knitr: A General-Purpose Tool for Dynamic Report Generation in R

Yihui Xie

May 8, 2016

The original paradigm of literate programming was brought forward mainly for software development, or specifically, to mix source code (for computer) and documentation (for human) together. Early systems include WEB and Noweb; Sweave (**?**) was derived from the latter, but it is less focused on documenting software, instead it is mainly used for reproducible data analysis and generating statistical reports. The **knitr** package (**?**) is following the steps of Sweave. For this manual, I assume readers have some background knowledge of Sweave to understand the technical details; for a reference of available options, hooks and demos, see the package homepage `http://yihui.name/knitr/`.

1 Hello World

A natural question is why to reinvent the wheel. The short answer is that extending Sweave by hacking SweaveDrivers.R in the **utils** package is a difficult job to me. Many features in **knitr** come naturally as users would have expected. Figure 1 is a simple demo of some features of **knitr**.

I would have chosen to hide the R code if this were a real report, but here I show the code just for the sake of demonstration. If we type *qplot()* in R, we get a plot, and the same thing happens in **knitr**.

```
fit <- lm(dist ~ speed, data = cars)  # linear regression
par(mar = c(4, 4, 1, 0.1), mgp = c(2, 1, 0))
with(cars, plot(speed, dist, panel.last = abline(fit)))
text(10, 100, "$Y = \\beta_0 + \\beta_1x + \\epsilon$")
library(ggplot2)
qplot(speed, dist, data = cars) + geom_smooth()
```

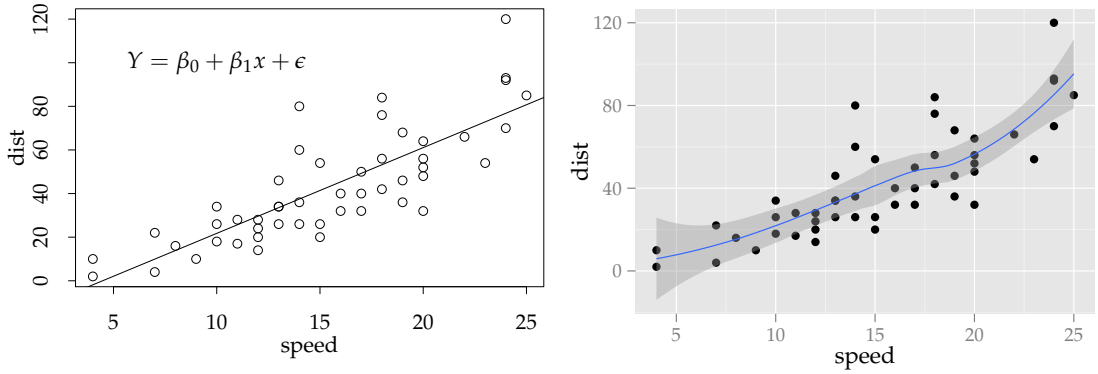


Figure 1: A simple demo of possible output in **knitr**: (1) multiple plots per chunk; (2) no need to *print()* objects in **ggplot2**; (3) device size is 4×2.8 (inches) but output size is adjusted to `.45\textwidth` in chunk options; (4) base graphics and **ggplot2** can sit side by side; (5) use the *tikz()* device in **tikzDevice** by setting chunk option `dev='tikz'` (hence can write native LaTeX expressions in R plots); (6) code highlighting.

If we draw two plots in the code, **knitr** will show two plots and we do not need to tell it how many plots are there in the code in advance. If we set `out.width='.49\\textwidth'` in chunk options, we get it in the final output document. If we say `fig.align='center'`, the plots are centered. That's it. Many enhancements and new features will be introduced later. If you come from the Sweave land, you can take a look at the page of transition first: `http://yihui.name/knitr/demo/sweave/`.

