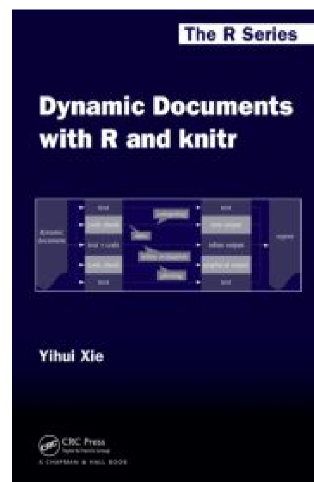
Dynamic Documents with R and knitr
Summary

RK

December 5, 2013

Abstract

The purpose of this document is to summarize the main points from the book "Dynamic Documents with R and knitr".



Background

When you use **Sweave** for the first time, you really appreciate its utility. It's like you are given the tools to think, code, reflect, tweak the code, write down your observations about the code and output, all in one document. No distractions, No switching softwares. I had thoroughly enjoyed doing literate programming in the last few years. However you reach a point when you think **Sweave** package should have had some convenient features. I was always not happy with the code chunk for including graphics. If I use **ggplot2** package, I had to write `print` so as to include the figure in the report. However if it was a base graphics, I could just include it with out any such additional statement. Also to have control on the size of figure, I had to fall back on LaTeX. Shouldn't there be an option in the code chunk that combines these two and make it easier for the user ? That's when I stumbled on to **knitr** and I am thrilled after using it. In this document, I will try to summarize the main points from the book on knitr, written by the package author, Yihui Xie

Preface

There are dangers in doing analysis at a separate place and then tabulating all the results in to a report manually.

- it is error-prone due to too much manual work.
- it requires lots of human effort to do tedious jobs such as copying results across documents.
- the workflow is barely recordable especially when it involves GUI (Graphical User Interface) operations, therefore it is dif- ficult to reproduce.
- a tiny change of the data source in the future will require the author(s) to go through the same procedure again, which can take nearly the same amount of time and effort.
- the analysis and writing are separate, so close attention has to be paid to the synchronization of the two parts.

In this book, dynamic documents refer to the kind of source documents containing both program code and narratives. Sometimes we may just call them source documents since "dynamic" may sound confusing and ambiguous to some people(no interactivity or animations). The book follows the convention :

- **package** - bold text for package.
- *plot()* - italics for function names.
- `apply(y,1,sum)` - typewrite font for inline text.
- figure/foo.pdf - sansseriff font for figures.

I will keep in mind these conventions when I do create my own RR(Reproducible Research) documents.

Introduction

The basic idea behind dynamic documents stems from literate programming, a programming paradigm conceived by Donald Knuth (Knuth, 1984). The original idea was mainly for writing software: mix the source code and documentation together; we can either extract the source code out (called tangle) or execute the code to get the compiled results (called weave). A dynamic document is not entirely different from a computer program: for a dynamic document, we need to run software packages to compile our ideas (often implemented as source code) into numeric or graphical output, and insert the output into our literal writings (like documentation).

Literate programming paradigm has two tasks:

1. write program code to do computing, and
2. write narratives to explain what is being done by the program code.

The traditional approach to doing the second task is to write comments for the code, but comments are often limited in terms of expressing the full thoughts of the authors. Normally we write our ideas in a paper or a report instead of hundreds of lines of code comments. Technically, literate programming involves three steps:

1. parse the source document and separate code from narratives.
2. execute source code and return results.
3. mix results from the source code with the original narratives.

These steps can be implemented in software packages, so the authors do not need to take care of these technical details. Instead, we only control what the output should look like.

Reproducible Research

Dynamic report generation is a step towards RR, as the latter encompasses many more activities. If you had taken a particular seed for simulation, you can generate a dynamic report and be done with it. However may be the seed turned to be a lucky seed and the inferences are actually incorrect. These type of aspects cannot be ensured by merely creating a dynamic report. Hence one must carefully understand the limitations of Dynamic reports, i.e. what can be done via RR ? and what RR doesn't do ?

