Dynamic LaTeX reports with RSP

Henrik Bengtsson

April 19, 2011

Abstract

An important part of a statistical analysis is to document the analysis and its results. A common approach is to build up an R script as the the analysis progresses. This script may generate image files and tables that are later inserted manually into a, say, LaTeX report. This strategy works alright for small one-off analyzes, whereas for larger and partly repetitive analyzes an automatic report generator is more suitable.

In this document we will illustrate how a LaTeX document can be extended with the RSP markup language resulting in a very powerful tool for generating dynamic reports in R. As we will discover, with RSP it is possible to generate document constructs that are not possible in Sweave, e.g. looping of a mix of R and LaTeX blocks. Because RSP is a so called *context-indepent* markup language, RSP can be used to produce documents of any text-based format, e.g. plain text, HTML, XML, SVG, as well as Javascript, R, and Sweave. This document was produced using RSP-embedded LaTeX.

Keywords: reproducible research, report generator, markup language, LaTeX

This document is under construction.

Contents

1 Compiling RSP-markup documents **3**

2 The RSP markup language **3**
2.1 Evaluating code (<%{code}%>) . . . 3
2.2 Evaluating and embedding code (<%:{code}%>) . . . 4
2.3 Echoing evaluated code (<%=evalWithEcho({code chunk})%>) . . . 4
2.4 Inlining values of variables (<%={code}%>) . . . 4
2.5 Iterating over a mixture of RSP and text blocks . . . 5

3 Generating and inserting figures **5**
3.1 Brief on including image files in LaTeX . . . 6
3.2 Defining a function that creates an image file . . . 6
3.3 Generating and embedding figures . . . 6

4 Templates - Reusing RSP and text blocks **7**

5 Preprocessing directives **8**
5.1 Hidden (non-nested) comments (<%--{anything}--%>) . . . 8

1 Compiling RSP-markup documents

In order to utilize the methods explained in this document, the *R.rsp* package[1] needs to be loaded, i.e. `library("R.rsp")`. As a first example, consider the following RSP-embedded R string

```
"A random number in [1,100]: <%=sample(1:100, size=1)%>\n"
```

It can be compiled as

```
> rsp(text="A random number in [1,100]: <%=sample(1:100, size=1)%>\n")
A random number in [1,100]: 77
```

Any document with RSP markup can be compiled using the `rsp()` method. For instance, consider an RSP-embedded text file `README.txt.rsp`. In order to compile it into a plain text file, do

```
rsp("README.txt.rsp")
```

This will (i) translate the RSP text file into a valid R script (`README.txt.rsp.R`), and (ii) run the R script resulting in a plain text file (`README.txt`). Other document formats can be compiled in a similar way, e.g. `rsp("index.html.rsp")` and `rsp("report.Rnw.rsp")`.

For certain document formats recognized by `rsp()`, further processing will also be done. For instance, consider a RSP-embedded LaTeX document `report.tex.rsp`. When compiling it by

```
rsp("report.tex.rsp")
```

the `rsp()` method will as before (i) translate the LaTeX RSP document into a valid R script (`report.tex.rsp.R`), and (ii) run the R script resulting in a LaTeX document (`report.tex`), but it will also (iii) compile the LaTeX document into a PDF (`report.pdf`). To try this yourself, compile this very document by calling

```
library("R.rsp")
path <- system.file("doc", package="R.rsp")
rsp("report.tex.rsp", path=path)
```

The PDF (`report.pdf`) will be available in the current directory of R (see `getwd()`).

2 The RSP markup language

2.1 Evaluating code (<%{code}%>)

The RSP markup `<%{code}%>` evaluates the code (without inserting it into the document). For instance,

```
<%
n <- 3
type <- "horse"
%>
```

evaluates the code such that `n == 3` and `type == "horse"` afterward.

[1] To install the *R.rsp* package, call `install.packages("R.rsp")` at the R prompt.

2.2 Evaluating and embedding code (<%:{code}%>)

Just as <%{code}%>, the RSP markup <%:{code}%> also evaluates code, but in addition it also inserts the code verbatim into the document. For instance,

```
<%:
n <- 3
type <- "horse"
%>
```

evaluates the code and insert the following into the output document:

```
n <- 3
type <- "horse"
```

Formatting of the inserted code has to be taken care of by LaTeX. For instance, here we have explicitly wrapped the RSP markup inside a \begin{verbatim}...\end{verbatim} block.

2.3 Echoing evaluated code (<%=evalWithEcho({code chunk})%>)

The evalWithEcho() function allows us to evaluate, embed and echo the output of a code chunk[2]. For example,

```
<%=evalWithEcho({
n <- 3;   # Comments are not displayed
n
print(Sys.time())
type <- "horse"
type
})%>
```

produces

```
> n <- 3
> n
[1] 3
> print(Sys.time())
[1] "2011-05-04 22:39:34 CEST"
> type <- "horse"
> type
[1] "horse"
```

Note that code is parsed and formatted by the R parser, meaning that indentation, spacing and so on are *not* preserved when echoing this way. This is also why comments, semicolons and other code constructs are dropped from the code.