2 Design

The flow of processing an input file is similar to Sweave, and two major differences are that **knitr** provides more flexibility to the users to customize the processing, and has many built-in options such as the support to a wide range of graphics devices and cache. Below is a brief description of the process:

1. **knitr** takes an input file and automatically determines an appropriate set of patterns to use if they are not provided in advance (e.g. file.Rnw will use `knit_patterns$get('rnw')`);
2. a set of output hooks will also be set up automatically according to the filename extension (e.g. use LaTeX environments or HTML elements to wrap up R results);
3. the input file is read in and split into pieces consisting of R code chunks and normal texts; the former will be executed one after the other, and the latter may contain global chunk options or inline R code;
4. for each chunk, the code is evaluated using the **evaluate** package (**?**), and the results may be filtered according to chunk options (e.g. `echo=FALSE` will remove the R source code)
   (a) if `cache=TRUE` for this chunk, **knitr** will first check if there are previously cached results under the cache directory before really evaluating the chunk; if cached results exist and this code chunk has not been changed since last run (use MD5 sum to verify), the cached results will be (lazy-) loaded, otherwise new cache will be built; if a cached chunk depends on other chunks (see the `dependson` option) and any one of these chunks has changed, this chunk must be forcibly updated (old cache will be purged)
   (b) there are six types of possible output from **evaluate**, and their classes are `character` (normal text output), `source` (source code), `warning`, `message`, `error` and `recordedplot`; an internal S3 generic function *wrap()* is used to deal with different types of output, using output hooks defined in the object `knit_hooks`
   (c) note plots are recorded as R objects before they are really saved to files, so graphics devices will not be opened unless plots have really been produced in a chunk
   (d) a code chunk is evaluated in a separate empty environment with the global environment as its parent, and all the objects in this environment after the evaluation will be saved if `cache=TRUE`
   (e) chunk hooks can be run before and/or after a chunk
5. for normal texts, **knitr** will find inline R code (e.g. in `\Sexpr{}`) and evaluate it; the output is wrapped by the `inline` hook;

The hooks play important roles in **knitr**: this package makes almost everything accessible to the users. Consider the following extremely simple example which may demonstrate this freedom:

```
1 + 1

## [1] 2
```

There are two parts in the final output: the source code `1 + 1` and the output `[1] 2`; the comment characters `##` are from the default chunk option `comment`. Users may define a hook function for the source code like this to use the `lstlisting` environment:

```
knit_hooks$set(source = function(x, options) {
    paste("\\begin{lstlisting}\n", x, "\\end{lstlisting}\n", sep = "")
})
```

Similarly we can put other types of output into other environments. There is no need to hack at Sweave.sty for **knitr** and you can put the output in any environments. What is more, the output hooks make **knitr** ready for other types of output, and a typical one is HTML (there are built-in hooks). The website has provided many examples demonstrating the flexibility of the output.

3 Features

The **knitr** package borrowed features such as tikz graphics and cache from **pgfSweave** and **cacheSweave** respectively, but the implementations are different. New features like code reference from an external R script as well as output customization are also introduced. The feature of hook functions in Sweave is re-implemented and hooks have new usage now. There are several other small features which are motivated from my everyday use of Sweave. For example, a progress bar is provided when knitting a file so we roughly know how long we still need to wait; output from inline R code (e.g. `\Sexpr{x[1]}`) is automatically formatted in TEX math notation (like 1.2346×10^8) if the result is numeric. You may check out a number of specific manuals dedicated to specific features such as graphics in the website: `http://yihui.name/knitr/demos`.

3.1 Code Decoration

The **highr** package (**?**) is used to highlight R code, and the **formatR** package (**?**) is used to reformat R code (like `keep.source=FALSE` in Sweave but will also try to retain comments). For LATEX output, the **framed** package is used to decorate code chunks with a light gray background. If this LATEX package is not found in the system, a version will be copied directly from **knitr**. The prompt characters are removed by default because they mangle the R source code in the output and make it difficult to copy R code. The R output is masked in comments by default based on the same rationale. It is easy to revert to the output with prompts (set option `prompt=TRUE`), and you will quickly realize the inconvenience to the readers if they want to copy and run the code in the output document:

```
> x <- rnorm(5)
> x
[1]  0.2649 -0.4324 -2.5270 -0.3021  1.4652
> var(x)
[1] 2.103
```

The example below shows the effect of `tidy=TRUE/FALSE`:

```
## option tidy=FALSE
for(k in 1:10){j=cos(sin(k)*k^2)+3;print(j-5)}
```

```
## option tidy=TRUE
for (k in 1:10) {
    j <- cos(sin(k) * k^2) + 3
    print(j - 5)
}
```

Note = is replaced by <- because `options('formatR.arrow')` was set to be `TRUE` in this document; see the documentation of *tidy.source()* in **formatR** for details.