A few good practices for RR:

- Manage all source files under the same directory and use relative paths whenever possible - This was something that I was not doing at all.
- Do not change the working directory after the computing has started. I always do this and I realize now that it is a bad practice.
- Compile the documents in a clean R session.
- Avoid the commands that require human interaction. My analysis has never required human interaction.
- Avoid environment variables for data analysis. I have taken care of this, I think. But at least now I am consciously aware of it.
- Attach *sessionInfo()* and instructions on how to compile the document - I have never done this till date.

I just got to know about (Rpubs). I hope to use in the days to come.

A First Look

The knitr package is a general-purpose literate programming engine - it supports document formats including LATEX, HTML, Markdown, and programming languages such as R, Python, awk, C++, and shell scripts.
Two basic examples are shown in this chapter, one that converts a `.Rnw` in to `.pdf` and the second a `.Rmd` in to `.html`. Just to understand how publishing works, I have posted it on (Rpubs).If you have a `R` code, you can quickly convert in to a report using the *stitch()* command. Also given a `.Rnw`, you can extract the code using *purl()* command.

Editors

This chapter gives information about various editors that can be configured so that working with **knitr** becomes a pleasant experience. It mentions RStudio, LyX,ESS, Tinn-R,Texmaker, Eclipse, TextMate,TEXShop, and Vim. I am happy to know that I can configure StatET to work with **knitr**. Will get it installed on StatET sometime soon.

Document Formats

The three components of **knitr** are

- a source parser.
- a code evaluator.
- an output renderer.

The parser parses the source document and identifies computer code chunks as well as inline code from the document; the evaluator executes the code and returns results; the renderer formats the results from computing in an appropriate format, which will finally be combined with the original documentation.
The pattern for beginning of a chunk in `Rnw` is a regular expression

```
all_patterns$rnw$chunk.begin

## [1] "^\\s*<<(.*)>>=.*$"
```

The chunk options can be any piece of `R` code that is compatible with the kind of value that option expects. You can write a code that results in `TRUE/FALSE` and then assign to it option `eval`. This is very flexible as compared to **Sweave** where you can't use code to set option values.

Chunk labels are supposed to be unique id's in a document, and they are mainly used to generate external files such as images and cache files. If two non-empty chunks have the same label, knitr will stop and emit an error message, because there is potential danger that the files generated from one chunk may override the other chunk. If we leave a chunk label empty, **knitr** will automatically generate a label of the form `unnamed-chunk-i`, where `i` is an incremental chunk number. One can also set chunk options globally, for example,`opts_chunk$set (echo = FALSE)`, does not echo code for any chunk. You can override this setting for individual chunk though. `<<>>=` denotes opening of a code chunk and `@` denotes close of a code chunk and opening of documentation chunk.

In a **Sweave** document, the start and end of code syntax is `<<*>>=` and and `@`. Inline code syntax is `\Sexpr{}`. In a mardown document, the start and end of code syntax is ` ```{r *} ` and and ` ``` `. Inline code syntax is `` `r x` ``. The chapter talks about markdown language and praises its simplicity that anyone who has ever written an email can create a markdown document. It also goes on to list a host of derivative markdown packages that have appeared. RStudio used **markdown** package for its implementation. It also mentions **Pandoc**, that is usually called the *swiss knife* of document conversion as it supports the conversion of markdown in to a host of formats. **knitr** will parse the code, be it a a chunk or inline fragment and then renders it to the appropriate output using various output hooks.

```
grep("^render_", ls("package:knitr"), value = TRUE)

## [1] "render_asciidoc" "render_html"     "render_jekyll"   "render_latex"
## [5] "render_listings" "render_markdown" "render_rst"      "render_sweave"
```

`render_sweave()` uses the default **Sweave** output whereas `render_listings()` decorates the output. So, based on type of output you want, you can configure the setting. This chapter explains the set of functions for various hooks such as *plot*, *chunk*, *inline*, *document* so that the hooks produce appropriate content for the chosen output format. There is also a *spin()* function that takes plain simple R code that has been commented using roxygen and converts in to LaTeXor markdown document.

Text Output

knitr default rounding is 4 digits and numbers $> 10^{-5}$ will be shown in scientific notation. Here are some of the main points mentioned in the context of text output.

- `eval` - A boolean that controls whether the code should be shown asis or evaluated.
- `tidy` - Specify whether the code should be tidied up or not?