2.4 Inlining values of variables (<%={code}%>)

The RSP markup <%={code}%> evaluates the code (without inserting it into the document) and inserts the character representation[3] of the returned object. For instance,

[2] {code chunk} must be a complete and valid R expression, because evalWithEcho() is a function call.
[3] The *character representation* of an object x is what as.character(x) gives.

```
There are <%=n%> red <%=type%>s
```

would produce the string 'There are 3 red horses'. If inlining multiple values, they are all pasted together without any separator. For example,

```
The letters of the alphabet are '<%=LETTERS%>'
```

produces 'The letters the alphabet are 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. To separate the elements with commas, use `<%=paste(LETTERS, collapse=", ")%>`. Alternatively, use `<%=hpaste(LETTERS)%>` to output 'A, B, C, ..., Z'[4].

2.5 Iterating over a mixture of RSP and text blocks

A useful feature of RSP is that it is possible to use RSP constructs that span multiple code and text blocks. For instance, the following will iterate over a set of text and RSP blocks:

```
The <%=n <- length(letters)%> letters in the English alphabet are:
<% for (i in 1:n) { %>
  <%=letters[i]%>/<%=LETTERS[i]%><%=if(i < n) ", "%>
<% } %>.
```

which generates:[5]: 'The 26 letters in the English alphabet are: a/A, b/B, c/C, d/D, e/E, f/F, g/G, h/H, i/I, j/J, k/K, l/L, m/M, n/N, o/O, p/P, q/Q, r/R, s/S, t/T, u/U, v/V, w/W, x/X, y/Y, z/Z .'

A more complex example is where one wish to generate a report on human genomic data across all of the 24 chromosomes and where the same type of analysis should be repeated for each chromosome. With RSP markup, this can be achieved by an outer loop over chromosomes:

```
<% for (chromosome in 1:24) { %>
\section{Chromosome <%=chromosome%>}
...
A mix of RSP and text blocks constituting
the analysis of the current chromosome.
...
<% } # for (chromosome ...) %>
```

Note that there exist no corresponding markup in Sweave. Instead, contrary to RSP, Sweave requires the each code chunk contains a complete R expression. This means that, in terms of the above example, in Sweave it is not possible to begin a for loop in one code chunk and end it in a succeeding one. This has to do with the fundamentally different way RSP and Sweave documents are processed. If using Sweave, one solution to this is to use RSP to generate the Sweave document, e.g. `rsp("report.Rnw.rsp")`.

3 Generating and inserting figures

Since the above RSP markups are powerful enough, there is no need for a specific markup for figures. This section shows how to create and embed image files into the final document.

[4]The `hpaste()` function of *R.utils* provides "human-readable" pasting of vectors.

[5]Of course, in this particular case, the above for-loop can be replaced by `<%=paste(letters, LETTERS, sep="/", collapse=", ")%>`.

3.1 Brief on including image files in LaTeX

When insert a figure in LaTeX, it is recommended to do so without specifying neither the path nor the filename extension of the image file, e.g. `\includegraphics{MyFigure}`. In order for this to work, one must specify the "image search path", e.g.

```
\graphicspath{{figures/}{figures/external/}}
```

which is preferably added to the beginning of the LaTeX file. This tells LaTeX to search for image files in directory figures/ as well as directory figures/external/. Moreover, when leaving out the filename extension, LaTeX will automatically search for image files with different filename extensions, e.g. `*.png`, `*.eps`, and `*.pdf`.

3.2 Defining a function that creates an image file

The `devEval()` function of the *R.utils* package is useful for creating an image file from a set of plot commands. For instance,

```
devEval("png", name="myFigure", width=840, aspectRatio=0.6, {
  curve(dnorm, from=-5, to=+5)
})
```

creates a PNG file named `myFigure.png` displaying the Guassian density distribution, an image file that is 840 pixels wide and 0.6*840=504 pixels high. Moreover, (by default) `devEval()` writes the image file to the figures/ directory. For more information, see `help("devEval")`.

To spare ourselves from having to repeat the same arguments each time an image is created, we define the following custom function for creating PNG image files with a certain default dimension (840 by 840 since the default aspect ratio is 1) and default graphical parameters (see `help("par")`):

```
# Use greater objects by default
setOption("devNew/args/par", list(cex=2, lwd=2));
toPNG <- function(name, ..., width=840) {
 devEval(type="png", width=width, name=name, ..., force=TRUE)$fullname;
}
```

This function creates a PNG file based on a set of plot commands and returns the so called *fullname* of the image file. The fullname of an image file is the filename without the filename extension. The following code creates a PNG image file 'figures/MyFigure,yeah,cool.png' and returns `"MyFigure,yeah,cool"`:

```
toPNG("MyFigure,yeah,cool", aspectRatio=0.6, {
  curve(dnorm, from=-5, to=+5);
})
```

