An introduction to rggobi

Hadley Wickham, Michael Lawrence,
Duncan Temple Lang, Deborah F Swayne

Introduction

The rggobi package provides a command-line interface to GGobi, an interactive and dynamic graphics package. Rggobi complements GGobi's graphical user interface, providing a way to fluidly transition between analysis and exploration, as well as automating common tasks. It builds on the first version of rggobi to provide a more robust and user friendly interface. In this article, we will show you how you can starting using rggobi now, and learn some ways to gain insight into your data using a combination of analysis and visualisation.

We do assume some familiarity with GGobi. If you're not familiar, but would like to learn more have a look at the GGobi web site, `http://www.ggobi.org`, especially the demos of GGobi's capabilities found at `http://www.ggobi.org/docs`. You will also need to have a copy of GGobi installed before continuing: download a version for your platform from `http://www.ggobi.org/downloads`. You can then install rggobi and it's dependencies using `install.packages("rggobi", dep=T)`.

This article introduces the three main components of rggobi, with examples of how you might use them in day-to-day tasks:

- Getting data into and out of GGobi.
- Modifying observation-level attributes, or automatic brushing.
- Basic plot control.

We will also disucess some advanced techniques such as creating animations with GGobi, edges and longitudinal data. Finally, a case study shows how to use rggobi to create a visualisation for a statistical algorithm: manova.

Data

Getting data from R into GGobi is easy: `g <- ggobi(mtcars)`. This creates a `GGobi` object called `g`. Getting data out isn't much harder, just index that `GGobi` object by position, `g[[1]]`, or by name, `g[["mtcars"]]`, `g$mtcars`. These return `GGobiData` objects which are linked to the data in GGobi. These act just like regular data frames, except that changes are synchronised with GGobi. You can get a static copy of the data using `as.data.frame`.

Once you have your data in GGobi, it's easy to do something that was hard before: finding multivariate outliers. It is customary to look at uni- or bivariate plots to look for uni- or bivariate outliers, but higher dimensional outliers may go unnoticed. Looking for these outliers is easy to do with the tour. Open your data with GGobi and then change to the tour view and select all the variables. Watch the tour and look for points that are far away or move differently from the others—these are outliers.

Adding more data sets to an open GGobi is also easy: `g$mtcars2 <- mtcars` will add another data set named "mtcars2". You can load any file type that GGobi recognises by passing the path to that file. In conjunction with `ggobi_find_file`, which locates files in the GGobi installation directory, this makes it easy to load GGobi sample data. This example loads the olives data set included with GGobi.

```
ggobi(ggobi_find_file("data", "olives.csv"))
```

Modifying observation-level attributes, or automatic brushing

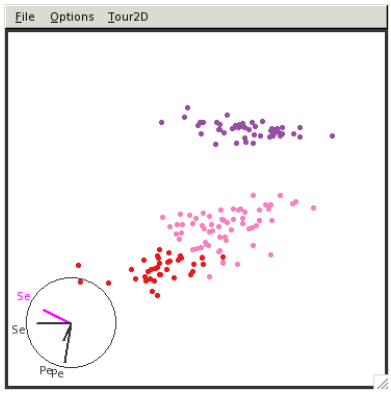
Brushing lets you interactively change the colour, glyph type and size of points. Brushing is linked, which means that these changes will propagate to every plot in which the brushed observations are displayed. Brushing includes shadowing, where points sit in the background and have less visual impact, and exclusion, where points are completed excluded from the plot. You can brush points "automatically", from R, using the following functions to:

- change glyph colour with `glyph_colour`
- change glyph size with `glyph_size`
- change glyph type with `glyph_type`
- shadow and unshadow points with `shadowed`
- exclude and include points with `excluded`

Each of these get or set the current values for the specified `GGobiData`. The getters are useful for retrieving information that you have created while brushing in GGobi, and the setters can be used to change the appearance of points based on model information, or to create animations. They can also be used to store, and then later recreate, the results of a complicated sequence of brushing steps.

This example demonstrates the use of the `glyph_colour` to show the results of clustering the infamous Iris data using hierachical clustering. Using GGobi allows us to investigate the clustering in the original dimensions of the data. The graphic shows a single projection from the grand tour.

```
g <- ggobi(iris)
clustering <- hclust(dist(iris[,1:4]),
  method="average")
glyph_colour(g[1]) <- cuttree(clustering, 3)
```



Another function, `selected`, returns a logical vector indicating if each point is currently under the brush. This could be used to further explore interesting or unusual points.

Displays

A `GGobiDisplay` represents a window containing one or more related plots. With rggobi you can create new displays, change the interaction or projection mode of an existing plot, or change which variables are displayed.

