", "head")
```

5.2 From Descriptions to Plots

To instantiate the descriptions into visible plots, one uses the `plotLayout`. By default, this creates a new ggobi display - a separate window - associated with a given variable selection panel. Into this, it is adds each of the plots created by converting the corresponding description into an actual plot. By default, the plots are added vertically to form a $k \times 1$ table or grid, where k is the number of plot descriptions passed to the function.

As noted in the previous paragraphs, the plots do not have to be homogeneous and can even be plots within plots (i.e. the container of plots).

The layout of the plots can be controlled in two ways. Firstly, we can specify a different arrangment for the grid into which the plots are added. Rather than accepting the default $k \times 1$ layout, one can specify the number of rows and columns in that same way that one partitions an *R* graphics device using *mfrow*. This is an integer vector containing two elements: the number of rows and the number of columns of the grid, in that order.

```
plotLayout(plot4, plot5)
plotLayout(plot4, plot5, mfrow=c(1,2))
```

More fine grained control over the positioning of the individual plots can be obtained using the *cells* argument. One can specify the left, right, top and bottom cells that are to border a particular plot. For example, suppose we have a 2×2 screen and we want to display 3 plots within it. The first two plots are to occupy the first row and the final plot is to span both columns of the second row. To do this, we specify the three plot descriptions and provide a 4×2 matrix containing the boundaries of each plot as the *cells* argument. Each column in this matrix contains the index of the bordering cells in the grid layout. The first plot therefore is given as 1 for the column on the left, 2 for the column on the right and 1 and 2 as the top and bottom rows. Similarly, the second plot which is to appear in the top-right cell is given as $(1, 2, 1, 2)$, differeing only in the column column cells.

The final plot occupies the second row and so has top and bottom rows of 2 and 3 respectively. The column specifications are $(1, 3)$ indicating that first column is on the left and the right side of the plot borders the third column of the 2×2 grid.

The result is something like

The function `gtkCells` is a simple way to create the cell-border specification for a regularly spaced $r \times c$ grid. One can the subset the resulting matrix and insert other specifications to create the appropriate specification for a particular cell layout for a collection of plots.

As an example, suppose we have a 3×2 grid and we want to place 3 plots in the following positions: $(1, 1)$, $(2, 2)$ and $(3, 1)$ where each pair gives the row and column indeces.

```
plotLayout(plota, plotb, plotc, mfrow=c(3,2),
               cells = t(gtkCells(3,2)[c(1,4,5),]))
```

Note that the result of `gtkCells` is (currently) a matrix $(r * c) \times 4$ matrix and must be transposed to be used in `plotLayout`. Also, the elements are organized row-wise, i.e. the first r elements correspond to the cells in the first row of the grid, the second r elements to the next row, etc.

Rather than creating a new display for the collection of plot descriptions being instantiated, one can add them to an existing ggobi display window. This is usually done when it is more convenient to create an empty display and add plot descriptions as they are available. To add a plot to an existing display, merely provide the index of the display within the specified ggobi instance as the argument *display*. This is the value returned from an earlier call to `plotLayout`.