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

```
for (k in 1:10) {
    j = cos(sin(k) * k^2) + 3
}
```

- `tidy.opts` - width.cutoff to specify the max width.
- options in `tidy`

```
names(formals(formatR::tidy.source))
##  [1] "source"             "keep.comment"       "keep.blank.line"
##  [4] "replace.assign"     "left.brace.newline" "reindent.spaces"
##  [7] "output"             "text"               "width.cutoff"
## [10] "..."
```

- code decoration

```
# highlight = FALSE
x <- rnorm(5)
var(x)
## [1] 2.139
```

```
x <- rnorm(5)
var(x)
## [1] 0.8289
```

- A very painful problem is solved, i.e. copy pasting the code from RR documents Before **knitr** , I used to see code like this

```
> x <- rnorm(5)
> var(x)
[1] 0.9248
```

but with **knitr** the default is this

```
x <- rnorm(5)
var(x)
## [1] 0.2604
```

The output is put in as comments which makes it easy to copy paste the code. Superb feature.

- font sizes - footnotesize, small, large, and Large, The default is normal size.

```
# footnotesize
x <- rnorm(5)
```

```
# small
x <- rnorm(5)
```

```
# normal
x <- rnorm(5)
```

```
# large
x <- rnorm(5)
```

```
# Large
x <- rnorm(5)
```

- `echo` - Should the code chunk be shown. Note that this has nothing to do with the evaluation of the code chunk
- `results` - asis value for this option is very useful. When you need to produce tables using `R`, you can use asis option so that raw output is dumped on to the document.
- `warnings/error/message` - All these can be set as a boolean value based on whether you want to see them in the report. They can also be set at a global level.

```
<<>>=
opts_chunk$set (warning = FALSE, message = FALSE)
@
```

- To stop on error, use `opts_knit$set (stop_on_error = 2L)`. I gather from the author's blog that this option is deprecated. Wow! already things in the book are deprecated. I wonder how long will the functionality of the other options get deprecated. I hope not, for there is no point in wasting time learning and building muscle memory for things that go in to oblivion quickly. Anyway, now the preferred way to stop **knitr** is to use `error=FALSE`
- For tables, you can use **xtable** package and set `results="asis"`
- There are about 80 themes shipped via **knitr**

```
head(knit_theme$get(), 20)
##  [1] "acid"        "aiseered"    "andes"       "anotherdark" "autumn"
##  [6] "baycomb"     "bclear"      "biogoo"      "bipolar"     "blacknblue"
## [11] "bluegreen"   "breeze"      "bright"      "camo"        "candy"
```

```
## [16] "clarity"     "dante"       "darkblue"     "darkbone"     "darkness"
```

These themes work for LaTeXand HTML output. The link on the web, themes, contains a preview of all themes.

Graphics

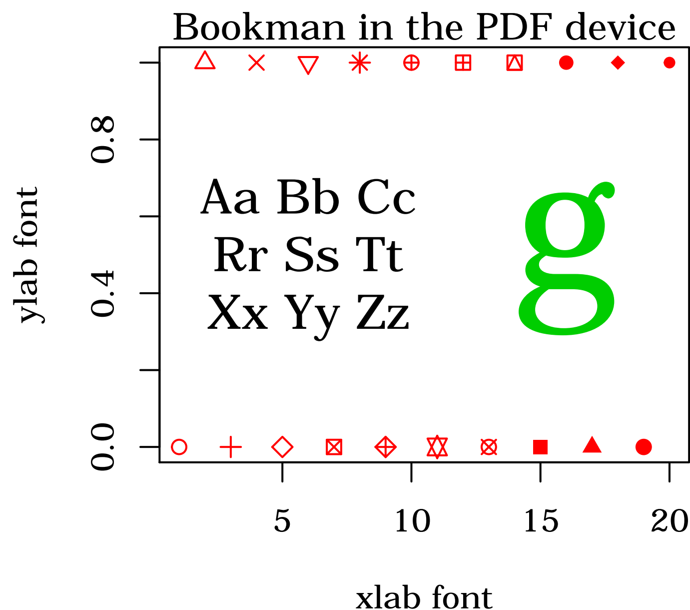
Yippee!,finally reached the section for which I was looking forward to read. Firstly no need of *print(p)* to include a **ggplot(2)** visual. The default device for Rnw documents is PDF and for Rmd/Rhtml/Rrst documents, it is PNG because normally PDF does not work in HTML output. Some of the options for graphics output are