Many highlighting themes can be used in **knitr**, which are borrowed from the **highlight** package by Andre Simon[1]; it is also possible to use themes from `http://www.eclipsecolorthemes.org/` by providing a theme id to **knitr**[2]. See `?knit_theme` for details.

3.2 Graphics

Graphics is an important part of reports, and several enhancements have been made in **knitr**. For example, grid graphics may not need to be explicitly printed as long as the same code can produce plots in R (in some cases, however, they have to be printed, e.g. in a loop, because you have to do so in an R terminal).

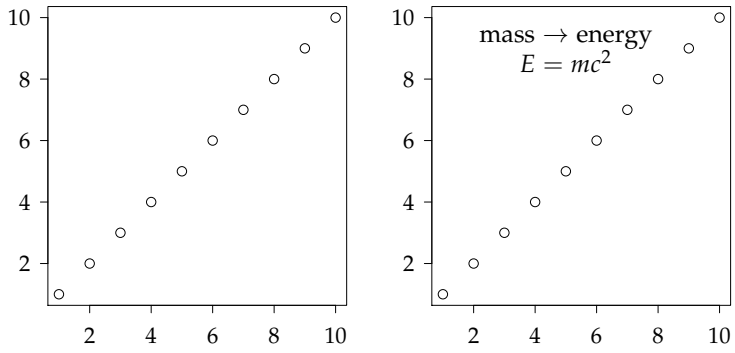
Graphical Devices

Over a long time, a frequently requested feature for Sweave was the support for other graphics devices, which has been implemented since R 2.13.0. Instead of using logical options like `png` or `jpeg` (this list can go on and on), **knitr** uses a single option `dev` (like `grdevice` in Sweave) which has support for more than 20 devices. For instance, `dev='png'` will use the *png()* device, and `dev='CairoJPEG'` uses the *CairoJPEG()* device in the **Cairo** package (it has to be installed first, of course). If none of these devices is satisfactory, you can provide the name of a customized device function, which must have been defined before it is called.

Plot Recording

As mentioned before, all the plots in a code chunk are first recorded as R objects and then "replayed" inside a graphical device to generate plot files. The **evaluate** package will record plots per *expression* basis, in other words, the source code is split into individual complete expressions and **evaluate** will examine possible plot changes in snapshots after each single expression has been evaluated. For example, the code below consists of three expressions, out of which two are related to drawing plots, therefore **evaluate** will produce two plots by default:

```
par(mar = c(3, 3, 0.1, 0.1))
plot(1:10, ann = FALSE, las = 1)
text(5, 9, "mass $\\rightarrow$ energy\n$E=mc^2$")
```



This brings a significant difference with traditional tools in R for dynamic report generation, since low-level plotting changes can also be recorded. The option `fig.keep` controls which plots to keep in the output; `fig.keep='all'` will keep low-level changes as separate plots; by default (`fig.keep='high'`), **knitr** will merge low-level plot changes into the previous high-level plot, like most graphics devices do. This feature may be useful for teaching R graphics step by step. Note, however, low-level plotting commands in a single expression (a typical case is a loop) will not be recorded accumulatively, but high-level plotting

[1] not the R package mentioned before; for a preview of these themes, see `http://www.andre-simon.de/dokuwiki/doku.php?id=theme_examples`

[2] many thanks to Ramnath Vaidyanathan for the work on themes

commands, regardless of where they are, will always be recorded. For example, this chunk will only produce 2 plots instead of 21 plots because there are 2 complete expressions:

```
plot(0, 0, type = "n", ann = FALSE)
for (i in seq(0, 2 * pi, length = 20)) points(cos(i), sin(i))
```

But this will produce 20 plots as expected:

```
for (i in seq(0, 2 * pi, length = 20)) {
    plot(cos(i), sin(i), xlim = c(-1, 1), ylim = c(-1, 1))
}
```

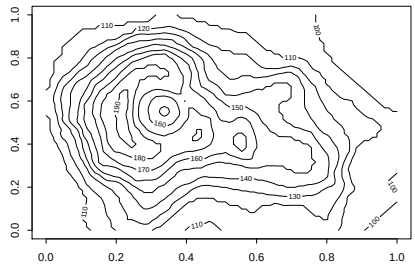
As I showed in the beginning of this manual, it is straightforward to let **knitr** keep all the plots in a chunk and insert them into the output document, so we no longer need the `cat('\\includegraphics{}')` trick.