3.3 Generating and embedding figures

With this setup, it is possible to create and embed a figure with the following tidy markup:

```
\includegraphics{<%=toPNG(name="MyFigure,yeah,cool", aspectRatio=0.6, {
  curve(dnorm, from=-5, to=+5);
})%>}
```

After the RSP code has been processed, and the image file has been created, the above simply produces the following markup in the generated LaTeX document:

```
\includegraphics{MyFigure,yeah,cool}
```

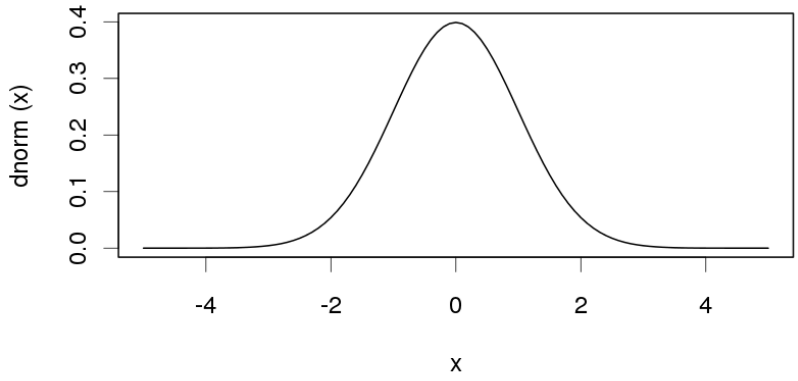

Figure 1: This figure was generated and inserted into the LaTeX document by RSP-embedded markup.

4 Templates - Reusing RSP and text blocks

Sometimes the very similar paragraphs of text, tables, or figures are used throughout a document with only minor differences. Instead of cut'n'pasting the same pieces of RSP and code to other places in the document, it is more robust and much easier to setup a template which is then reused in place. Because of the nature of RSP, setting up a template is as simple as wrapping the mixture of RSP and code blocks in a function definition. For example, assume you wish to reuse the following RSP and R blocks multiple times:

```
The sum of $x=<%=hpaste(1:n, abbreviate="\\ldots")%>$ is <%=sum(1:n)%>.
```

Then place it in a function definition:

```
<% myTemplate <- function(n, ...) { %>
The sum of $x=<%=hpaste(1:n, abbreviate="\\ldots")%>$ is <%=sum(1:n)%>.<%----%>
<% } # myTemplate() %>
```

We use a trailing (empty) RSP comment (`<%----%>`) to escape the following newline. Also, note that there is no limitation in how many RSP and text blocks you can use. After having defined the template, it can be reused any number of times by simply calling it as a function:

```
<% myTemplate(n=3) %>
```

which produces 'The sum of $x = 1, 2, 3$ is 6.'. Without the trailing RSP comment, the final document would contain a whitespace after the period and before the closing single-quote. A template can also be used within for loops. For example:

```
\begin{itemize}
<% for (ii in c(3,5,10,100)) { %>
 \item <% myTemplate(n=ii) %>
<% } # for (ii ...) %>
\end{itemize}
```

produces:

- The sum of $x = 1, 2, 3$ is 6.
- The sum of $x = 1, 2, 3, 4, 5$ is 15.
- The sum of $x = 1, 2, 3, \ldots, 10$ is 55.
- The sum of $x = 1, 2, 3, \ldots, 100$ is 5050.

5 Preprocessing directives

When an RSP-embedded document processed, it is first *preprocessed* before it is *translated* into an R script. During this step it is possible to modify the document by removing parts of it and inserting new pieces to it. For instance, by using the RSP hidden comments, multiple lines of the document can be silently dropped.

5.1 Hidden (non-nested) comments (<%--{anything}--%>)

The RSP markup `<%--{anything}--%>` will be treated as a comment that can contain *anything* (but `--%>`), which will not be translated and not part of the output, i.e. it will immediately be dropped. RSP comments are useful for excluding large sections of an RSP document. It is useful to understand that RSP comments are *greedy*, that is, anything between (and including) the `<%--` and the *first* following `--%>` will be dropped, which means that they cannot be nested. For example,

```
<%-- This is an RSP comment that will be dropped --%>
You can write a paragraph and drop a large portion of it using
RSP comments, <%-- All the below will be dropped
There are <%=n%> red <%=type%>s
<%-- and this --%> but they must not be nested --%>, because then
the output will be like this.
```

produces

```
You can write a paragraph and drop a large portion of it using
RSP comments,  but they must not be nested --%>, because then
the output will be like this.
```

Moreover, RSP comments are special in the sense that if (and only if) the remainder of the line following the comment consists of only whitespace symbols, then they are also dropped, including the newline. Thus, an `<%----%>` at the end of a line will prevent a newline from being inserted.

Appendix

Session information

- R version 2.13.0 Patched (2011-05-03 r55748), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=C`, `LC_MONETARY=C`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: R.methodsS3~1.2.1, R.oo~1.8.0, R.rsp~0.5.3, R.utils~1.7.5

This report was automatically generated using `rsp()` of the R.rsp package. Total processing time after RSP-to-R translation was 0.86 secs.