- `fig.width,fig.height`

```
library(ggplot2)
p <- qplot(carat, price, data = diamonds) + geom_hex()
p
```

- `dev.args`

```
plot(rep(0:1, 10), pch = 1:20, col = 2, xlab = "xlab font", ylab = "ylab font")
mtext("Bookman in the PDF device", side = 3, cex = 1.2)
text(6, 0.5, "Aa Bb Cc\nRr Ss Tt\nXx Yy Zz", cex = 1.5)
text(16, 0.5, "g", cex = 6, col = 3)
```



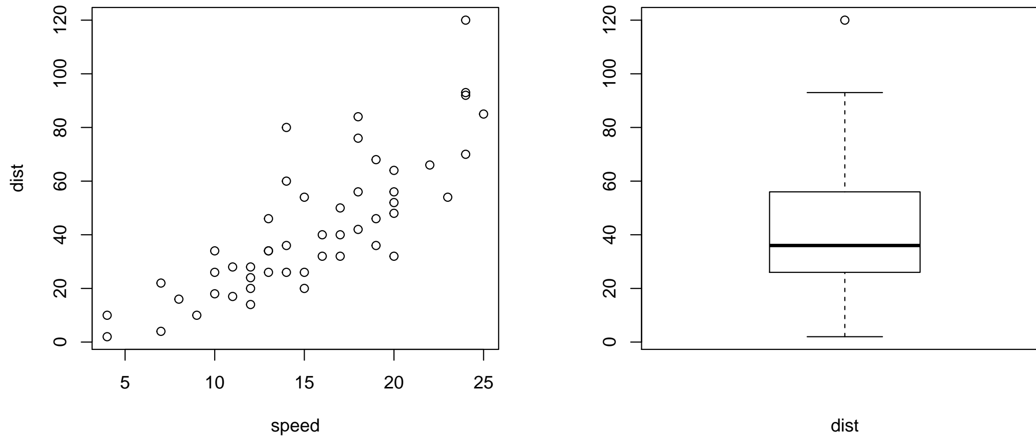
- `encoding`

```
pdf.options(encoding = "CP1250")
```

- `dingbats` - To reduce the size of scatter plots

- The great thing about **knitr** is that if the chunk has three plots, the plots are shown in the report with out any additional effort. This is unlike **Sweave** where only the recent plot is maintained.

```
## two plots side by side (option fig.show='hold')
plot(cars)
boxplot(cars$dist, xlab = "dist")
```



- One thing I am really impressed with graphics options is that I can do away with the LaTeXcode that I had to always put in after a **Sweave** code chunk. With the range of options available, I can forget about that task.
- The fact that all the figures automatically go to a specific directory is so much nicer as it will not clutter the workspace of `Rnw` files.
- code chunk name should be unique. I was following a crazy naming convention of using fig1, fig2 etc. in my **Sweave** code. Now I can just forget about all that and let **knitr** take care of all those things. Having said that, may be it makes sense to follow some naming convention for all the figures. I think I will specify `fig.ext` so that I know that a identify the figures for various chapters.
- `fig.env,fig.pos,fig.scap,fig.lp` - All these are built in as chunk options. So, there is no need to add LaTeXcode.
- `out.width` - This controls the The `fig.width` and `fig.height` options specify the size of plots in the graphical device, and the real size in the output document can be different and can be specified by `out.width`

Cache

This is another section of the book that I was eagerly awaiting to get to. Most of the times my **Sweave** documents take a long time to run, mainly because some of my code chunks take a long time to run. I knew there was a way around in **Sweave** to use caching but somehow never found time to learn about it. The other day I was working on a document and realized that there was no option but to cache various chunks. I used a funny work around, in fact an ugly work around of placing all code chunks in separate child **Sweave** files. Needless to say, organizing the whole thing was a nightmare. **knitr** promised a convenient way and indeed it is so easy to incorporate caching.

The basic ideas of caching is that a chunk will not be re-executed as long as it has not been modified since the last run, and old results will be directly loaded instead.