We can discard all previous plots and keep the last one only by `fig.keep='last'`, or keep only the first plot by `fig.keep='first'`, or discard all plots by `fig.keep='none'`.

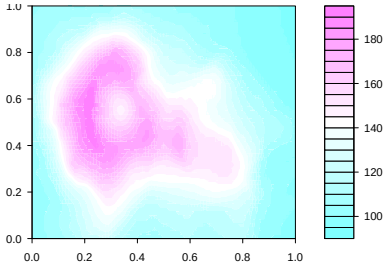
Plot Rearrangement

The option `fig.show` can decide whether to hold all plots while evaluating the code and "flush" all of them to the end of a chunk (`fig.show='hold'`), or just insert them to the place where they were created (by default `fig.show='asis'`). Here is an example of `fig.show='asis'`:

```
contour(volcano)  # contour lines
```



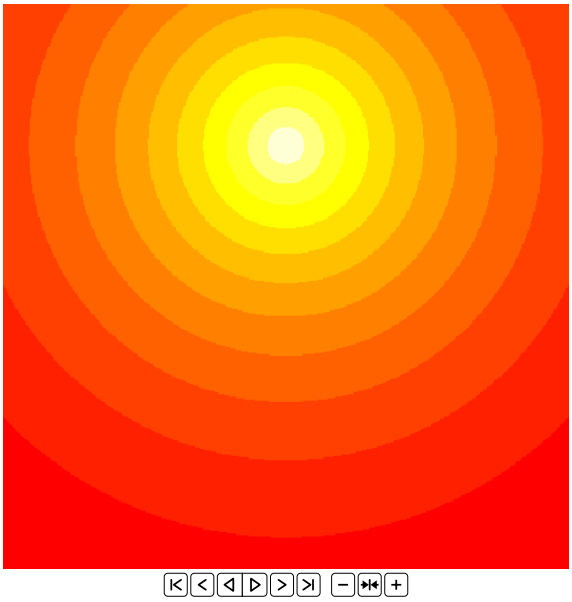
```
filled.contour(volcano)  # fill contour plot with colors
```



Beside `hold` and `asis`, the option `fig.show` can take a third value: `animate`, which makes it possible to insert animations into the output document. In LaTeX, the package **animate** is used to put together image frames as an animation. For animations to work, there must be more than one plot produced in a chunk. The option `interval` controls the time interval between animation frames; by default it is 1 second. Note you have to add `\usepackage{animate}` in the LaTeX preamble, because **knitr** will not add it automatically. Animations in the PDF output can only be viewed in Adobe Reader.

As a simple demonstration, here is a Mandelbrot animation taken from the **animation** package (**?**); note the PNG device is used because PDF files are too large. You should be able to see the animation immediately with Acrobat Reader since it was set to play automatically:

```
library(animation)
demo("Mandelbrot", echo = FALSE, package = "animation")
```



Plot Size

The `fig.width` and `fig.height` options specify the size of plots in the graphics device, and the real size in the output document can be different (see `out.width` and `out.height`). When there are multiple plots per chunk, it is possible to arrange more than one plot per line in LaTeX – just specify `out.width` to be less than half of the current line width, e.g. `out.width='.49\\linewidth'`.

The tikz Device

Beside PDF, PNG and other traditional R graphical devices, **knitr** has special support to tikz graphics via the **tikzDevice** package (**?**), which is similar to **pgfSweave**. If we set the chunk option `dev='tikz'`, the *tikz()* device in **tikzDevice** will be used to save plots. Options `sanitize` and `external` are related to the tikz device: see the documentation of *tikz()* for details. Note `external=TRUE` in **knitr** has a different meaning with **pgfSweave** – it means `standAlone=TRUE` in *tikz()*, and the tikz graphics output will be compiled to PDF *immediately* after it is created, so the "externalization" does not depend on the **tikz** package; to maintain consistency in (font) styles, **knitr** will read the preamble of the input document and use it in the tikz device. At the moment, I'm not sure if this is a faithful way to externalize tikz graphics, but I have not seen any problems so far. The assumption to make, however, is that you declare all the styles in the preamble; **knitr** is agnostic of *local* style changes in the body of the document.

