The Quick Overview of the RGnumeric Package

Duncan Temple Lang

November 11, 2002

As the name suggests, this is the very brief overview of the `RGnumeric` package. The package allows one to call R functions from within Gnumeric spreadsheets. R acts as a plugin for Gnumeric and one can export individual R functions so that they behave like regular, built-in Gnumeric functions. This makes it easy to use existing functionality written in R directly in Gnumeric. This applies for both genral programming functionality, and also the extensive statistical methodology, random number generation and graphical facilities in R.

Most plugins simply provide a way to export functions from the plugin to Gnumeric. However, our interface allows R programmers to access Gnumeric objects from within R code and to both query and modify them. One can access individual cells, getting and setting their values. One can also change the formatting of individual cells, such as changing the color, font, alignment, data type (e.g. currency, percent, date, etc.), and border and interior pattern settings. Additionally, one can create and destroy other spreadsheets within workbooks, and even create new workbooks. Othe controls allow one to force recalculation of the workseet, clearing different aspects of cells. This provides a bidirectional connection between R and Gnumeric and allows one to program in whichever language is most appropriate for the task.

The basic plugin exports functions from R to Gnumeric and provides facilities within R to access Gnumeric objects. Three other plugins make up the R-Gnumeric connection and these consist of facilities for reading and writing Gnumeric sheets containing R functions and data, and a graphical interface for viewing and editing the exported R-Gnumeric functions.

Now that we have outlined the basic features, we will give a very brief overview of how to use them in their *current state*. (Things will change soon to smoothen out some of the details.)

1 Installing the Package

The first thing to do is to compile and install the package. To do this, you will need to have the source code for Gnumeric itself, and this should match the version of Gnumeric in which the plugin will be used. The plugin is installed as a regular R package, but one needs to specify where the gnumeric source distribution can be found and also where we can find the `gnomesupport.h` file. The simplest way to specify these is via the environment variables GNUMERIC_DIR and EXTRA_INCLUDES, and to specify the corresponding directories. Having done this, one can run the usual R package installation command

```
 R CMD INSTALL -c -l <wherever> RGnumeric.
```

where the term `wherever` is the directory into which the package should be installed and this is frequently omitted, defaulting to `\Env {R_HOME}/library/`.

The `--c` causes the plugin to be installed in a place that Gnumeric can find it. We do this by putting the file `plugin.xml` in a subdirectory of the user's home directory. Specifically, we put `plugin.xml` and the shared library in the directory `\Env {HOME}/.gnumeric/gnumeric-<version>/plugins/`. If one wants to specify a different location (such as a system wide installation directory for all users), one can set this via the environment variable INSTALL_DIR before running

And, finally, you can make Gnumeric load this plugin by default. Run `gnumeric` and select the *Tools* menu and within this, the *Plug-ins...* item. Scroll through the *Available inactive plugins* list (in the bottom left) and select the *R Plugin* entry. This act of selection should make the Activate plugin button on the right active. Click it to enable the plugin.

More details can (perhaps) be found in the `README.html` file in the distribution.

2 Registering Functions for Gnumeric

Now with the plugin enabled, the next thing to do is export functions so that they can be used in Gnumeric sheets. We register R functions in Gnumeric by calling the R function *gnumeric.registerFunction()*. Currently, this is best done when gnumeric is started, or more specifically when the R plugin is initialized. There are variety of ways to specify one or more files to *source()* when R is started. This can be done in your `.Rprofile` file in your home directory, or in the site startup script that is defined by the environment variable R_PROFILE or located in `$R_HOME/etc/Rprofile`. Putting the Gnumeric initialization code here will affect *all* R sessions, including those that do not involve Gnumeric. So, a more appropriate alternative is to use one or more files that are only read by the R-Gnumeric plugin. The plugin will look for three such files:

1. HOME`/.gnumeric/Rprofile`
2. HOME`/.gnumeric/<gnumeric-version>/plugins/R/Rprofile/`
3. the value of R_GNUMERIC_PROFILE

These can then be used to do Gnumeric-specific initialization such as registering functions, etc.

At this point, it is simplest to take a look at an example, specifically the file `example` in the `examples/` directory of the installed library. To run these within Gnumeric, set the environment variable R_GNUMERIC_PROFILE to the fully qualified name of the file `examples/bernoulli.S`.

```
 setenv R_GNUMERIC_PROFILE $R_HOME/library/RGnumeric/examples/bernoulli.S
```

Now, start `gnumeric` and open the file `example`.

```
 gnumeric $R_HOME/library/RGnumeric/examples/example
```

This example illustrates how we can call functions such as *rpois()*, *rbernoulli()* and evaluate R expressions. It also illustrates how we can use R graphics devices within gnumeric to create displays in separate windows.

