R and Calc Integration

User's Guide

v.0.1.12

Table of Contents

Please note that this manual is current as of February 23, 2008.

Introduction

RCalc is Java-based add-on for OpenOffice.org versions 2.1 and above. It allows you to send data from a Calc spreadsheet to R and write the outputs directly into the spreadsheet. In addition to a basic coding window where one can write R scripts, RCalc also comes with built-in Calc functions that allow basic R commands to be passed directly from cells, and also a GUI scripting tool to create user interfaces for automating statistical processes.

This package was developed with the support of Google, through its Summer of Code program in 2007.

Installation

Installing the package can be done in one of two ways, either through downloading the compiled package and using `unopkg`, or compiling the entire program from its source code. In both cases, however, the following software is needed:

- OpenOffice.org 2.1 or higher
- R
- Rserve 0.4.7 – note that only this version is supported.

Additionally, those interested in compiling the software need to have the OpenOffice.org SDK installed.

Installing Using the Extension Manager

The easiest way to get the add-on running in Calc is by going to `Tools > Extension`

Manager. Once in the manager window, simply click on Add and find the RAddon.uno.pkg file you downloaded and click Open. Once the package is installed, restarted OpenOffice and a new menu button should appear at the top of the Calc window.

Installing the Package Manually

To install the package alone, you need to download the RAddon.uno.pkg file from the R/Calc wiki, or a mirror thereof. The latest version of the program is always available at:

```
http://wiki.services.openoffice.org/wiki/R_and_Calc
```

Once this has been downloaded, go into the Command Prompt in Windows or the Terminal in Linux and change the directory to <OOHOME>/program where <OOHOME> is your Openoffice.org home directory. Once there, type:

```
unopkg add <RADDON PATH>/RAddon.uno.pkg
```

where <RADDDON PATH> is the path to the RAddon.uno.pkg file you downloaded.

The add-on depends on on Rserve, which can be installed in R. Briefly, here are the three commands in R that load and activate Rserve:

- install.packages("Rserve")
- library(Rserve)
- Rserve()

More information is available at http://cran.r-project.org/ and http://rosuda.org/Rserve/.

Recommended R Packages

In addition to the R base package, the built-in GUI functions that come with the add-on use packages that may need to be installed in addition to these packages. To experience everything the add-on has to offer, the following packages should be installed:

- odesolve
- scatterplot3d
- sna
- survival

Using the Add-on

The add-on itself is built to allow one to use R's statistical functions without prior knowledge of the scripting language itself. As such, it comes with a large amount of dialog-based functions through which one can input information and simply receive outputs. This section is meant to illustrate how one can start using the add-on, and pass information to R using the dialog-based tools.

Starting RServe

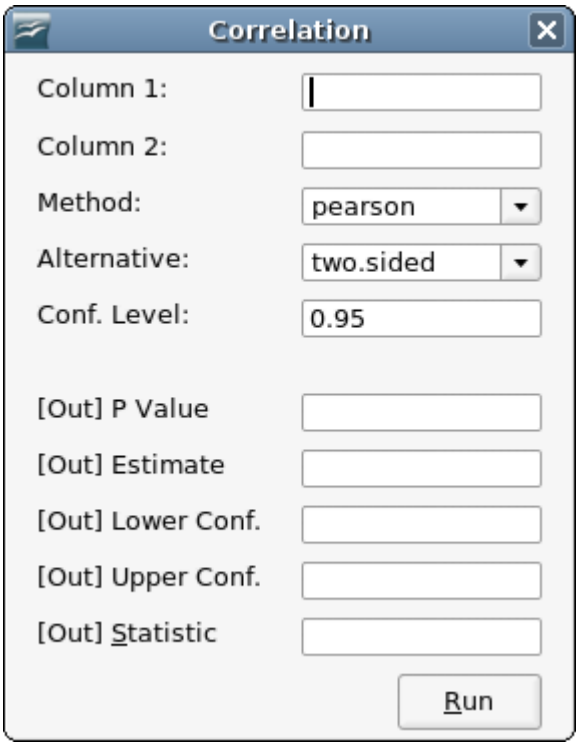
At the time of writing, the add-on is unable to launch R and RServe from within OpenOffice, so the two must be launched manually. Assuming they have been installed properly, one can do this by launching R and running the following code:

```
library(Rserve)
Rserve()
```

This will launch the server, and the add-on should automatically detect the server when R calls are made.