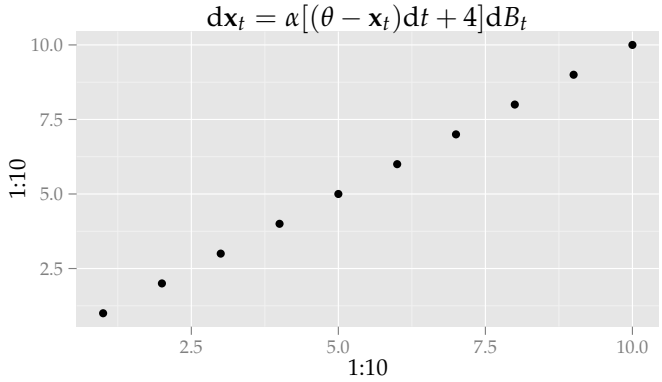
Below is an example taken from StackOverflow[3]; we usually have to write R code like this to obtain a math expression $\mathrm{d}\mathbf{x}_t = \alpha[(\theta - \mathbf{x}_t)\mathrm{d}t + 4]\mathrm{d}B_t$ in R graphics:

```
qplot(1:10, 1:10) + opts(title = substitute(paste(d * bolditalic(x)[italic(t)] ==
    alpha * (theta - bolditalic(x)[italic(t)]) * d * italic(t) + lambda *
        d * italic(B)[italic(t)]), list(lambda = 4)))
```

With the tikz device, it is both straightforward and more beautiful:

```
library(ggplot2)
qplot(1:10, 1:10) + labs(title = sprintf("$\\mathrm{d}\\mathbf{x}_{t} = \\alpha[(\\theta - \\mathbf{x}_{t
    4))
```

[3] http://stackoverflow.com/q/8190087/559676



The advantage of tikz graphics is the consistency of styles[4], and one disadvantage is that LaTeX may not be able to handle too large tikz files (it can run out of memory). For example, an R plot with tens of thousands of graphical elements may fail to compile in LaTeX if we use the tikz device. In such cases, we can switch to the PDF or PNG device, or reconsider our decision on the type of plots, e.g., a scatter plot with millions of points is usually difficult to read, and a contour plot or a hexagon plot showing the 2D density can be a better alternative (they are smaller in size).

The graphics manual contains more detailed information and you can check it out in the website.

3.3 Cache

The feature of cache is not a new idea – both **cacheSweave** and **weaver** have implemented it based on Sweave, with the former using **filehash** and the latter using .RData images; **cacheSweave** also supports lazy-loading of objects based on **filehash**. The **knitr** package directly uses internal base R functions to save (*tools:::makeLazyLoadDB()*) and lazy-load objects (*lazyLoad()*). These functions are either undocumented or marked as internal, but as far as I understand, they are the tools to implement lazy-loading for packages. The **cacheSweave** vignette has clearly explained lazy-loading, and roughly speaking, lazy-loading means an object will not be really loaded into memory unless it is really used somewhere. This is very useful for cache; sometimes we read a large object and cache it, then take a subset for analysis and this subset is also cached; in the future, the initial large object will not be loaded into R if our computation is only based on the object of its subset.

The paths of cache files are determined by the chunk option `cache.path`; by default all cache files are created under a directory `cache` relative to the current working directory, and if the option value contains a directory (e.g. `cache.path='cache/abc-'`), cache files will be stored under that directory (automatically created if it does not exist). The cache is invalidated and purged on any changes to the code chunk, including both the R code and chunk options[5]; this means previous cache files of this chunk are removed (filenames are identified by the chunk label). Unlike **pgfSweave**, cache files will never accumulate since old cache files will always be removed in **knitr**. Unlike **weaver** or **cacheSweave**, **knitr** will try to preserve these side-effects:

1. printed results: meaning that any output of a code chunk will be loaded into the output document for a cached chunk, although it is not really evaluated. The reason is **knitr** also cache the output of a chunk as a character string. Note this means graphics output is also cached since it is part of the output. It has been a pain for me for a long time to have to lose output to gain cache;

2. loaded packages: after the evaluation of each cached chunk, the list of packages used in the current R session is written to a file under the cache path named `___packages`; next time if a cached chunk needs to be rebuilt, these packages will be loaded first. The reasons for caching package names are, it can

[4]Users are encouraged to read the vignette of **tikzDevice**, which is the most beautiful vignette I have ever seen in R packages: `http://cran.r-project.org/web/packages/tikzDevice/vignettes/tikzDevice.pdf`

