Dynamic document creation using RSP

Henrik Bengtsson

July 28, 2013

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, say, a 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 describe the RSP markup language and explain how it can be embedded in virtually any text-based source document which then may be compile into a final document. We will describe all of the RSP constructs available and illustrate how they can be used for plain text as well as for LaTeX documents.

RSP provides an easy, yet powerful and flexible way for generating reports and other documents in R. With a single R command it is possible to compile an RSP-embedded LaTeX, Sweave or knitr document into a read-to-view PDF file to name a few examples. As an example, we show how to produce this very document from an RSP-embedded LaTeX file. RSP can also be used for R package vignettes, which has become particularly simple since R v3.0.0.

Furthermore, because RSP is a so called content-independent markup language, it can be used to produce documents of any kind, e.g. plain text, LaTeX, Sweave, knitr, AsciiDoc, Markdown, HTML, XML, SVG, and even Javascript, R scripts, tab-delimited data files and so on. More over, with RSP it is possible to use literate programming constructs that are not possible in Sweave, e.g. looping over a mix of R source code and LaTeX text blocks.

This vignette is distributed as part of the R.rsp package, which is available on CRAN.

Contents

1 Introduction to RSP

The RSP markup language makes it possible to interweave text and R source code in a compact and powerful way for the purpose of generating text of any format. An example of an RSP-embedded string is

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

Using the *R.rsp* package[1], this RSP string can be processed and outputted as

```
> library("R.rsp")
> rcat("A random integer in [1,100]: <%=sample(1:100, size=1)%>\n")
A random integer in [1,100]: 77
```

As explained further in Section 2, the RSP construct `<%=⟨code chunk⟩%>` causes the R expression '⟨**code chunk**⟩' to be evaluated and its value to be inserted into the text (replacing the RSP construct). To process the RSP string without outputting the result, but instead returning it in a character string, do

```
> s <- rstring("A random integer in [1,100]: <%=sample(1:100, size=1)%>\n")
> print(s)
[1] "A random integer in [1,100]: 43\n"
```

1.1 rfile()

Consider an RSP-embedded text file `random.txt.rsp` that contains the above RSP string, i.e.

```
A random integer in [1,100]: <%=sample(1:100, size=1)%>
```

This file can be compiled as

```
> rfile("random.txt.rsp")
```

which generates a plain text file (`random.txt`) that contains

```
A random integer in [1,100]: 96
```

The filename of the output is the same as the RSP file with the file extension dropped. To try this example yourself, do

```
path <- system.file("exData", package="R.rsp")
rfile("random.txt.rsp", path=path)
```

To process the same RSP file into a string (without generating a file), do

```
path <- system.file("exData", package="R.rsp")
s <- rstring(file="random.txt.rsp", path=path)
cat(s)
```

The latter example illustrates how `rstring()` also can load strings from either a file (or a URL or a connection) as an alternative to taking plain strings as input.

---

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

1.2 Automatic postprocessing

For RSP documents that output certain file formats such as LaTeX, Sweave, and knitr, further processing of the RSP file output is often necessary before obtaining the final RSP product, e.g. a PDF file. The `rfile()` function will, based on the content type of the generated RSP output, detect if further processing ("postprocessing") is possible. If so, it is postprocessed to arrive at the final product. Currently, postprocessing to generate PDF and HTML output is supported for LaTeX, Markdown, AsciiDoc, Sweave and knitr documents. For instance, consider an RSP-embedded LaTeX document `report.tex.rsp`. When compiling it by

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

the `rfile()` method will (i) parse and evaluate the RSP-embedded LaTeX document resulting in a LaTeX document (`report.tex`), but in addition it will also (ii) compile the LaTeX document into a PDF (`report.pdf`). To disable RSP postprocessing, specify argument `postprocess=FALSE` when calling `rfile()`. To try postprocessing yourself, compile this very document by

```
> path <- system.file("doc", package="R.rsp")
> rfile("Dynamic_LaTeX_reports_with_RSP.tex.rsp", path=path)
```

The generated PDF (`Dynamic_LaTeX_reports_with_RSP.pdf`) will be available in the current directory of R (see `getwd()`). It is also possible to compile online RSP documents, e.g.

```
> url <- "http://r-forge.r-project.org/scm/viewvc.php/*checkout*/pkg/R.rsp/inst/doc/
          Dynamic_document_creation_using_RSP.tex.rsp?root=r-dots"
> rfile(url)
```

which produces a PDF of the current developer's version of this vignette.