Using the Dialog Windows

A fairly complex example for a dialog window is given below, which can be seen by clicking on the `Correlation` menu item under the add-on menu.



In the add-on, all of the input variables required are placed at the top of the dialog, while information sent to the spreadsheet is preceded by the `[Out]` prefix. Thus, the dialog window above accepts two different columns of data ,the correlation test and type, as well as a confidence interval value. The last three options have default values already entered when the dialog window starts.

One can send data to R using these dialog windows in a similar manner to other forms in Calc. By writing the addresses of the various cells, the add-on will send the appropriate data to R. For

the sake of this example, support we have a spreadsheet similar to the cropped image below.

	A	B
1	-1.45	-1.55
2	-0.53	-0.45
3	-0.5	-0.47
4	-0.36	-0.54
5	-0.31	-0.29
6	-0.08	-0.17
7	0.44	0.23
8	0.55	0.3
9	0.66	0.64
10	1.27	0.81
11	1.33	1.24
12	1.38	1.34
13	1.44	1.3
14	1.58	1.36
15	1.67	1.84
16		

If we wanted to pass this information to R and run a correlation test to obtain the confidence interval, correlation estimate, and the P value, we would first type `A1:A15` and `B1:B15` into `Column 1` and `Column 2` form fields, respectively. Supposing we want the outputs to be sent to column D, we would put `D1` for P value, `D2` for the estimate, `D3` for Lower Conf., and `D4` for Upper Conf. Clicking `Run` will then calculate the various values and send them to the spreadsheet.

Notes on Data Manipulation

It is important to note that when dealing with missing values, R uses the `NA` keyword to denote a missing value. In the add-on, a similar process should be adopted, though empty cells are also treated as missing values. Other text-based cell values, such as a period or asterisk, are sent without any modification to R. This can cause errors when analyzing data, and care should be taken to avoid such problems.

Coding Window

The coding window is available in the R Add-on menu, by going to `Advanced > Coding Window`. The window allows one to write or load R scripts, directly reference data from Calc

spreadsheets, and send output back to the sheets themselves.

Coding in the Window is similar to writing scripts in R, with the main addition being reading data from the spreadsheet and sending output back. To read data from the spreadsheet, one simply needs to reference the necessary cells surrounded with curly brackets and the pound sign, like so:

```
{#A1}
```

or

```
{#A1:C10}
```

The add-on will convert these values to the necessary commands when code is being executed.

Sending output is similar, though requires knowledge of the R object structure, which is described in another section of this manual. The code for outputs is based on assigning values to variables in R. Thus, it follows the following structure:

```
[Target Cell] <- [Value]
```

In this case, the target cell needs to be marked with `$OUT` to denote it is used for outputs, like so:

```
{$OUT#A1}
```

The coding window takes two types of line of code: R scripts and output commands. It is not possible to mix commands with output. As such, if you would like to output the variance of the numbers, 1, 2, and 3 into cell `A1`, one line of code needs to find this value and the second line is the output command. If the R script returns a real number, using the `{$BASE}` keyword will allow you

to print the resulting value in a cell. Thus, you would write:

```
var(c(1,2,3))
{$OUT#A1}<-{$BASE}
```

One can denote matrices by appending `[x]` to the variable name or keyword, where `x` is the index of the array element. For example, if we generate 10 random numbers and want to put the third number into cell `C3`, the following code would be used:

```
rnorm(10)
{$OUT#C3}<-{$BASE[3]}
```

As mentioned in the discussion on the R object structure, complex objects actually have properties or values that have names in them. In these cases, one would not use the `$BASE` keyword but instead the name of the actual property. A typical example is the Pearson correlation test in R, which is accessed through the `cor.test()` function. The object structure for output from this function is complex because it contains the correlation (called `estimate`), the P value (called `p.value`), a two-element array for the confidence interval (called `conf.int`), and other information. In this case, we would use the property names to print the output to a cell, like so:

```
cor.test(c(1,2,3,4), c(1,1,2,3))
{$OUT#A1}<-estimate
{$OUT#A2}<-p.value
```

Furthermore, if you can access such values using the `$BASE` key word by calling them like you would in R. For example:

```
correlObj <- cor.test(c(1,2,3,4), c(1,1,2,3))
correlObj$estimate
{$OUT#C1}<-{$BASE}
correlObj$p.value
{$OUT#C2}<-{$BASE}
```

The code above will have the same results as the earlier example.