```
dpy <- plotLayout(mfrow=c(3,2))
plotLayout(plot1, plotb, plotc, cells= t(gtkCells(3,2)[c(1,4,5),]),
                            display = dpy)
```

One thing to note about creating an empty display is that it will become the active display and all variables in the control panel will become unselected.

5.3 Nested/Container Plots

The scatterplot matrix and parallel coordinate plots can contain more than one sub-plot. However, they can be treated as a single cell of the grid/table layout. The entire plot or *embedded display* receives the space allocated the particular cell (as specified by the *cells* argument.) Each sub-plot is then given the appropriate amount of that space. In a parallel coordinates plot, the sub-plots are arranged in $1 \times k$ table (i.e. one row and k columns, one per variable). In a scatter plot matrix, the sub-plots are arranged in a regular grid of dimensions $k \times k$, all having the same dimensions.

In the next version of this code (when it is tidied up and consolidated), we will use the general *GGobiPlotList* function that acts as a nested description of plots along with their layouts. In this way, one can provide recursively defined nested plots that are self-describing.

6 Managing Windows

Because one can easily create multiple ggobi instances and many displays within each instance, managing the display windows can be difficult. Accordingly, we provide some facilities for programmatically controlling them. The first set work on the displays. The functions `getActivePlot.ggobi` and `setActivePlot.ggobi` operate on either a display or a display and sub-plot object.

Firstly, one can move one or all of the windows within a ggobi instance to the background or foreground. This uses the function `raisePlot.ggobi` and the argument *raise* indicates whether the windows should be lowered or raised. The former pushes them below all the other windows on the desktop. The latter raises them above the other windows. The location of each window remains unaltered, and merely its "depth" or stacking order is changed.

Similarly, one can iconify and de-iconify one or more windows. The function `iconify.ggobi` provides this functionality via the *icon* argument.

7 Callbacks

Associating R function calls with events in GGobi is a valuable way to allow customization of the existing interactivity provided by GGobi. Consider an example where we have a scatterplot of two variables X and Y. As we identify individual points in the scatterplot, we might want to examine the result of fitting the regression of Y on X with that point omitted. GGobi does not currently provide such a facility. However, it is trivial in R. By allowing one to register an R function to be called when a particular event (the identification) occurs, we can easily arrange to update other plots. Given the available primitives with which we can control ggobi or other graphics devices, we can

Clustering or model selection might serve as other examples. The basic idea is that we would have some data that represents the fits for different models. The data frame is made up of, say, 2 columns - Model identifier (an integer) and the fit statistic (e.g. Cp, etc.). As we identify or brush over one or more points, we want to link with a plot in another ggobi using the indeces of the other model. For example, we might want to show the residuals resulting from that fit. This is non-trivial in GGobi since the data does not correspond to the model it employs which is a rectangular array of data. (Again, an object oriented approach would be convenient, but still difficult.) Thus, we need two ggobi

instances, with different but related data. Additionally, we need a mechanism to specify linking between plots and displays within and between ggobi instances.

Covering all such cases is not feasible in GGobi both from a programming perspective and additionally as an issue of complexity in the user interface. Instead, a programmatic mechanism for "linking" or associating/connecting displays and or plots within displays both between and within ggobi instances is necessary. Furthermore, while many of the common linking actions can be performed efficiently in GGobi itself, allowing arbitrary R functions to be invoked for these connection events is important to allow new styles of linking to be explored and implemented easily.

The basic functions for performing these types of actions is `setIdentifyHandler.ggobi`.

```
setIdentifyHandler.ggobi(function(i) {print(i)})
```

More arguments should be supplied to this function.

We may also want to make this plot specific, thus allowing the data to be stored in the function rather than having to be determined at evaluation time. For example, suppose we want to know what variables are being identified and in what ggobi. We can create a closure with that information in the environment.

```
handlerTemplate <- function(x,y, ggobi) {
  handler <- function(which) {
    cat("Observation", which, "in", ggobi,"\n")
  }

  return(handler);
}
```

Of course, we can do this for the ggobi instance as it stands now.

To remove the handler