2 The RSP markup language

An *RSP-embedded text document*, or short an *RSP document*, is a text document (string or file) that contains a set of *RSP constructs*, which each is defined by a pair of RSP start (`<%`) and end (`%>`) tags. There are three main types of RSP constructs:

1. *RSP comments* (`<%--...--%>`),
2. *RSP preprocessing directives* (`<%@...%>`), and
3. *RSP code expressions* (`<%...%>`).

Everything outside of RSP constructs is referred to as *RSP text*, or short just as '*text*'. When an RSP document is *processed* ("compiled"), it is first (i) *parsed* where comments are dropped and preprocessing directives are processed (and replaced by either text or RSP code expressions), then (ii) *translated* into plain R code, which when (iii) evaluated generates an *RSP product* (as a string or an output file).

Remark: An RSP comment may contain anything, including other RSP constructs. An RSP preprocessing directive may contain RSP comments (which are dropped), but not RSP code expressions. An RSP code expression may contain RSP comments (dropped) and RSP preprocessing directives (processed before parsing the RSP code expression), but not other RSP code expressions.

The simplest RSP document possible is a string that contains none of the above RSP constructs, just a single RSP text, which will be outputted "as is"[2] in the outputted RSP product. A slightly more elaborate example is the RSP string

---

[2]Except for escaped RSP start and end tags as explained in Section 2.2.

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

which is parsed into the following sequence of RSP constructs: (i) text 'A random integer in [1,100]: ', (ii) RSP code expression '<%=sample(1:100, size=1)%>', and (iii) text '\n'. When this sequence is later evaluated, the two text components are inserted into the RSP product "as is", whereas for the RSP code expression its contained R code is evaluated and the return value is inserted into the RSP product as text.

2.1 Comments (`<%--⟨anything⟩--%>`)

The RSP markups `<%--⟨anything⟩--%>`, `<%---⟨anything⟩---%>` and so on represent RSP comments, which can be used to drop large sections of an RSP document[3]. Unlike other RSP constructs, an RSP comment may contain other RSP constructs (also incomplete ones) including RSP comments. In order to nest RSP comments, any nested comment must use a different number of hyphens compared to the surrounding RSP comment[4]. For example, the following RSP document

```
<%-- This is an RSP comment that will be dropped --%>
You can write a paragraph and drop a large portion of it using
<%--- This comment contains both regular RSP expressions
There are <%=n%> red <%=type%>s
<%-- as well as another RSP comment --%>
which is nested. ---%>RSP comments.
```

is from the parser's point of view equivalent to the following RSP document

```
You can write a paragraph and drop a large portion of it using
RSP comments.
```

which produces

```
You can write a paragraph and drop a large portion of it using
RSP comments.
```

Remark: The number of prefix and suffix hyphens in an RSP comment must match, i.e. `<%--- A comment --%>` is *not* recognized as an RSP comment.

Remark: As explained in Section 2.4, *all* RSP constructs with a suffix hyphen causes any following whitespace including the first following line break to be trimmed, if and only if no other characters appear after the construct. This means that all RSP comments automatically trim such empty text. This explains why there are no empty lines after dropping the RSP comments in the above example.

Remark: Trailing whitespace can be trimmed from any line by adding `<%----%>` at the end of that line. For convenience, one can also use `<%-%>`, which is a short form for `<%----%>`.

---

[3]An RSP comment may can contain *anything* because it is dropped by the RSP parser at the very beginning before parsing the text and the other RSP constructs.

[4]The reason for needing a different number of hyphens in nested RSP comments is because RSP comments are *non-greedy*, that is, anything between (and including) the `<%--` and the *first* following `--%>` will be dropped.

2.2 Escaping RSP start and end tags (<%% and %%>)

To include '<%' and '%>' in the text of the RSP product they need to be escaped so that the RSP parser does not recognize them as being part of an RSP construct. This can be done as '<%%' and '%%>', respectively. For instance,

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

To clarify this further, consider the output of the following steps

```
> a <- "A random integer in [1,100]: <%%=sample(1:100, size=1)%%>\n"
> b <- rstring(a)
> rcat(b)
A random integer in [1,100]: 8
```

Remark: The string '<%%>' is interpreted as an escaped '<%' followed by a '>' and *not* as an empty RSP code expression.