[5]One exception is the `include` option, which is not cached because `include=TRUE/FALSE` does not affect code evaluation; meanwhile, the value `getOption('width')` is also cached, so if you change this option, the cache will also be invalidated (this option affects the width of text output)

be slow to load some packages, and a package might be loaded in a previous cached chunk which is not available to the next cached chunk when only the latter needs to be rebuilt. Note this only applies to cached chunks, and for uncached chunks, you must always use *library()* to load packages explicitly;

Although **knitr** tries to keep some side-effects, there are still other types of side-effects like setting *par()* or *options()* which are not cached. Users should be aware of these special cases, and make sure to clearly separate the code which is not meant to be cached to other chunks which are not cached, e.g., set all global options in the first chunk of a document and do not cache that chunk.

Sometimes a cached chunk may need to use objects from other cached chunks, which can bring a serious problem – if objects in previous chunks have changed, this chunk will not be aware of the changes and will still use old cached results, unless there is a way to detect such changes from other chunks. There is an option called `dependson` in **cacheSweave** which does this job. We can explicitly specify which other chunks this chunk depends on by setting an option like `dependson='chunkA;chunkB'` or equivalently `dependson=c('chunkA', 'chunkB')`. Each time the cache of a chunk is rebuilt, all other chunks which depend on this chunk will lose cache, hence their cache will be rebuilt as well.

Another way to specify the dependencies among chunks is to use the chunk option `autodep` and the function *dep_auto()*. This is an experimental feature borrowed from **weaver** which frees us from setting chunk dependencies manually. The basic idea is, if a latter chunk uses any objects created from a previous chunk, the latter chunk is said to depend on the previous one. The function *findGlobals()* in the **codetools** package is used to find out all global objects in a chunk, and according to its documentation, the result is an approximation. Global objects roughly mean the ones which are not created locally, e.g. in the expression `function() {y <- x}`, `x` should be a global object, whereas `y` is local. Meanwhile, we also need to save the list of objects created in each cached chunk, so that we can compare them to the global objects in latter chunks. For example, if chunk A created an object `x` and chunk B uses this object, chunk B must depend on A, i.e. whenever A changes, B must also be updated. When `autodep=TRUE`, **knitr** will write out the names of objects created in a cached chunk as well as those global objects in two files named `___objects` and `___globals` respectively; later we can use the function *dep_auto()* to analyze the object names to figure out the dependencies automatically. See `http://yihui.name/knitr/demo/cache/` for examples.

Yet another way to specify dependencies is *dep_prev()*: this is a conservative approach which sets the dependencies so that a cached chunk will depend on all its previous chunks, i.e. whenever a previous chunk is updated, all later chunks will be updated accordingly.

3.4 Code Externalization

It can be more convenient to write R code in a separate file, rather than mixing it into a LaTeX document; for example, we can run R code successively in a pure R script from one chunk to the other without jumping through other texts. Since I prefer using LyX to write reports, Sweave is even more inconvenient because I have to recompile the whole document each time, even if I only want to know the results of a single chunk. Therefore **knitr** introduced the feature of code externalization to a separate R script. Currently the setting is like this: the R script also uses chunk labels (marked in the form `## ---- chunk-label` by default); if the code chunk in the input document is empty, **knitr** will match its label with the label in the R script to input external R code. For example, suppose this is a code chunk labelled as `Q1` in an R script named homework1-xie.R which is under the same directory as the Rnw document:

```
## ---- Q1 ---------------------
gcd <- function(m, n) {
    while ((r <- m%%n) != 0) {
        m <- n
        n <- r
    }
    n
}
```

In the Rnw document, we can first read the script using the function *read_chunk()*:

```
read_chunk("homework1-xie.R")
```

This is usually done in an early chunk, and we can use the chunk `Q1` later in the Rnw document:

```
<<Q1, echo=TRUE, tidy=TRUE>>=
@
```

Different documents can read the same R script, so the R code can be reusable across different input documents.

3.5 Evaluation of Chunk Options

By default **knitr** uses a new syntax to parse chunk options: it treats them as function arguments instead of a text string to be split to obtain option values. This gives the user much more power than the old syntax; we can pass arbitrary R objects to chunk options besides simple ones like `TRUE/FALSE`, numbers and character strings. The page `http://yihui.name/knitr/demo/sweave/` has given two examples to show the advantages of the new syntax. Here we show yet another useful application.