For the confidence interval, we would use the array indices:

```
{$OUT#A3}<-conf.int[0]
{$OUT#A4}<-conf.int[1]
```

If you try to print a property with a name that does not exist, the add-on will simply print the actual name in the cell. This is useful to note because if you would like to print text (for example, to label output), simply write it as an assignment. Finally, lines can be commented out using the `//` sign at the start of the line.

With this in mind, the script below takes two columns, runs the correlation test, and then prints the output to a final column.

```
// R Script.
cor.test({#A1:A10},{#B1:B10})

// Labels.
{$OUT#C1}<-Correlation Estimate
{$OUT#C2}<-P Value
{$OUT#C3}<-Confidence Interval (Lower)
{$OUT#C4}<-Confidence Interval (Upper)

// Values.
{$OUT#D1}<-estimate
{$OUT#D2}<-p.value
{$OUT#D3}<-conf.int[0]
{$OUT#D4}<-conf.int[1]
```

It is also possible to add matrix output to a spreadsheet. In this case, you need to know the exact dimensions of the matrix being sent to the spreadsheet, and would write the coordinates by writing `[start cell]:[end cell]` in the formats listed above. For example, we know that `conf.int` is an array with two members: a lower bound and an upper bound. As such, we can rewrite the last two lines of the code above to the following:

```
{$OUT#D3:D4}<-conf.int
```

Avoiding Parsing

If for some reason the script you are writing should not be parsed by the add-on before being sent to R, you can start the line by writing `LIT:` to avoid parsing. For example, this is useful when writing code that includes `{` and `}`, because this syntax is normally parsed by the add-on.

Using `LIT:` allows you to define your own R functions and use loops and conditional statements in the scripting window. The only catch is they all need to be on one line, and no outputs are analyzed following a `LIT:` statement. As such, one would need to use R-based variables to run calculations, and then call them on a new line before sending them as output to a cell.

For example, suppose you wanted to make a linear function that simply takes an input `x` and returns `mx+b` where m and b are pre-defined constants. The code below would define such a function in the scripting window, call the function, and send the output to cell `A1`.

```
LIT: linFunc <- function(x) { m = 3; b = 2; return(m*x + b); }
linFunc(3.7)
{$OUT#A1}<-{$BASE}
```

Writing GUI Files

The graphical windows available within the add-on use the same scripting language as the coding window for the actual communication with R. The main difference is they also come with a layout component, where one can add specific types of graphical interfaces to allow other users to easily use the add-on, without actually needing to know how to use R itself.

These files are organized into three sections:

1. `__ETC:` This section is a section for comments, and is ignored when loaded into the add-on.

2. `__GUI:` This is where the user interface code is placed.

3. `__CALL:` The actual script is included here, and is used in much the same way as the coding window.

Scripting the GUI

Every line in this section follows the same pattern:

```
[Object Name] <- [Object Type](
            [x position], [y position],
            [width], [height],
            [additional information])
```

In this case, the `[Object Name]` is simply a way to represent the object's value in the scripting section of the code. The `[Object Type]` is the actual name of the object type, and is similar to a function call. The parameters of the call start with the `x` and `y` position of the object, with the origin being at the top left corner of the dialog window. The height and width represent the size of the object in pixels. The `[additional information]` section allows one to pass additional information, such as the title of the dialog window, text within the label or button, or similar values. The table below lists all of the available objects within the add-on, along with what `[additional information]` actually does.

Object Type	**Description**	**What [add. info.] does.**
`Dialog`	The dialog frame.	Changes the title of the window.
`Label`	A basic label.	The text within the label.
`RunButton`	Button to launch the	The text in the button.

	script.	
`CancelButton`	Button to close the window without executing any scripts.	The text in the button.
`TextField`	Accepts text inputs.	The default value.
`NumberField`	Accepts an address for one cell in the spreadsheet.	The default value.
`ArrayField`	Accepts the address of multiple cells in a spreadsheet.	The default value.
`ComboBox`	A combo box with various options.	A list of options, delimited by a semi-colon (";"). The first item in the list is the default.
`OutputField`	Accepts the address for outputting a specific value.	The default value.

The only requirement in the `GUI` section is that there needs to be a `Dialog` object instantiated, and that it must be the first object in the list.

R Scripting

The last section of the file contains information on both code execution and outputting results into the spreadsheet. At its most basic, the code works just like the coding window scripts. The only difference is that fields and combo boxes may be called for information – this feature is not available in the coding window.