2.3 RSP code expressions (<%...%>)

2.3.1 Evaluating code (<%⟨code⟩%>)

The RSP markup <%⟨code⟩%> evaluates the source code (without displaying it in the output document). For instance,

```
<%
n <- 3;  # Comments are kept
type <- "horse"
%>
```

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

Note that with this construct it is possible to insert incomplete code expression and completed it in a later RSP code block. For example,

```
Counting:<% for (i in 1:3) { %> <%=i%><% } %>.
```

For more details on this, see Section 2.3.3.

Remark: The source code specified in an RSP code expression must have at least one character (which may be a blank). Notably, <%%> is *not* an RSP code expression with reasons is explained in Section 2.2.

2.3.2 Inlining values of variables and expressions (<%=⟨code chunk⟩%>)

The RSP markup <%=⟨code chunk⟩%> evaluates the code chunk (without inserting the code itself into the document) and inserts the character representation[5] of the returned object. For instance,

```
Today's date is <%=Sys.Date()%>
```

would produce the string 'Today's date is 2013-07-29'. If inlining a vector of values, they are all pasted together without a separator. For example,

---

[5]In R, the *character representation* of an object `x` is what generic function `rpaste(x)` gives. The default returns `paste0(as.character(x), collapse="")`, but may be refined and customized for particular classes in the RSP document.

```
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'[6].

Remark: When inlining results, the ⟨code chunk⟩ must be a complete and valid R expression, otherwise it would not be able to evaluate it and return the result.

2.3.3   Iterating over a mixture of RSP constructs 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 code 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[7]: '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, as illustrated by

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

Remark: Note that there exist no corresponding construct in noweb-style markup languages, e.g. Sweave. Instead, contrary to RSP, Sweave requires that each of the code chunks 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 is to use RSP markup in the Sweave document, process it via RSP to generate a pure Sweave document, which can then in turn be processed using the Sweave engine to get a TeX file and eventually the final PDF document. Alternatively, one can use `rfile()` to complete all these steps at once, e.g. `rfile("report.Rnw.rsp")` will generate `report.pdf` as well as the intermediate `report.Rnw` and `report.pdf` files. This illustrates the power of the RSP language and how it can be used as a preprocessor of any other literate programming language.

---

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

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

2.3.4   Templates - reusing a mixture of RSP and text blocks

Sometimes rather similar paragraphs of text, tables, or figures are used throughout a document with only minor differences. Instead of manually cut'n'pasting the same pieces to other places of the document over and over, it is more robust and much easier to setup a template function which can then used in place.

Because of the nature of the RSP language, setting up a template is as simple as wrapping the mixture of RSP and code blocks (Section 2.3.3) in a function definition. For example, assume you wish to reuse the following RSP passage multiple times with different values of `n` in a LaTeX document

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

The solution is to wrap it up in a *template* function

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

Recall that there is no limitation in how many text and RSP blocks you can use. Note also how we use an empty RSP comment (`<%-%>`) at the end of the template function. That is used in order to escape the trailing line break (Section 2.1). Without the trailing RSP comment, the final document would contain a whitespace after the period (and before the closing single-quote in the previous sentence). The above template function can now be used by calling it within an RSP code block as

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

which produces 'The sum of $x = 1, 2, 3$ is 6.'. Naturally, a template function can be reused any number of times. For example, the following RSP-embedded LaTeX block

```
\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.

Remark: When using templates, use only `<% ... %>` and not `<%= ... %>`. The reason for this is that the way templates are setup they already output their contents/results to the generated RSP product.

2.4   Trimming whitespace for RSP constructs

Because of readability of the RSP source document, RSP constructors are often placed on lines separate from the text blocks but also other RSP constructs. Depending on how whitespace and line breaks (as introduced by the above coding style) are handled in the output format, this may or may not affect how the output document renders. For instance, line breaks explicitly introduced in an RSP source document will make the an outputted plain text document look different whereas single line breaks will not affect how a LaTeX document is compiled.