We define the Gnumeric functions *rpois()*, *rbernoulli()* and *reval()* in the file `bernoullis.S` using the S function *gnumeric.registerFunction()*. These definitions look like

```
gnumeric.registerFunction(
       function(lambda) {rpois(1,lambda)},
       "rpois",
       "f" , "mean",
       "return a value sampled from a Poisson random variable.")
```

This generates a single observation from a Poission random variable with mean `lambda`. We can can call these functions in a Gnumeric sheet by specifying the value in cell such as

```
 =rpois(10)
```

The random value returned from this call will be displayed in the cell. Recalculating the sheet (the F9 key) will generate a new random value.

The basics of the function *gnumeric.registerFunction()* are relatively straightforward. One specifies an S function object that will be called with arguments provided by Gnumeric. This function object contains the parameters it expects and the body which is evaluated. The second argument to *gnumeric.registerFunction()* is the name by which Gnumeric sees this function and it can be used within the sheet. The third argument controls how the arguments are passed to the function. This is typically a single string that lists the type of each of the arguments. The types are given as characters from the following table.

f	A floating point value.
s	A string.
b	A boolean.
r	A Range, e. g. A1:A5 - See 'A'
a	An Array e. g. 1,2,3;4,5,6 (a 3x2 array) - See 'A'
A	Either an Array or a Range.
?	Any type.
|	This designates that the arguments in the token string following this character are optional.

For example, if we register a function that expects a string and real number as its arguments in that order, the third argument to *gnumeric.registerFunction()* is

```
"sf"
```

The other arguments to *gnumeric.registerFunction()* are used by Gnumeric to offer help to the user. These are the names of the arguments and a general help string describing the function.

The graphics device in the `example` sheet was created by evaluating an R expression given as a string. In the sheet, the cell $(3, D)$ is specified as

```
=reval("x <- rnorm(100); par(mfrow=c(2,1)); plot(x); hist(x); mean(x)")
```

This generates a random vector of normals and displays them in a scatter plot and histogram. Each time the sheet is recalculated, a new vector of random variables is generated and replotted.

2.1 Return Values

Typically, the S functions that are exported will return scalar values of the basic types: integer, numeric, logical and character. These are the inserted directly into the cell from which the function was called. The conversion from the S type to the corresponding Gnumeric type happens automatically for the user.

In the future, one will be able to return more complex objects such as lists and objects with classes. These are useful not for their representation in a single cell, but since they can be used as arguments to other functions. This allows one to avoid global variables in the S functions but to store them locally in a gnumeric cell. Then these values are available in other calls.

This basically describes how scalar values are passed to and from S functions. One can use this setup to call R functions from Gnumeric to pass 'constant' values and insert the results into Gnumeric cells.

2.2 Accessing the Sheet

If the S function being invoked from Gnumeric needs access to the sheet or the Gnumeric workbook or any other Gnumeric internals, it will need to get a reference to the sheet object when it is called. For this to happen, the function should have an argument named *.sheet.* For simplicity, this should be its last argument. In this case, the glue between R and Gnumeric will supply an opaque value of class *GnumericSheetRef* for this argument. This can then be used in calls to other R functions to access information about the sheet, get a reference to the containing workbook (via *getGnumericWorkbook()*). The *.sheet* argument is typically the last argument to the S function.

In addition to the *.sheet* argument, it is sometimes useful to know the identity of the cell associated with the invocation of the S function. This is passed in the call from Gnumeric if and only if the function has an argument named *.cell.* The value is a simple integer vector containing the row and column index.

2.3 Cell Ranges as Arguments

In some cases, one will want the function to get the identity of a cell rather than the cell value itself. In such cases, one should specify the argument as being of type *Range* via the argument type identifier `r`. This type of argument is passed to the function as an object of class *GnumericCellRange* and consists of two elements named *start* and *end.* These specify the top-left and bottom-right cells defining the box of interest. When this is a single cell, the *start* and *end* elements are the same. Each of these elements is a named integer vector of length two giving the row and column. These currently use 0 based counting.

An example of when we might pass a cell range rather than value is when we want the function to output auxillary values into that region rather than simply return a single value that goes into the cell from which the function was called. For example, suppose we have a function that displays the coefficients of a previousl fitted linear model into a range of cells in the sheet. The function might be defined as follows:

```
function(cell, .sheet) {
 .sheet[cell$start["row"], cell$start["col"]] <- "my value"
}
```

This uses just the *start* element of the *GnumericCellRange* and extracts the row and column values. These are used to specify to which cell the value `"my value"` is assigned.

