Implementing Plugins for GGobi with R

Duncan Temple Lang

May 6, 2004

Abstract

We describe a mechanism by which one can create plugins for GGobi, a direct manipulation data visualization system, using the S programming language, and specifically the R environment. The ease of programming using S and the large collection of data analysis tools available in R make this a natural environment for providing extensions for GGobi. This is complimentary to the Java plugin mechanism also supported by GGobi and the R-GGobi interface which allows GGobi to be embedded and programmatically controlled by R users.

Plugins for GGobi provide a powerful mechanism for providing functionality that is not directly in ggobi. It allows one to add new ways to read data, save output from GGobi and provide new run-time features and tools. While one can develop plugins in C, the programming language used to develop GGobi, it is often more convenient to use a higher-level interpreted language such as R or Python. This partly because one can typically develope code more rapidly using these languages than with C. More importantly however it is because of the range of existing functionality related to data analysis in a language like R, Matlab or Octave.

1 R Plugins for GGobi

There are two potential styles of plugins. The most commonly used one involves supplying the R engine within GGobi a list of function objects that are invoked by GGobi when the plugin needs to react or provide information to GGobi. The functions in the collection typically share a common state that is mutable, i.e. can change across calls to these functions. This is achieved using R's closure mechanism.

The second style of plugin involves providing an S object to the R-GGobi engine and then the R-GGobi connection calls specific generic functions with that object as the primary argument. The idea is that the developer of the plugin will provide methods for the class of the plugin object and, in this way, provide different functionality for different plugin types. The methods can use either the S3 or newer S4-style class mechanism. While this is feasible, we will not pursue it in the near future as the first approach using closures is more general and is almost as simple.

Both input plugins and regular plugins can be implemented in this setup. In this way, we can use R to create run-time, analysis functionality and also new facilities for reading data to populate a GGobi instance.

One specifies a plugin using the usual XML *plugin* syntax in an initialization file for GGobi. For an R plugin, one specifies a language attribute with a value `"R"`. (One should declare the R (meta-) plugin itself before this.) In addition to the usual *description* and *author* information, one provides values for the init and create attributes of the *plugin* tag.

The init attribute gives the name of an R script that is *source()*'d by the R interpreter and is equivalent to the `onLoad()` routine of other plugins. This provides an opportunity to perform the necessary initializations for using this plugin class (not instance) such as loading libraries, defining functions and methods to create and use the plugin, etc.

The create attribute should be an S-language command that is (parsed and) evaluated and is expected to return the plugin object consisting of a collection of functions.

A simple example is given in the following XML stanza.

```
<plugin name="Rtest" source="plugins/R/R/testPlugin.S"
                     command="ggobiTestPlugin()"
                     language="R">
 <description>Test of the R plugin mechanism.</description>
```

```
  <author>Duncan Temple Lang</author>
  <dependencies>
     <dependency name="R"/>
  </dependencies>
</plugin>
```

When this plugin is loaded by GGobi, we source the file **plugins/R/R/testPlugin.S**. This defines a function named *ggobiTestPlugin()* and this is the function that is called to create the plugin instance.

The *ggobiTestPlugin()* function defines two functions within its body and returns them in a list. The first element is the *onClose()* function which is called when the plugin is no longer needed (i.e. when it is deactivated by the user or the GGobi instance disappears). The second function is *onUpdateDisplay()*, and is called by GGobi each time the display menu is recreated. Again, these functions are direct parallels of the C routines that are called for plugins implemented directly in C.

We do not have an *onCreate()* as we do in C-level plugins. The reason for this is that the command in the *create()* attribute performs the same role, and it is in this command that one does any initialization of the S object.

We will change this format and make the meta-plugin understand it. Done now. onCreate and onLoad.

Need to figure out whether they are expressions, file names or function names.

1.1 Plugins by Function Lists

This is not worth supporting. Almost every plugin will need to be able to store state somehow. We could pass the arguments for each call, but that is not as simple just using a closure.

1.2 Plugins by Explicit Methods

This is the second style of plugins that uses calls to generic global functions in S, using S-level dispatching on the class of the plugin object. This does not support mutability. This is not likely to be implemented unless people really want it and find a good reason why closures cannot be used!

2 Examples

Plotting examples.

Using Grid to dynamically modify plots in R based on brushing/identification/move events in ggobi.

R to annotate with model fits, etc.

Clustering, trees, etc. Dendograms with brushing in ggobi identifying leaves.

Using Gtk to build new GUI components. (Ideally, make ggobid, etc. GtkObject's with their own GtkTypes, etc. so we can use reflectance.)

3 Input Plugins

4 Examples

Database connectivity via the Postgres, MySQL, Oracle, ODBC packages.

Reading R objects and data from S3, SAS, Stata, formats using the foreign package.

5 R Plugins using R GGobi

One can also control GGobi directly from the S language using the `Rggobi` package. In this case, R is in control and GGobi is treated as a library of functions. However, running GGobi in this manner still allows for GGobi's plugins and even R plugins. Specifically, we can run R, start GGobi and GGobi can create plugins that are implemented in R. This nesting does not cause any conceptual problems.