2.4.1 Trimming of lines with only non-text RSP constructs

The whitespace before and after RSP constructs that are on their own lines without no other non-whitespace text will *always* have their surround whitespace trimmed off. They will also have the first following line break removed, unless it is manually controlled as describe in the next subsection. This automatic trimming of "stand-alone" RSP constructs enhanced readability of RSP source documents without introducing unnecessary whitespace and line breaks. Consider the following example and think about how whitespace and line breaks are removed or kept:

```
You don't have to worry too much about whitespace, e.g. the
  <%
     s <- "will have its surrounding whitespace"
  %>
above RSP expression <%=s%>
trimmed off as well as its trailing line break.
```

which produces

```
You don't have to worry too much about whitespace, e.g. the
above RSP expression will have its surrounding whitespace
trimmed off as well as its trailing line break.
```

2.4.2 Manually controlling trailing whitespace and line breaks (-%> and +%>)

Also for RSP constructs that are not surrounded by only whitespace, it is possible to control how whitespace and line breaks following RSP constructs are trimmed. This is done via so called *RSP end tag specifications*. Those apply to all types of RSP constructs. By adding an *end tag hyphen* at the end of an RSP construct (i.e. -%>), all following whitespace (space and tabs) including the first line break (the newline) is dropped ("trimmed"), if and only if there are no other characters following the RSP construct. How this works is illustrated in the following example:

```
> rstring("A random integer in [1,100]: <%=sample(1:100, size=1)%>\n")
[1] "A random integer in [1,100]: 48\n"
> rstring("A random integer in [1,100]: <%=sample(1:100, size=1)-%>\n")
[1] "A random integer in [1,100]: 48"
> rstring("A random integer in [1,100]: <%=sample(1:100, size=1)-%> \t \n\n\n")
[1] "A random integer in [1,100]: 48\n\n"
```

To disable the trimming of trailing whitespace and line breaks, add an *end tag plus sign*, i.e. '+%>'. For instance,

```
> rstring("abc\n<%='DEF'%>\nGHI")
[1] "abc\nDEFGHI"
> rstring("abc\n<%='DEF'-%>\nGHI")
[1] "abc\nDEFGHI"
> rstring("abc\n<%='DEF'+%>\nGHI")
[1] "abc\nDEF\nGHI"
```

2.5 Preprocessing directives (<%@...%>)

An *RSP preprocessing directive*, or short an *RSP directive*, is an RSP construct that starts with an '@' followed by a *directive* and zero or more *named arguments*, e.g. <%@include file="random.tex.rsp"%>. RSP directives can for instance be used to include the content of another file or conditionally include/exclude

parts of the existing document. RSP preprocessing directives facilitate modularization of RSP documents such that the same RSP module can be reused in many places, which in turn minimizes and often eliminates the manual and often error-prone process of cut-and-pasting content between documents. RSP preprocessing directives are processed during the parsing of the RSP document. This is done *after* RSP comments have been dropped and *before* RSP code expressions are parsed. This means that:

- RSP comments can be used to commenting out and hence exclude RSP preprocessing directives, e.g. `<%--- <%@include file="random.tex.rsp"%> ---%>`.

- RSP preprocessing directives may insert new RSP code expressions as if they where part of the original RSP document, e.g. `<%@include file="random.tex.rsp"%>`. Any RSP comments inserted will be dropped as expected. If other RSP directives are inserted, they are processed immediately.

- After being processed, only RSP text and RSP code expressions remains (no RSP preprocessing directives). Notably, RSP code expressions *do not see or have access* to RSP preprocessing directives or any of their assigned variables.

The value of an RSP directive argument is parsed as a *GString*[8]. A GString is a regular string that optionally may contain bash-like syntax for inserting values of system environment variables. For instance, if the environment variable 'R_HOME' is set to the home directory of the R installation, then `<%@include file="${R_HOME}/doc/AUTHORS"%>` is parsed as `<%@include file="/path/to/R/doc/AUTHORS"%>` (resulting in the content of that 'AUTHORS' file to be included "as is"). Note how this substitution is indepent of the R language. For convenience, when processing an RSP document using the RSP parser of the *R.rsp* package, GString also looks for a matching variables elsewhere. Notably, it searches (i) among the variables in the R workspace, (ii) among the R options, and then (iii) among the system environment variable.[9]

Remark: Variables set in RSP code expressions *cannot* be used in RSP directives, because they are only assigned after the RSP preprocessing directives have been processed.

Remark: All RSP directive arguments (also known as "attributes") must be named with names having the correct case and not being abbreviated. For instance, `<%@include file="random.tex.rsp"%>` is correct whereas `<%@include "random.tex.rsp"%>`, `<%@include File="random.tex.rsp"%>` and `<%@include f="random.tex.rsp"%>` are not.

