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.

Using the R-GGobi Link

Andreas Buja
Di Cook
Deborah Swayne
Duncan Temple Lang

September 2, 2000

1 Getting Started

The first thing to do is create a ggobi instance. Each instance is associated with a particular 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, such as

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

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

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

Both of these create ggobi in the same manner as the stand-alone application via the command line. One can pass a vector of strings as arguments which are processed as command line arguments. When specifying the name of a file, one can specify whether this is XML, ASCII or binary.

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

We can create a ggobi instance with no data and set the data in a later command.

```
> ggobi()
```

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 or reduced by adding or removing variables.

```
addVariable.ggobi()
removeVariable.ggobi()
```

2

3 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. (In the near future, these may be indepdently settable across sub-plots.) The possible values for the glyph type and size are fixed for a ggobi session. They 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)
```

4 Brushing

One can determine which observations are contained within the current brushing region with the function `getSelectedIndices.ggobi`. This returns a vector containing the (unordered) indices of the observations in the dataset that have been selected. (For efficiency purposes, the labels of the observations are not returned with the vector. This can be easily changed.)

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.

5 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) {
  setBrush.ggobi(i)
  print(getSelectedIndices.ggobi())
  prompt()
}
```

6 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.

6.1 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

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

Table 1: Plot Description Functions

plot). These can be 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")
```

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

6.2 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)
plotLayout(plot1, plotb, plotc, cells= t(gtkCells(3,2)[c(1,4,5),]),
                            display = dpy)
```

6.3 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.

7 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.

8 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)})
```