```
setIdentifyHandler.ggobi(NULL)
```

See `IdentifyProc()` in **ggobi.h** and add the appropriate set and get for the ggobid specific instance. Already arrange to do callback.

We can make this slightly more flexible and efficient. The `setIdentifyHandler.ggobi` can specify how many arguments the function wants to receive. This can be stored with the function reference in the user˙data field of the ggobi handler structure. Then, when we go to invoke the function, we can determine how many arguments to pass. This allows some functions to avoid the overhead of creating unecessary arguments that will never be used. It would be ideal if there was an easy way for the R function to access the "environment" of the C-level calling routine (as is possible in Omegahat due to the seamless access between the interpreted and native language).

Bad things happen when we get an error in the R functions. We must handle this with a better jumping mechanism.

8 API Routines

In this section we describe the different R/S functions that one can invoke to query and modify the state of the GGobi session. There are some common concepts shared between most of these functions.

Firstly, there are some global variables that are properties of the entire ggobi system. These are things such as the color table, the glyph types and sizes. These can be considered fixed, but queryable.

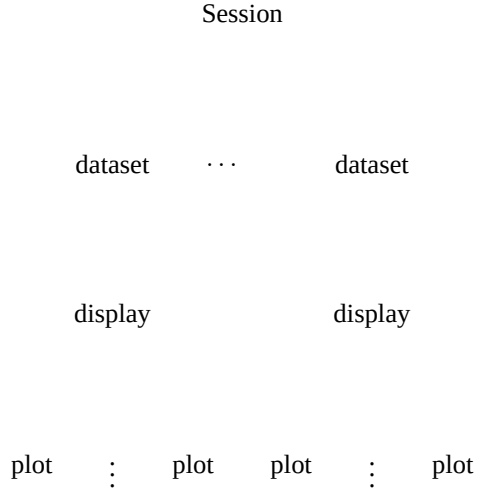
Within a ggobi session, one can view different datasets simultaneously. Each dataset corresponds to its own ggobi data instance within the single session. (Note that this is different from xgobi in which only one dataset could be introduced into the a process.)

Within each ggobi instance, one can have multiple windows, also known as displays. Each display has one or more plots within it. The plots can be a

- 1 variable plot such as ASH plots,
- scatterplot of two variables
- scatter matrix of 2 or more variables

- parallel coordinates plot of k variables.

This hierarchical setup is displayed in the following figure.



When operating with multiple datasets in the ggobi session at any one time, it is important to identify to which instance a command should be directed. For example, if we wish to get information on the variables within a particular instance of a ggobi instance, we can call `getVariableNames.ggobi`. However, we must specify the ggobi instance in question. All the functions that operate on a ggobi instance have an optional argument, *.ggobi*, which is an integer. If omitted, this defaults to the currently active ggobi instance.

This allows us to send repeated commands easily to a ggobi instance.

```
> ggobi(args="../data/tes")
> ggobi(args="../data/flea") #
> getVariableNames.ggobi()
[1] "tars1" "tars2" "head"  "aede1" "aede2" "aede3"
> setDefaultGGobi(1)
> getVariableNames.ggobi(.gobi=2)
```

Additionally, commands intended for another can be directed without setting that instance to the default.

[What is intended here?]

By default, this is This identifies

have multiple ggobi instances in existence at any one time. Each ggobi has its own

8.1 getData.ggobi

Since the user can load data in a non-programmatic way, there is an opportunity for XGobi to have data that has not been introduced to the R/S session. As a result, we need a mechanism to retrieve the current values of the active data set. This is especially important if we allow the data to be edited in any way.

To get the R data values from a GGobi dataset, one can use the function `getData.ggobi`. This takes the dataset identifier (either a reference, index or name) and the GGobi instance. (By default, it fetches the first dataset in the active GGobi.) The return value is a matrix of numeric values containing the data values. The names of the rows and columns are the XGobi row labels and variable names respectively. *We might make this a dataframe in the future in order to handle categorical variables.*

Note that subsetting on a ggobi instance reference (basically an integer returned by `getDefaultGGobi` and `ggobi`) works in a manner consistent with R, returning a reference to the corresponding dataset. Then one can uses this to access some or all of the values within that dataset.