3 Accessing Gnumeric From R

We have seen in the example that assigned a value to a particular cell in the sheet that the R functions can modify the sheet directly. This can be done via the regular *<-()* operator for the class *GnumericSheetRef*. Similarly, one can get values from different cells via the *[()* operator. I have attempted to implement the usual matrix-like assignment operations. (When these do not work please let me know.) There are obvious differences in that sheets have "infinite" dimension even if there is no data. Thus, accessing an entire column should probably not return 65535 observations. Instead, it should return from row 0 the row that defines the extent of the sheet. The extent is the maximum used dimension. And this is what is done. Similarly, when assigning to a collection of cells, one would like to be able to specify the starting cell rather than having to give the entire range of indices. This is another place we differ from matix assignment. We allow one to specify the starting cell as the left hand side of an assignment of a vector. The multiple values on the right hand side are inserted into the adjacent cells from this starting cell, inclusively. One can indicate whether the adjacent cells are determined row- or column-wise using the *byrow* argument. Some examples of these are given here and in the different example files (`ExampleProfile` and `bernoulli.S`).

```
.sheet[1,2] # get the cell in the 1st row and 2nd column
.sheet[1,]  # get the first row
.sheet[,1]  # get the first column

.sheet[1,2] <- 1.0     # set the cell in the 1st row and 2nd column
.sheet[1,]  <- letters # set the first 26 cells in the first row
.sheet[1,10]  <- letters # set the 26 cells in the first row starting
                         # at 10
.sheet[1,10, byrow=F]  <- letters # set the first 26 cells in column 10
```

In the future, we should support using column names rather than numbers.

3.1 Other Functions for Sheets

One can also query the structure of a sheet. For instance, one can get the number of rows and columns – i.e. the dimensions – that the sheet occupies via the function *dim.GnumericSheetRef()*. This is akin to a bounding box as it only reports on the number of rows and columns that are occupied, and not the maximum row and column that is referenced. In other words, it ignores entirely empty rows and empty columns. One can compute the extent using *sheetExtent()* or specifying the *collapse* argument as F in the call to *dim.GnumericSheetRef()*.

One can clear all or certain cell regions within sheet using *clearSheet()*. This allows one to specify what attributes of each cell should be cleared.

3.2 Cell Formats

It is possible to query and set the appearance of a cell. There are two approaches to this. One involves getting or setting the entire formatting information for a cell. This includes the color, font, border, patter, alignment. The function *getCellFormat()* retrieves a list with the current settings for each of these settings.

Note that currently setting any attribute of a cell applies to all cells in that column. This will be fixed in the future.

3.3 The Workbook

From the *.sheet* argument, one can get a reference to the containing workbook via the function *getGnumericWorkbook()*. The *length()* and *names()* method for this class (i.e. appended with the *.GnumericWorkbookRef*) give the number and names of the sheets in a workbook, respectively. We have defined the *[()* operator for the class *GnumericWorkbookRef* so that one can easily access different sheets in the workbook either by index or name. Additionally, one can also create a new workbook using the function *newGnumericWorkbook()*. This can create an empty workbook or load a previously saved one from a file/URL.

The utility functions *uniqueSheetName()* and *recalcBook()* operate on *GnumericWorkbookRef* objects. The first of these returns a unique name for a new sheet that will not conflict with others. It allows the caller to suggest a name and it returns a slightly modified version that avoids duplicating an existing name. The *recalcBook()*, as its name suggests, forces a recalculation of all the sheets in the workbook and all of their cells.

4 Saving Workbooks with R functions

When one develops a Gnumeric sheet that contains R functions, it is important to be able to reload this and find the R functions. Whether it be the original author reloading the sheet or a different reader to whom the sheet has been sent, we want the saved file to be self contained. We must assume that each reader has access to the plugin. Given that, we want the Gnumeric functions that call R functions to be defined within the workbook file itself. We do this by storing the definitions of the gnumeric and R functions within the file itself. When one saves a sheet containing R-Gnumeric functions, it is *essential* that one selects the appropriate format and not the default compressed XML format. The basic R-Gnumeric plugin comes with 2 other plugins: one for writing files containing R-Gnumeric functions and one for reading these files. The extension for this type of file is .rng. To save a file in this format, select the Save As in the *File* menu. From the choice menu for the *File format*, select the entry *Gnumeric XML file with R functions (*.rgn)*.

All files with extension the extension .rgn will be read into Gnumeric using the special file reader provided by the plugin. This reads and registers the functions in the file. Additionally, it extracts the R search list settings from the file and attaches the packages that were in effect when the file was saved.

The .rgn files are simply extended versions of the standard gnumeric format. These are compressed XML files. We add a new namespace declaration (R) to the top of the file and then add *functions* and *searchPath* sections. One can manually edit these files by uncompressing them and modifyin the XML.