Remark: The formal syntax for an RSP preprocessing directive is `<%@⟨directive⟩ [⟨name⟩="⟨value⟩"]*%>`. An argument name may consist of alphanumeric characters and underscores where the first being a letter or an underscore. The argument values are parsed as strings and must be quoted either using matching single or double quotes.

Remark: Just like RSP comments, RSP directives are designed to be independent of the programming language. In other words, by design these constructs are not relying on the R language. Another way look at it is that the processing of RSP comments and RSP directives may be done by an *RSP processor* written in a different language than R, e.g. Python or C.

2.5.1 Including text and file contents

It is possible to include the contents of other text and RSP documents. These can be either specified as text strings (argument 'content') or as files (argument 'file'), which may be either a file on the file system or URL/document online.

---

[8]GStrings can be processed in R by the `gstring()` function in the *R.utils* package.

[9]Although the RSP parser of the *R.rsp* package also looks for GString variable in the R environment and its options, it does not mean that these variables are dependent on the R language. The substituted value will always be interpreted as a character string. An RSP parser implemented in Python could take a similar approach.

2.5.1.1 Including file contents (`<%@include file="⟨pathname|URL⟩"%>`) To include the content of a text file "as is" during preprocessing, use the '`file`' argument. This may be used to include a local file or a file online (specified as a URL). For instance, to include the content of text file '`randoms.txt`' as is into the RSP document, use

`<%@include file="randoms.txt"%>`

To include the *RSP preprocessed* content of RSP file '`random.txt.rsp`', use

`<%@include file="randoms.txt.rsp"%>`

Any file pathname is *relative to the directory* of the (host) RSP document that contains the include directive. For instance, consider an RSP file '`rsp/overview.txt.rsp`' and a text file '`inst/NEWS`'. Next, if an RSP document includes the RSP file as

`<%@include file="rsp/overview.txt.rsp"%>`

then that RSP file can in turn include the text file as

`<%@include file="../inst/NEWS"%>`

Remark: Only *relative pathnames* are allowed[10]. If an RSP document could use absolute pathnames, then it would only compile on a particular computer system/user account and would have to be altered in order to compile elsewhere. Note that it is always possible to use file system links[11] to link to files elsewhere on the file system without having to copy them the directory of the source RSP document (or a subdirectory therein).

2.5.1.2 Including plain text (`<%@include content="⟨content⟩"%>`) Although not commonly needed, it is, as an alternative to contents of files, possible to include contents of strings. For example, the template RSP document

```
The R_HOME environment variable was '<%@include content="${R_HOME}"%>'
at the time when this document was preprocessed.
```

results in an preprocessed RSP document consisting of

```
The R_HOME environment variable was 'C:/PROGRA~1/R/R-30~1.1PA'
at the time when this document was preprocessed.
```

2.5.2 Document metadata (`<%@meta ...%>`)

It is possible to assign non-visible metadata to RSP documents. This metadata can be set and retrieved by RSP preprocessing directives. To set an RSP metadata variable, use directive `<%@meta name="⟨name⟩" content="⟨content⟩"%>`. To insert RSP metadata into the RSP output, use directive `<%@meta name="⟨name⟩"%>`. For example, in LaTeX you can do

---

[10]The RSP compiler in the *R.rsp* package will given an error if it detects an absolute pathname.

[11]The `createLink()` function in *R.utils* provides a platform-independent way for creating file system links.

```
<%@meta name="title" content="My Report"%>
<%@meta name="author" content="John Doe"%>
<%@meta name="keywords" content="statistics, report"%>

% Set the PDF metadata
\usepackage[hidelinks]{hyperref}
\hypersetup{
  pdfauthor={<%@meta name="author"%>},
  pdftitle={<%@meta name="title"%>},
  pdfkeywords={<%@meta name="keywords"%>}
}

\title{<%@meta name="title"%>}
\author{<%@meta name="author"%>}
\keywords{<%@meta name="keywords"%>}
```

Note how this sets both the displayed metadata as well as the non-visible PDF metadata.

For convenience, there is a short format for setting metadata, which is `<%@meta ⟨name⟩="⟨content⟩"%>`.