To retrieve a list of displays, use the `displays` function. To create a new display use the `display` method on a `GGobiData` object. You'll need to specify the type of plot you want (the default is a XY Plot) and which variables to include. For example:

```
g <- ggobi(mtcars)
display(g[1], vars=list(X=4, Y=5))
display(g[1], vars=list(X="drat", Y="hp"))
display(g[1], "Parallel Coordinates Display")
display(g[1], "2D Tour")
```

The following types of displays are available in GGobi:

Name	Variables
1D Plot	1 X
XY Plot	1 X, 1 Y
1D Tour	*n* X
Rotation	1 X, 1 Y, 1 Z
2D Tour	*n* X
2x1D Tour	*n* X, *n* Y
Scatterplot Matrix	*n* X
Parallel Coordinates Display	*n* X
Time Series	1 X, *n* Y
Barchart	1 X

After creating a plot you can get and set the displayed variables using the `variable` and `variable<-` methods. Because of the range of plot types in GGobi, variables should be specified as a list containing `X`, `Y` and `Z` character vectors listing the variable or variables to be used for each component.

```
g <- ggobi(mtcars)
d <- display(g[1],
  "Parallel Coordinates Display")
variables(d)
variables(d) <- list(X=8:6)
variables(d) <- list(X=8:1)
variables(d)
```

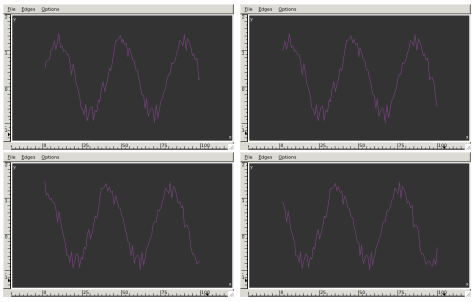
Another useful function is `ggobi_display_save_picture`, which saves the contents of a GGobi display to a file on disk. This is what we used to create the images in this document. This creates an exact (raster) copy of the GGobi display. If you want to create publication quality graphics from GGobi, have a look at the `DescribeDisplay` plugin and package at `http://www.ggobi.org/describe-display`. These create R versions of your GGobi plots.

Animation

Any changes that you make to the `GGobiData` objects are updated in GGobi immediately, so you can easily create animations. This example scrolls through a long time series:

```
df <- data.frame(
  x=1:2000,
  y=sin(1:2000 * pi/20) + runif(2000, max=0.5)
)
g <- ggobi_longitudinal(df[1:100, ])

df_g <- g[1]
for(i in 1:1901) {
  df_g[, 2] <- df[i:(i + 99), 2]
}
```



Edge data

Edge data sets are a special type of dataset. Instead of representing points, they represent connections, or edges, between observations. These can be used to represent many different types of data, for example, distances between observations, social relationships, biological pathways, and so on.

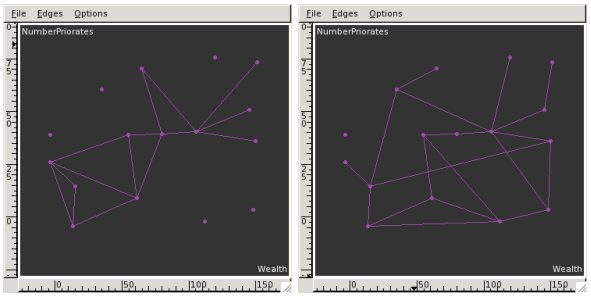
In this example we are going to visualise some data from the social network analysis package: marital and business relationships between Florentine families in the 15th century.

```
library(graph)
library(SNAData)

data(business, marital, florentineAttrs)

g <- ggobi(florentineAttrs)
edges(g) <- business
edges(g) <- marital
```

This example creates two edge datasets. We can use the edges menu in GGobi to change between the different edge sets.



How is this stored in GGobi? An edge dataset records the names of the source and destination observations for each edge. You can convert a regular dataset into a edge dataset with the `edges` function. This takes a matrix with two columns, source and destination names, with a row for each edge observation. Typically, you will need to add a new data frame with number of rows equal to the number of edges you want to add

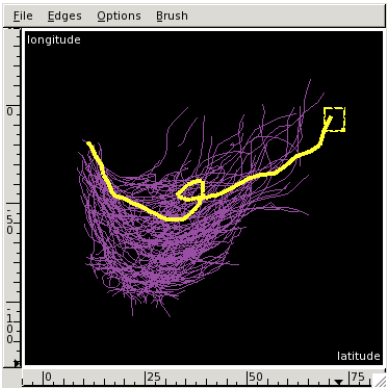
Longitudinal data

A special case of data with edges is time series or longitudinal data, in which observations adjacent in time are connected with a line. Rggobi provides a convenient function for creating edge sets for longitudinal data, `ggobi_longitudinal`. This will link together observations in sequential time order and is good for looking at time series or longitudinal data.

This example uses the `stormtracks` data included in rggobi. The first argument gives the dataset to use, the second the variable that specifies the time component, and the third variable separates different observations.

```
ggobi_longitudinal(stormtracks, seasday, id)
```

For regular time series data (already in order, with no grouping variables), just use `ggobi_longitudinal` with no other arguments.



Case study

Graphical methods are great for determining if two (or more) clusters of data are non-overlapping, but they are less useful for examining differences between means. This case study explores using rggobi to add model information to data; here will add confidence ellipsoids around the means so we can perform a graphical manova.