Saving the functions is relatively simple. We store them in text form in CDATA blocks of XML to avoid escaping. For top-level functions whose environment is the regular global environment, dumping the function as text is sufficient to be able to recreate it. For functions that have a different environment, we must ensure that we restore that environment and its contents. We do this by dumping each of these environments in XML format and putting a reference to this environment in each of the functions that share the environment.

Currently, we do not store the contents of the global environment (i.e. the work area). This can be easily arranged if it proves to be useful. In the future, we would like to use the `RSXMLObjects` package for serializing the S objects directly to XML. This would provide greater flexiblity in processing these files with non-R tools.

One can also specify an R script or code in the file. One can manuall edit the .rgn file and add a *R:init* tag whose content is R source code. This is usually enclosed within a *CDATA* construct to escape the special characters such as <, ... When the file is read, this will be evaluated after the environments have been created and the functions have been registered, but just before the workbook cells have been created. If greater resolution is needed to control when this script is run, we can easily arrange this, using other tags or attributes. Unfortunately, one cannot specify this code except for manually at present. We could perhaps store a *.First()* function in the global environment and arrange to call this when we load a sheet.

This introduces yet another location where one can initialize the R session or at least run R code. To recap, the possible ways to do this are

.rgn file In the *R:init* tag of the self-contained file.

`R_GNUMERIC_PROFILE` A script specified by the environment variable `R_GNUMERIC_PROFILE`.

Plugin-specific init file A file named `R` in the file `~/.gnumeric/<version number>/plugins/` directory.

Plugin Init file A version-independent plugin initialization file named `Rprofile` in `~/.gnumeric/`

5 Defining and Editing Functions

While one is creating a worksheet that calls R functions, one has to have access to the appropriate Gnumeric functions that call the R functions. In other words, one has to define these functions before using them. Additionally, we don't necessarily want to define them within a cell of the sheet (as this will cause it to be redefined each time the sheet is recalculated). There are two basic ways to define Gnumeric functions that call R functions. The first is to define them in an external file containing R code. Then one can have these automatically read when the R-Gnumeric plugin is started Alternatively, one can explicitly load these into an already running Gnumeric session using the R *source()* function. This will however require making an explicit call to *source()* within a cell in the sheet.

The second mechanism for defining functions within an existing Gnumeric session is to define and edit them in a separate graphical interface. The `RGnumeric` package provides a Gnumeric function *edit()* that can be called from within a cell in a sheet. When this is invoked as

```
=edit()
```

a separate window is created. This lists all of the existing R-Gnumeric functions and allows one to edit these and/or create new ones. On the right of this window, there fields for specifying the name of the Gnumeric function and its body and argument list. Having created this window, one can immediately delete the call to *edit()*. The window will remain in existence and can be used at any moment in the Gnumeric session. This allows one to define and redefine functions. When an .rgn file is stored, it will contain the definitions of any functions defined at the time it is saved.

A better way to do this would be to have a menu item in the worksheet interface that brings up the editor. Unfortunately, the Gnumeric plugin design does support creating a plugin instance for each workbook. Therefore we cannot add to each workbook's interface. See the ggobi plugin for how this is done more flexibly.

6 Unfinished Aspects

This package was put together reasonably quickly and is still far from finished.

references I will add references so that one can have R objects returned into cells.

documentation I will add regular help files for the functions to the package. The primary reason for them not being there is that I am using the XML format that we are developing concurrently.

CellRef attributes I need to finish supporting the getting and setting of the formatting attributes for the *CellRef* class. This is relatively trivial since we have the hard work done for the general formatting and just need to provide wrappers for these attributes or provide separate C routines.

object formatting libraries It would be nice to have a collection of functions for formatting S objects as boxes in a spreadsheet. An example of the display of an *lm.summary* object in a a spreadsheet is included in the examples in `ExampleProfile` via the Gnumeric function *lmResults*.

0 or 1-based counting We can smooth out the different cases in which we use 0 and 1-based referencing of arrays, cells, etc.

Data frames We can return data frames or matrices for subsets of the form

```
.sheet[,]
.sheet[1:r,1:c]
```

Unfortunately, the types of the cells may be non-homogeneous and so converting to a matrix may loose a lot of information. Similarly, columns in a data frame may not work for this. Hence, lists of lists may be the best default type.

7 Other Approaches

An entirely different way to connect Gnumeric and R is to use CORBA. Gnumeric provides a CORBA interface which allows one to access different functionality of the workbook and sheets from a remote application. The `CORBA package` for R can dynamically determine the available methods provided by this interface and allow R users to invoke them from within R. The embedding is more direct, slightly less complex and provides a richer functionality than the CORBA interface.