2.5.2.1 Setting metadata via R vignette metadata The R framework uses a LaTeX-flavored markup for specifying metadata in vignettes, e.g. `\VignetteIndexEntry{My Report}` and `\VignetteKeyword{statistics, report}`. It is possible to "import" this metadata information via `<%@meta content="⟨R vignette metadata markup⟩" language="R-vignettes"%>`. For instance,

```
<%@meta content="
  DIRECTIVES FOR R:
  %\VignetteIndexEntry{My Report}
  %\VignetteKeyword{statistics, report}
  %\VignetteAuthor{John Doe}
" language="R-vignette"%>

\title{<%@meta name="title"%>}
\author{<%@meta name="author"%>}
\keywords{<%@meta name="keywords"%>}
```

will import the three R `\Vignette⟨Name⟩{}` "commands" into RSP metadata variables 'title', 'keywords' and 'author'. As illustrated, any content that is not recognizes as R vignette metadata, or is of unknown markup, is ignored, which means that it is possible to include "anything" part of the '`content`' attribute. Note that this construct will (i) import the metadata into the RSP document at the same time as (ii) the R vignette mechanism can utilize it. This eliminates the need for reproducing the same information in the RSP source document.

2.5.3 Preprocessing variables (`<%string ...%>`)

In addition to metadata, it is possible to define, set and get custom variables available during the pre-processing of the RSP document. To set an RSP preprocessing *string* variable, use directive `<%@string name="⟨name⟩" content="⟨content⟩" default="⟨default⟩"%>`, where '`default`' is optional and the value used if '`content`' is empty. To insert a preprocessing variable into the RSP output, use directive `<%@string name="⟨name⟩"%>`. There exist analogue directives for *numeric*, *integer* and *logical* variables.

```
<%@string name="page_format" content="article"%>
<%@string name="page_size" content="a4paper"%>
```

```
\documentclass[<%@string name="page_size"%>]{<%@string name="page_format"%>}
```

For convenience, there is a short format for setting a variable, which is `<%@string ⟨name⟩="⟨content⟩"%>`.

2.5.4   Conditional inclusion/exclusion (`<%@if ...%>`)

Using so called RSP 'if-then-else' preprocessing directives, it is possible to include (exclude) parts of an RSP source document such that they appear (do not appear) in the output RSP product. The full syntax of an RSP 'if-then-else' variable is `<%@if test="⟨test⟩" name="⟨variable⟩" content="⟨value⟩" negate="⟨logical⟩"%>` ⟨include⟩ `<%@else%>` ⟨exclude⟩ `<%@endif%>`, where the `<%@else%>` statement is optional. The argument '`test`' specifies the type of test to conduct, '`name`' the variable name to be tested, and '`content`' the value to compare the variable to. The '`negate`' argument is optional and specifies whether the result of the test should be negated or not. The following tests are available (aliases in parentheses):

- `test="exists"`: check if a variable exists.
- `test="equal-to"` (`"=="`): check if a variable is equal to a specific value.
- `test="not-equal-to"` (`"!="`): check if a variable is not equal to a specific value.
- `test="less-than-or-equal-to"` (`"<="`): check if a variable is less than or equal to a specific value.
- `test="less-than"` (`"<"`): check if a variable is less than a specific value.
- `test="greater-than-or-equal-to"` (`">="`): check if a variable is greater than or equal to a specific value.
- `test="greater-than"` (`">"`): check if a variable is greater than a specific value.

For example,

```
<%@string name="version" content="devel"%>
<%@if test="exists" name="version"%>
 <%@if test="equal-to" name="version" content="devel"%>
This document presents methods that are under development.
 <%@else%>
This document presents methods that are well tested and stable.
 <%@endif%>
<%@else%>
Preprocessing variable 'version' was not set.
<%@endif%>
```

produces

```
This document presents methods that are under development.
```

If one removes the first line, then the output is

```
Preprocessing variable 'version' was not set.
```

For convenience, it is also possible to use a short format for testing a variable, which is `<%@if test="equal-to" ⟨name⟩="⟨value⟩"%>`.

2.5.4.1 If-then-else aliases (`<%@ifeq ...%>`) For some of the common tests, there are also so called *alias directives*. They are:

- `<%@ifeq ...%>` is an alias for `<%@if test="equal-to" ...%>`.

- `<%@ifneq ...%>` is an alias for `<%@if test="not-equal-to" ...%>`.

Remark: Note that by combining the above aliases with the short format, one can write `<%@ifeq A="42"%>` instead of `<%@if test="equal-to" name="A" content="42"%>`.