- `cache.path` - The directory where all the cache files will be stored
- There are some techniques mentioned in the book when a cache needs to be updated in the case of version change in `R` or an external file change.
- The chapter explains the limitations of packages like **weaver** and **cacheSweave**, i.e. they do not save side effects. However **knitr** caches side effects like graphics output, loaded packages, random seed.
- By default the option to cache is set to false in every code chunk. To set this option globally one can use a code chunk with

  ```
  <<>>=
   opts_chunk$set(cache.path="cache/graphics-)"
  @
  ```

- code chunk dependency is another thing that one needs to be careful about.

  ```r
  x <- 1
  ```

  ```r
  y <- x + 2


  y + 5
  ## [1] 8
  ```

  The dependency is necessary because `chunkC` uses the object y that was created in `chunkB`, and `chunkB` needs the value of x created in `chunkA`. When x in the first chunk is changed, the latter two chunks have to be updated accordingly.
- Instead of explicitly specifying the dependency, there is also an experimental feature, i.e. setting `autodep = TRUE`.

Overall I am so happy after reading this chapter on caching. Now I can use all these features and save a ton of my time in running RR reports.

Cross Reference

This chapter begins with the idea of chunk reuse. One can reuse whole chunks by following `ref` option The section on organizing child documents was very useful to me. The inclusion of child `Rnw` documents can be done by

```
<<D, child="chapt1.Rnw">=
@
```

Learnt a way to apply preamble to the child documents

```
<<parent, include=FALSE>=
set_parent ( "master.Rnw" )
@
```

All these aspects were very messy via `Sweave`. Indeed **knitr** makes things in RR very appealing.

Hooks

A hook is a userdefined R function to fulfill tasks beyond the default capability of **knitr**. This chapter deals with *chunk* hooks. A chunk hook is a function stored in `knit_hooks` and triggered by a custom chunk option.

- Create chunk hooks A chunk hook can be arbitrarily named, as long as it does not clash with existing hooks. It can then be triggered by the label name

```
<<>>=
knit_hooks$set(margin = function(before, options, envir){
  if(before)
    par(mar=c(4,4,0.1,0.1)) else NULL
})
@

<<margin=TRUE>>=
par(bg="gray")
plot(1:10)
@
```

- Three arguments to the chunk - **before**(whether it is called before the chunk or after the chunk),**options**(list of chunk options), **envir**(environment in which the chunk exists)
- Hooks and chunk options - One can define the chunk hook to non `NULL` globally

```
opts_chunk$set(margin = TRUE)
```

- One can add arguments to chunk hook

```
knit_hooks$set(margin = function(before, options, envir) {
    if (before) {
        m <- options$margin
        if (is.numeric(m) && length(m) == 4L) {
            par(mar = m)
        }
    } else NULL
})
```

- one can also write output from a chunk hook.

Out of the all examples mentioned in this chapter, I think the most useful one for my work is cropping of a figure via chunk hook.

Language Engines

This chapter talks about using **knitr** with other languages such as Python, Ruby, Haskell, awk/gawk, sed, shell scripts, Perl, SAS, TikZ, Graphviz and $C++$, etc. Not relevant for my future work and hence skipped the contents.

Tricks and Solutions

The author says in his blog :

> I do not have much to say about this book: almost everything in the book can be found in the online documentation, questions & answers and the source code. The point of buying this book is perhaps you do not have time to read through all the two thousand questions and answers online, and I did that for you.

This chapter is mainly a collection of important tricks that you can use for RR. Here are the some of tricks I will definitely use in my RR activity(the actual list in the book is long and the usage depends on purpose of RR one is creating):

- using `option` aliasing :

  ```
  set_alias(w = "fig.width", h = "fig.height")
  ```

- using `opt_template` : This is definitely a pain area for me that the package addresses. There are multiple times when you want to have different sizes for graphics and now you can do it conveniently as follows

  ```
  opts_template$set(fig.large = list(fig.width = 7, fig.height = 5),
      fig.small = list(fig.width = 3.5, fig.height = 3), fig.rk = list(fig.width = 4,
          fig.height = 4))
  ```

  ```
  <<opts.label= "fig.rk">>
  plot(1:10)
  @
  ```