To reference an input field, such as a `NumberField` or `ArrayField` object, simply write `{$NAME}` where `NAME` is the name of the object. For example, if we instantiate a number field using

the code below,

```
NUMF<-NumberField(10,10,100,12)
```

then we can reference the value in the field by writing,

```
{$NUMF}
```

Output fields are referenced the same way. The only difference is that to actually send the value to the proper cell, the word `$OUT` must precede `$NAME`. Thus, if we instantiate an `OutputField` with the code below,

```
OFIE<-OutputField(10,10,100,12)
```

then we can send the text `Hello World` to the cell whose address is in the field with the following code,

```
{$OUT$OFIE}<-Hello World
```

If you are interested in using the curly braces, `{` and `}`, within the R code here, you need to use special representations of them. Otherwise, the add-on will think they are surrounding a variable name and an error will occur. The special representations are both variables that point to the braces. So for `{`, use `{$CURLL}` and for `}`, use `{$CURLR}`.

Data Visualization

The add-in supports visualization of data in two ways. First, if one simply writes code that outputs a visual result (for example, `plot(rnorm(25), rnorm(25))`, then R will automatically create a new window to display the output. However, this method is not recommended because the

window will not update its image unless "Run" is clicked again.

The add-on also features a built-in tool that allows R graphics output to be embedded as an image in the actual spreadsheet. Doing so allows one to make plots and data visualizations, save the sheet, and view the output later without having to use R a second time. To insert an image into the sheet based on a plot, simply write `PLOT:` on the line whose graphical output should be inserted into the sheet. This will create a JPG file and embed it within the document. `PLOT_PNG:` will do the same with a PNG file rather than a a JPG, and `PLOT_EPS:` will use an EPS file.. An example is shown below:

```
PLOT: plot(rnorm(25), rnorm(25))
```

Images can also be imported manually. If you save an image through R, writing `IMPORT_ABS:` followed by the file location as a URL (for example, `file:///C:/Windows/Temp/myPicture.jpg`) will import the image. Thus, one can manually write scripts that embed images in the spreadsheet. The code below shows an example using a plot command.

```
// First, we create a new graphics device.
JPG(filename="/home/user/Desktop/myimage.jpg")

// Plotting random numbers.
plot(rnorm(25), rnorm(25))

// Shutting the device off.
dev.off()

// Embedding the image.
IMPORT_ABS: file:///home/user/Desktop/myimage.jpg
```

These graphics commands can be used within both the GUI scripting tool, and the Coding Window.

Using RDUMP for Object Structures

`RDUMP` is a basic tool that prints the entire object structure returned by an R script. It does not have support for inputting data from spreadsheets, as its goal is simply to shed light on the R structure of specific calls made to R. This allows one to see the names of properties available for a specific call, and whether or not these properties are arrays.

To use the tool, simply go to the add-on menu and click `Advanced > Dump R Output`. The tool is simple in that it accepts only one line of code. Type the code below:

```
cor.test(c(1,2,3), c(1,2,10))
```

This will create a new spreadsheet, with the actual code on the first row and subsequent rows containing property names and values. The specific output for the code above should look similar to this:

__CALLED:	cor.test(c(1,2,3), c(1,2,5))
statistic	3.46
parameter	1
p.value	0.18
estimate	0.96
null.value	0
alternative	two.sided

Within this tool, property names appear on the first column and values on the second column. Array values are printed on multiple lines.

If a property has no name but is a numeric value, then the name `numeric` is given to the property. Otherwise, unnamed non-numeric properties simply have a `.` as the name. For example, the output for `rnorm(10)` is shown below, which returns a numeric array of 10 values.

__CALLED:	rnorm(10)
numeric	0.81
numeric	-0.3
numeric	0.07
numeric	-1.08
numeric	0.04
numeric	-0.29
numeric	-1.06

The example below calls `(.packages())`, which is an array of text values.