2.6 Advanced

2.6.1 Compiling a standalone RSP document from a preprocessed but not evaluated RSP document template

However convenient it is to use a modularized RSP document that reuses many different RSP subdocuments, it may be tedious to share such as document with others. In order for someone else to compile the same RSP document, it is necessary to make sure all files being used are also shared with the receiver together with instructions where to put them. Some of these files may only be included under certain conditions, making the just add to the overall clutter at the receiver's end. In such cases, it may be useful to produce *a preprocessed but not yet evaluated RSP document* from the main RSP document. Such an RSP document will not contain any preprocessing directives and in that sense be self-contained and easier to share. It is also likely be easier for the receiver to follow. To compile such a stand-alone RSP document, do:

```
> s <- rclean(file="random.txt.rsp")
```

This outputs a file `preprocessed-random.txt.rsp` in the current directory. Documents compiled this way will contain neither RSP comments nor RSP preprocessing directives. This may be useful if the source RSP subdocuments contain private comments and private parts that are conditional excluded during preprocessing.

3   Solutions specific to the R environment

3.1   Working with figures

Because the RSP language is context unaware, it does not have constructs for inserting content such as figures and tables. However, as we will show in this section, it is still very easy to create and include figures in the output document. We will first explain how to create image files and then show how they can be included in LaTeX, Markdown and HTML documents in a standard and unified way.

3.1.1   Creating image files

The `toPDF()` and `toPNG()` functions of the *R.devices* package are useful for creating image files to be included in for instance LaTeX and HTML. For example,

```
library("R.devices")

# Use greater objects by default
setOption("devNew/args/par", list(lwd=2));

toPDF("myFigure,yeah,cool", aspectRatio=0.6, {
```

```
  curve(dnorm, from=-5, to=+5)
})
```

creates a PDF image file named `myFigure,yeah,cool.png` whose height is 60% of its width and displays the Guassian density distribution. By default, the `toPDF()` and `toPNG()` functions write image files to the `figures/` directory. For more information, see `help("toPDF", package="R.devices")` and the vignettes of the *R.devices* package.

3.1.2 Including image files in LaTeX

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

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

which is preferably added to the beginning of the LaTeX document. This tells LaTeX to search for image files in directory `figures/` as well as directory `figures/external/`. Moreover, by leaving out the filename extension, LaTeX will automatically search[12] for image files with different filename extensions, e.g. `*.png` and `*.pdf`. With this setup, it is possible to create and embed a figure as the one in Figure 1 using the
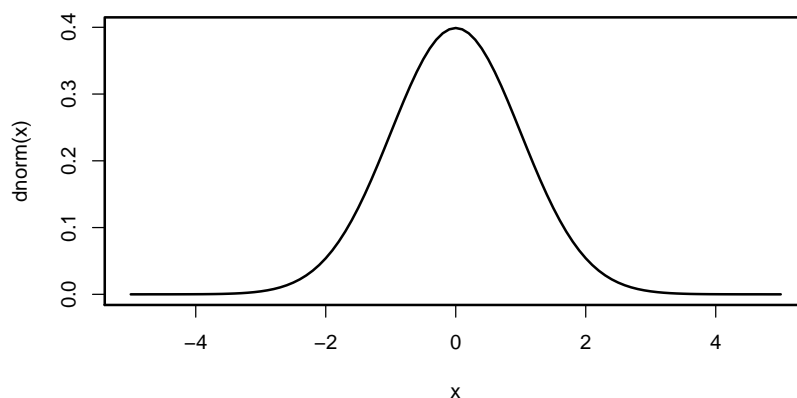

Figure 1: This figure was generated and inserted into the LaTeX document using RSP.

following tidy RSP and LaTeX markup[13]

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

---

[12]In order for `\includegraphics{}` to locate the proper image file when leaving out the filename extension, the image file must not have other periods in the filename other than the one used for the extension.

[13]For readability, we have dropped the usual LaTeX commands for setting up the `figure` environment, centering and rescaling the image and creating the figure caption.

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

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

The reason for the latter is that by default `toPDF()` and `toPNG()` return the filename of the created image file without its directory and without its filename extension, i.e. `"myFigure,yeah,cool"`. As a final remark, note that by replacing `toPDF()` with `toPNG()`, a PNG image file is inserted instead, which can dramatically reduce the size of the final PDF document, especially when scatter plots with a large number of data points are generated.