Before **knitr** 0.3, there was a feature named "conditional evaluation"[6]. The idea is, instead of setting chunk options `eval` and `echo` to be `TRUE` or `FALSE` (constants), their values can be controlled by global variables in the current R session. This enables **knitr** to conditionally evaluate code chunks according to variables. For example, here we assign `TRUE` to a variable `dothis`:

```
dothis <- TRUE
```

In the next chunk, we set chunk options `eval=dothis` and `echo=!dothis`, both are valid R expressions since the variable `dothis` exists. As we can see, the source code is hidden, but it was indeed evaluated:

```
## [1] "you cannot see my source because !dothis is FALSE"
```

Then we set `eval=dothis` and `echo=dothis` for another chunk:

```
dothis

## [1] TRUE
```

If we change the value of `dothis` to `FALSE`, neither of the above chunks will be evaluated any more. Therefore we can control many chunks with a single variable, and present results selectively.

This old feature requires **knitr** to treat `eval` and `echo` specially, and we can easily see that it is no longer necessary with the new syntax: `eval=dothis` will tell R to find the variable `dothis` automatically just like we call a function `foobar(eval = dothis)`. What is more, all options will be evaluated as R expressions unless they are already constants which do not need to be evaluated, so this old feature has been generalized to all other options naturally.

3.6 Customization

The **knitr** package is ready for customization. Both the patterns and hooks can be customized; see the package website for details. Here I show an example on how to save **rgl** plots (**?**) using a customized hook function. First we define a hook named `rgl` using the function *hook_rgl()* in **rgl**:

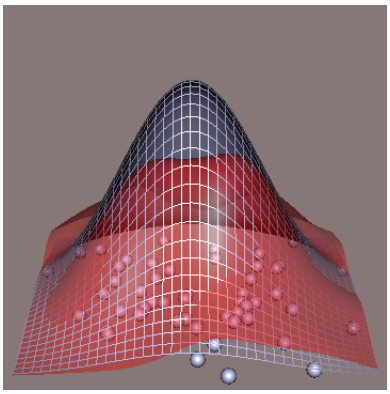
```
library(rgl)
knit_hooks$set(rgl = hook_rgl)
head(hook_rgl)  # the hook function is defined as this
```

[6] request from `https://plus.google.com/u/0/116405544829727492615/posts/43WrRUffjzK`

```
##
## 1 function (before, options, envir)
## 2 {
## 3     rglwidgetCheck()
## 4     rglwidget::.hook_rgl(before, options, envir)
## 5 }
```

Then we only have to set the chunk option `rgl=TRUE`:

```
library(rgl)
demo("bivar", package = "rgl", echo = FALSE)
par3d(zoom = 0.7)
```



Due to the flexibility of output hooks, **knitr** supports several different output formats. The implementation is fairly easy, e.g., for LaTeX we put R output in `verbatim` environments, and in HTML, it is only a matter of putting output in `div` layers. These are simply character string operations. Many demos in `http://yihui.name/knitr/demos` show this idea clearly. This manual did not cover all the features of **knitr**, and users are encouraged to thumb through the website to know more possible features.

4 Editors

You can use any text editors to write the source documents, but some have built-in support for **knitr**. Both RStudio (`http://www.rstudio.org`) and LyX (`http://www.lyx.org`) have full support for **knitr**, and you can compile the document to PDF with just one click. See `http://yihui.name/knitr/demo/rstudio/` and `http://yihui.name/knitr/demo/lyx/` respectively. It is also possible to support other editors like Eclipse, Texmaker and WinEdt; see the demo list in the website for configuration instructions.

About This Document

This manual was written in LyX and compiled with **knitr** (version 1.13). The LyX source and the Rnw document exported from LyX can be found under these directories:

```
system.file("examples", "knitr-manual.lyx", package = "knitr")  # lyx source
system.file("examples", "knitr-manual.Rnw", package = "knitr")  # Rnw source
```

You can use the function *knit()* to knit the Rnw document (remember to put the two .bib files under the same directory), and you need to make sure all the R packages used in this document are installed:

```
install.packages(c("animation", "rgl", "tikzDevice", "ggplot2"))
```

Feedback and comments on this manual and the package are always welcome. Bug reports and feature requests can be sent to `https://github.com/yihui/knitr/issues`, and questions can be delivered to the mailing list `https://groups.google.com/group/knitr`.