The first (and most complicated) step is to generate the confidence ellipsoids. The `ellipse` function does this. First we generate random points on the surface of sphere, by drawing *npoints* from a random normal distribution and then standardising each dimension. This sphere is then skewed to match the desired variance-covariance matrix, and its size adjusted to give the appropriate *cl*-level confidence ellipsoid. Finally, the ellipsoid is translated to match the column locations.

```
ellipse <- function(data, npoints=1000,
  cl=0.95, mean=colMeans(data), cov=var(data),
  n=nrow(data)
) {
  norm.vec  <-  function(x) x / sqrt(sum(x^2))

  p <- length(mean)
  ev <- eigen(cov)

  sphere <- matrix(rnorm(npoints*p), ncol=p)
  cntr <- t(apply(sphere, 1, norm.vec))

  cntr <- cntr %*%
    diag(sqrt(ev$values))
    %*% t(ev$vectors)
  cntr <- cntr * sqrt(p * (n-1) *
    qf(cl, p, n-p) / (n * (n-p)))
  if (!missing(data))
    colnames(cntr) <- colnames(data)

  cntr + rep(mean, each=npoints)
}
```

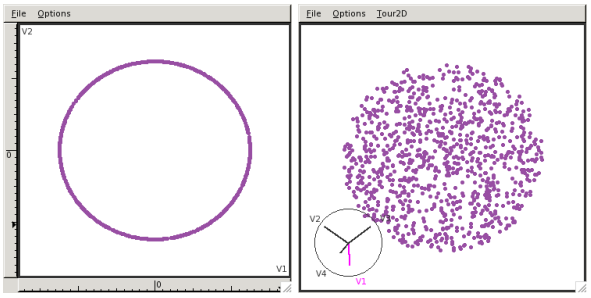
We can look at the output with ggobi:

```
ggobi(ellipse(mean=c(0,0), cov=diag(2), n=100))
```

```
cv <- matrix(c(1,0.15,0.25,1), ncol=2)
ggobi(ellipse(
  mean=c(1,2), cov=cv, n=100
))

ggobi(ellipse(
  mean=c(0,0,1,2), cov=diag(4), n=100
))

ggobi(ellipse(
  matrix(rnorm(20), ncol=2
))
```



In the next step, we will need to take the original data and supplement it with the generated ellipsoid:

```
manovaci <- function(data, cl=0.95) {
  dm <- data.matrix(data)
  ellipse <- as.data.frame(
    ellipse(dm, n=1000, cl=cl)
  )

  both <- rbind(data, ellipse)
  both$SIM <- factor(
    rep(c(FALSE, TRUE), c(nrow(data), 1000))
  )

  both
}
ggobi(manovaci(matrix(rnorm(30), ncol=3)))
```

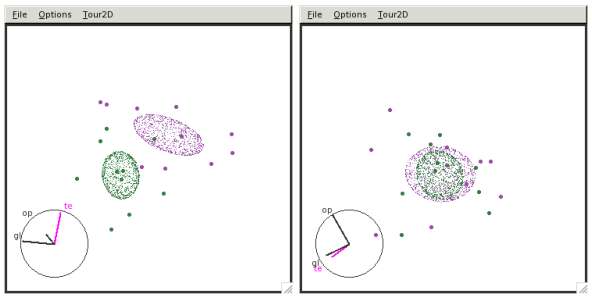
Finally, we create a method that will break a dataset into pieces based on a categorical variable and compute the mean confidence ellipsoid for each one. We will then use the automatic brushing functions to make the ellipsoid distinct, and colour each of the groups a different colour. Here we use 68% confidence ellipsoids so that non-overlapping ellipsoids imply a significant different in the means.

```
ggobi_manova <- function(data, catvar, cl=0.68) {
  each <- split(data, catvar)
  cis <- lapply(each, manovaci, cl=cl)

  df <- as.data.frame(do.call(rbind, cis))
  df$var <- factor(rep(
    names(cis), sapply(cis, nrow)
  ))

  g <- ggobi(df)
  glyph_type(g[1]) <- c(6,1)[df$SIM]
  glyph_colour(g[1]) <- df$var
  invisible(g)
}
```

These images show a graphical manova. You can see that in some projections the means overlap, but in others they do not.



Conclusion

GGobi is a powerful tool for data exploration, and the integration with R that rggobi allows a seamless workflow between analysis and exploration. Much of the potential of rggobi has yet to be explored, but some ideas are demonstrated in the `classifly` package, `http://had.co.nz/classifly` which visualises high-dimensional classification boundaries. We are also keen to hear about your work—if you develop a package using rggobi please let us know so we can highlight your work on the GGobi homepage.

We are currently working on the infrastructure behind GGobi and rggobi to allow greater control from within R. The next version of rggobi will offer a direct low-level binding to every function in GGobi, giving total control. We are also working on consistently generating events in Ggobi so that you will be able respond to events that you are interesting in from your R code. Together with the RGtk2 package, this should allow the development of custom interactive graphics for specific tasks, written purely with high-level R code.