3.1.3 Including image files in HTML

An image file is included in an HTML document via its pathname. To have `toPNG()`, `toSVG()` and so on, return the (relative) pathname of files created, add the following to the top of your RSP document:

```
<%
setOption("devEval/args/field", "pathname")
%>
```

With this setting, one can then include PNG image files in the HTML document as

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

which results in

```
<img src="figures/MyFigure,yeah,cool.png">
```

3.1.4 Including image files in Markdown

Just as for HTML documents, on need to set the option that images created by `toPNG()` etc are returned by their (relative) pathnames by specifying option `devEval/args/field` (see above subsection). With this setting, one can then include PNG image files in the Markdown document as

```
![This figure was generated and inserted into the Markdown document using RSP.]
(<%=toPNG("MyFigure,yeah,cool", aspectRatio=0.6, {
  curve(dnorm, from=-5, to=+5);
})%>)
```

which results in

```
![This figure was generated and inserted into the Markdown document using RSP.]
(figures/MyFigure,yeah,cool.png)
```

3.2 Including R code

3.2.1 Evaluating and capturing expression and output (`<%= evalCapture (⟨code chunk⟩)%>`)

The `evalCapture()` function allows us to evaluate, embed and echo the text output of an R code chunk. Note that it is only standard output that is echoed, i.e. plots are *not* automatically embedded as figures. To include figures, see Section 3.1. Continuing, any formatting of the capture has to be done in the format of the output document. For instance, the following RSP-embedded LaTeX text

```
\begin{verbatim}
<%=evalCapture({
for (kk in 1:3) {
  printf("Iteration #%d\n", kk)
}

print(Sys.time())
type <- "horse";  # Comments are dropped
type
})%>
\end{verbatim}
```

produces

```
> for (kk in 1:3) {
+     printf("Iteration #%d\n", kk)
+ }
Iteration #1
Iteration #2
Iteration #3
> print(Sys.time())
[1] "2013-07-29 06:06:43 PDT"
> type <- "horse"
> type
[1] "horse"
```

By adding arguments `code=FALSE` and/or `output=FALSE` to `evalCapture()` one can control whether the deparsed R source code and/or the output of each subexpression is captured or not.

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

3.2.2   Future directions for including code

The `evalCapture()` construct is rather rudimentary and has limitations on how the source code and echoed outputs are formatted. A much more powerful solution is then to utilize CRAN packages such as *formatR*, *evaluate* and *highlight*. The goal is to incorporate neat support for these in a future version of the *R.rsp* package.

4   History

The RSP markup language was first developed in May 2002. It was inspired by the JavaServer Pages (JSP) markup languare available for the Java programming language which is used to create dynamic web content via for instance the Apache Tomcat web server. A related solution is the Active Server Pages (ASP) markup language for the VBScript language.

Initially the RSP engine was implemented as part of the *R.io* package (deprecated) but was moved to the *R.rsp* package in July 2005. In early 2013, the RSP engine of *R.rsp* underwent a major redesign (compared to *R.rsp* v0.8.2 and before). This somewhat also affected the RSP language, but great efforts have been taken to keep it backward compatible.

The RSP language and the *R.rsp* engine has been in particularly heavy use by thousands of users of the Aroma Project (`http://aroma-project.org/`) as part of the generation of interactive Javascript and HTML reports. Several R packages are using RSP for their HTML and PDF vignettes, which has become particularly simple since R v3.0.0.

In June 2007, the *brew* package (on CRAN) was introduced independently of the *R.rsp* package. Although a different engine is used, the brew syntax is similar to the one in the RSP markup language.

Appendix

Session information

- R version 3.0.1 Patched (2013-07-25 r63414), `x86_64-w64-mingw32`

- Locale: `LC_COLLATE=C`, `LC_CTYPE=English_United States.1252`, `LC_MONETARY=English_United States.1252`, `LC_NUMERIC=C`, `LC_TIME=English_United States.1252`

- Base packages: base, datasets, grDevices, graphics, methods, stats, utils

- Other packages: R.devices 2.5.0, R.methodsS3 1.4.4, R.oo 1.13.10, R.rsp 0.9.17, R.utils 1.26.0

- Loaded via a namespace (and not attached): tools 3.0.1

This report was automatically generated using `rfile()` of the R.rsp package. Total processing time after RSP-to-R translation was 0.32 secs.