```
g <- ggobi(mtcars)
g[1][1:10,]
```

```
g[[1]][1:10, c("mpg","cyl")]
g[[1]][1:10, 1:2]
```

For large datasets and situations where one wants to extract relatively few columns, it might be more efficient to use the `getVariable.ggobi` function. This avoids copying the entire dataset into an R structure, and works only on the specified variables. This returns a named list giving the values for the variables of interest. We do not automatically convert this to a dataframe as the names of the records in the GGobi data need not be unique. (This restriction for dataframes has been made optional in R recently (version 1.4.0) so we migh create a dataframe default soon.)

In the same way that we have overloaded the `[` operator, we have defined methods for the functions `dim`, `ncol`, `nrow`, `dimnames`, `names` which act on ggobi dataset instances.

8.2 getColors

8.3 getSelectedIndices

This needs attention to access the correct variable in the xgobi structure.

8.4 getNumGGobis

Returns the number of ggobi instances within this session. Each ggobi instance has its own data set. Of course, two instances may have the same dataset, but they are independent. We may allow them to be linked in the future. Again, the ability to specific which displays and even plots within them inter- and intra- ggobi instance is most useful in a programmatic interface.

8.5 getDefaultGGobi

8.6 setDefaultGGobi

Since there can be multiple ggobi instances, one must identify the instance to which instructions are directed. To simplify this, R supports the notion of an active or default ggobi. All functions that refer to a ggobi instance use the default ggobi if the *.gobi* argument is not specified. This value is obtained via the function `getDefaultGGobi`.

When a new ggobi instance is created (see `ggobi`), it becomes the default ggobi. One can switch the active instance to another using the function `setDefaultGGobi`.

The return value from `getDefaultGGobi` is an integer which is 1-based. This has class `"ggobi"`.

The value can be passed directly to the functions `.GGobiC` and `.GGobiCall`. It is automatically decremented by these to be 0-based, corresponding to the C routines.

8.7 getRowNames.ggobi

Returns the observation labels for the specified ggobi instance.

See also `setRowNames.ggobi`.

8.8 getSmootherFunction.ggobi

8.9 setSmootherFunction.ggobi

GGobi provides different smoothing algorithms. However, it is impossible for it to provide all possible implementations. To allow others to provide their own, we allow an R function to be registered with the ggobi session so that it is called when the user slides the smoothing parameter backward and forward. This, of course, is not as fast as implementations in C code. But it is useful to allow one to experiment and get an impression of the results. The R function is called with three arguments: the x and y data values and the bandwith of the smoother, taken from the slider. The first two are numeric vectors of length n and the bandwidth is a scalar. One can register a function that performs the smoothing and returns the predicted value something like:

```
library(modreg)
function(x, y, w) {
  predict(loess(y ~ x, data.frame(x=x, y = y), span = w))
}
```

8.10 ggobi

This is the function that creates a ggobi instance. One specifies the name of a file or a data frame (or matrix). The newly created ggobi instance becomes the default

See `setDefaultGGobi`.

8.11 getActivePlot.ggobi

8.12 setActivePlot.ggobi

Each ggobi instance has one display that is considered active and within that display, a sub-plot on which brush and identify operations will take effect. This is the active plot. These two functions maniuplate the currently active plot. The first returns an integer vector identifying the display and within-display plot that is considered active. The second sets the active plot to the one identified by the integer pair. In this case, the second element can be omitted or NA.

8.13 getCurrentDisplayType.ggobi

Each ggobi has a concept of an active window or display. Within each of these displays, there is one or more sub-plots. This function returns the name of this active plot. See `getViewTypes.ggobi`. Also, see `getActivePlot.ggobi`.

At present, the sub-plots in a display are all of the same type. As a result, the plot type is taken from the display. When and if we allow non-homegeneous plot types within a display, we will have to modify how this routine is implemented.

8.14 getDescription.ggobi

This gives a brief description of the specified ggobi instance. This gives the name of the file from which the data was read, the type of file format (e.g. xml, ascii, binary) and the dimensions of the data.

See also `getNumGGobis`.

8.15 getDisplayOptions.ggobi

This returns the current settings of the values that control how new plots are displayed. Each ggobi instance has a set of options which are used to create new plots. These are logical values that govern the appearance of these plots. This function returns a named vector of those values currently in existence.

8.16 getDisplays.ggobi

Returns a list describing each display (or window) of plots within the specified ggobi instance. Each element in the list identifies a display and gives the title of the window, the type of the plot(s) contained in the display (since they are currently homogeneous) and a list describing each of the plots. The plot description depends on the type of plot. Generally it contains a list of the variables within that plot. This is an integer vector giving the indeces of the variables. The names of the elements in this vector are the names of the variables.

What about subsets in effect when the display was created, etc. This doesn't really apply as the plots are updated when the set of hidden observations is modified. If one wanted to display different views of the same data with a different collection of hidden variables, one could open multiple ggobi instances.

Some of this might change when we experiment with moving these types of attributes (color, hidden, etc.) to sub-plot instances rather than having them globally in the ggobi instance.

8.17 getFileNames.ggobi

Returns the names of the data source (e.g. file, URL, database) from which the datasets in GGobi were read. If the data was explicitly set from an R object, this returns a description supplied to the ggobi instance. The mode `getDescription.ggobi()[["Data mode"]]` of the data can be used to determine the format or source of the data

One can get slightly different information describing the data in a GGobi instance using `getDescription.ggobi`. Alternatively, one can first obtain a reference to a particular dataset within the GGobi instance and then ask get the name of the file by looking at the data element in that object. For example,