__CALLED:	(.packages())
.	stats
.	graphics
.	grDevices
.	utils
.	datasets

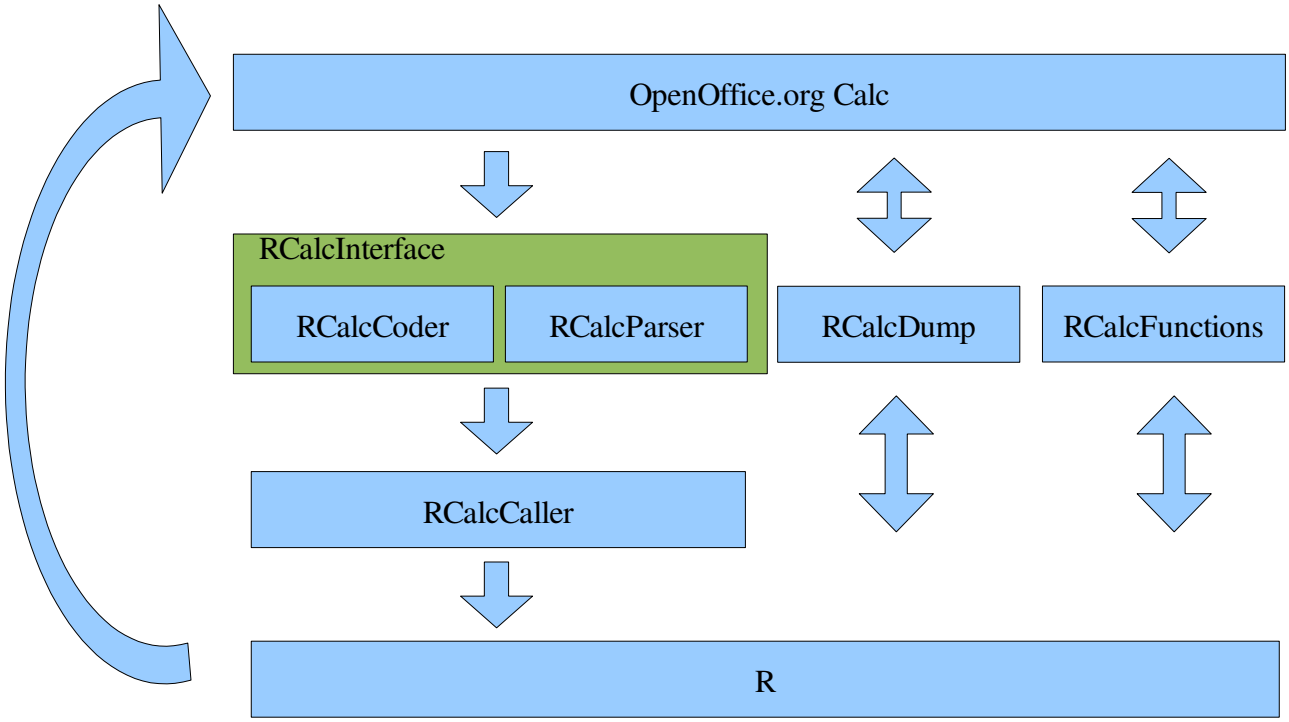
Add-On Architecture

The add-on is organized in a way that allows one to easily upgrade the actual R interface without making major changes to the add-on itself. Specifically, this is done through the use of the `RCalcCaller` class, which works as the main bridge between various classes in OpenOffice and R itself. The coding window and GUI tools both implement the `RCalcInterface` interface, which is used by `RCalcCaller` to organize inputs to R and send outputs back to spreadsheets.

`RCalcDump` is a unique feature of the add-on because it does not implement `RCalcInterface` but communicates with R directly. This is because the software actually scans the entire R object structure that is returned to it, and as such does not work in the same way as other parts of the add-on. Functions themselves also work in a different manner, as they do not allow for specific object

properties or values to be returned to a cell. If one is willing to implement this, it is recommended that the engine start implementing the `RCalcInterface`.

For clarity, a basic diagram of the various classes is drawn below.



Appendix 1: Commonly Used Properties of Various R Calls

The list below is meant as a reference for people interested in coding or programming in the R/Calc add-on. It provides the names of some properties available for the calls, and can help one become more familiar with scripting in the add-on.

Numeric Calls

Any R script that simply returns a number, such as `var()`, `cor()`, `rnorm(1)`, and so can be sent to a cell using the `{$BASE}` keyword.

Similarly, numeric arrays can be accessed with `{$BASE[x]}` where `x` is the index of the array member.

cor.test()

Property Name	**Type**	**Description**
`estimate`	number	The correlation.
`p.value`	number	The P value of the estimate.
`null.value`	number	The null hypothesis.
`alternative`	text	The type of test (e.g. “two-sided”).
`method`	text	Type of correlation (e.g. “Pearson”).
`conf.int`	numeric array (2 members)	An array with the confidence interval. The member at index 0 is the lower bound, while 1 has the upper bound.