```
g <- ggobi(system.file("data", "flea", package="ggobi"))
g[1]$data
```

The subsetting on *g()* is short-hand for

```
getDatasetReference.ggobi(1, .gobi=g)
```

8.18 getGlyphs.ggobi

Return a description of the glyph being used for each of the observations.

8.19 getEdges.ggobi

8.20 setEdges.ggobi

The first of these obtains an $r \times 2$ matrix where r is the number of edges in the given dataset. Each element contains the indices of the observations which are connected. The second function sets the connected observations. The format is similar to the return value of `getEdges.ggobi`. A matrix with 2 columns can be supplied to identify the observations to be connected by each line segment. Alternatively, two integer vectors can be supplied. The user has the option to add the specified segments to the existing ones or replace the latter with the specified connections.

8.21 getVariableIndex.ggobi

This is used to map one or more names into the corresponding variable indices used by the ggobi instance. It uses `getVariableNames.ggobi` and matches each of the elements passed to it. The resulting values are based on a 1-based counting scheme and must be decremented by 1 for use in a `.Call`.

8.22 getVariableNames.ggobi

Returns a character vector containing the names of the variables in the specified ggobi.

8.23 getViewTypes.ggobi

The Ggobi system supports several different plot types. This function returns a named integer vector. The names are descriptions of the types of plots. The values in the vector are the symbolic constants used within the C code to identify the plot type. These values are rarely used since functions to create instances of the different plot types are available. See `scatmat.ggobi`, `scatterplot.ggobi`, `parcoords.ggobi`.

8.24 parcoords.ggobi

8.25 scatmat.ggobi

8.26 scatterplot.ggobi

Each of these functions creates a new display/window and adds it to the specified ggobi instance. Each provides a mechanism for specifying what variables should be contained in the plot and this provides greater specifiability than

with the graphical interface. The specification of the variables can be done either by name or by index. If using names, these should match

- parcoords This creates a Parallel Coordinates plot. The variable

8.27 setData.ggobi

Specify a data frame or the name of a file

8.28 setDataFile.ggobi

8.29 setDataFrame.ggobi

This sets the dataset for the particular ggobi instance to the contents of the dataframe. Each column in the data frame corresponds to an observations. The row names of the data frame are used as the observation labels in the ggobi displays.

8.30 setDisplayOptions.ggobi

This controls the characteristics of plots that are subsequently created within the specified ggobi instance. These control issues such as

- whether lines are drawn,
- lines are directed or undirected
- missing values are displayed (not implemented)
- axes are shown.

8.31 setRowNames.ggobi

This sets the observation identifiers for the specified rows in the ggobi instance dataset. These are used in identifying points within plots.

The return value is a vector of the previous labels for the specified rows/observations.

8.32 `getBrushSize.ggobi`, `setBrushSize.ggobi`, `getBrushLocation.ggobi` & `setBrushLocation.ggobi`

8.33 `close.ggobi`

This destroys the specified ggobi instance, closing all its windows/displays and the main control-panel window.

9 Developer Functions

9.1 .ggobi.symbol

This is of little or no interest to the regular user. It is merely a function that takes the name of a C routine and maps it to the name of a routine in the GGobi chapter. It does so by prefixing the name with the "unique" identifier RS˙GGOBI in an effort to avoid symbol conflicts with other libraries.

9.2 `.GGobiCall` & `.GGobiC`

There is an interesting "bug" in the `.C` and `.Call` functions. If one creates a wrapper as with `.GGobiCall` and `.GGobiC` and pass a named argument, the gets symbol name of the routine being invoke becomes corrupted.