- Pushing all the code to the appendix. This is useful for some type RR documents where you want to put all the code in the appendix. neat hack!
- For the width of source code and text output, it is controlled by the global option width in `options()`. Alternatively use `\lstset{breaklines=true}`.
- Message colors

  ```
  \definecolor{messagecolor}{rgb}{0, 0, 0}
  \definecolor{warningcolor}{rgb}{1, 0, 1}
  \definecolor{errorcolor}{rgb}{1, 0, 0}
  ```

- Beamer : Beamer is a popular document class to create slides with LaTeX. Using **knitr** in beamer slides is not very different from other LaTeXdocuments; the only thing to keep in mind is that we need to specify the fragile option on beamer frames when we have verbatim output.
- suppress long output.
- publishing external images with the HTML file.
- extract source code.
- Dealing with reproducible simulation where there is a cache setting for a chunk.
- Turning package documentation in to html files.

Publishing Reports

When your RR is ready for publication, it is always better to set `message` and `warning` to `FALSE`. Using RStudio to produce pdf of html is via a simple click of a button. One can also use Pandoc to convert a markdown document to LaTeX, HTML, rtf, epub, word doc, open document text etc. Basically pandoc is

called swiss knife of document conversion and hence its always better to spend some time learning the basic aspects of it. The easiest way to create a HTML5 presentation is to create a markdown via RStudio and then run pandoc. This will be useful for those who want to embed code and text in to the presentation. These days it has become quite common for people to blog `R` code and the chapter presents two ways to do it. One is via Jekyll and other via the usual wordpress route. The infra needed to push documents to either of these places is provided in **knitr**. The thing with wordpress is that modification is a pain once you have published. With Jekyl it is said that it is far easier to maintain. But tell me who has time to edit a blog post? Publishing itself takes time and I would rather do it on wordpress and be done with it. Well, may be if you want users to check in and correct your code, probably Jekyl is a good way to go. I have never hosted code on Jekyl. Will try it sometime soon.

Applications

This chapter mentions four applications. First is *Doing Homeworks*. Indeed with minimal configuration, one can turn in high quality document will all the code and narrative at one place. The second application mentioned is *websites and blogs* built on **knitr**. Third application is very interesting, *creating vignettes*. As things stand, these are created using **Sweave** and there are no HTML vignettes on CRAN. Hopefully as things move, CRAN will allow HTML vignettes that are so much more convenient for bookmarking, sharing etc. The fourth application mentioned is *writing books*. I think this is similar to what is happening in the IPython world where people are writing books in an ipython document and sharing it via sites like nbviewer. The chapter mentions a few sites where **knitr** has been used

- Statistical Sleuth
- The Analysis of Data - HTML and pdf versions

Other Tools

This chapter talks about other tools such as **Sweave**, Dexy, IPython, Orgmode. Since I have been using **Sweave** for some time, I found this chapter extremely useful as it lists down almost all the differences between **Sweave** and **knitr**. This kind of information all at one place is so useful. I mean I might have had to go through a ton of posts on stackoverflow or other places to get this info. Let me list down some of the main differences here.

- `concordance` changed
- `keep.source` now becomes `tidy`
- `print` is redundant
- `strip.white` is dropped
- `prefix` is dropped
- `prefix.string` now becomes `fig.path` and has many more options
- `eps,pdf` options are dropped and new device option introduced for 20 graphical devices
- `fig` now becomes a set of options to control a ton of aspects of graphics. I love this part.
- `width, height` now become `fig.width` and `fig.height`
- `SweaveOpts` and `SweaveInput` deprecated.Instead one needs `opts_chunk$set()` and `child`

What are some of the problems that **knitr** solves ?

- empty figure chunks give errors in **Sweave**. no longer in **knitr**
- no need to use the painful `print` command for image display.
- you can customize individual width of each figure in the output. In **Sweave** there is no straightforward way.
- multiple figures from one figure chunk do not work by default in **Sweave**.
- it is easy to produce HTML output.

Takeaway

Imagine that you were using a clunky and a painful email service and suddenly one day you are shown gmail. ArenÂŠt you thrilled ?. It's elegant, quick and has a ton intuitive features. I had the same feeling with **knitr** after having painfully used **Sweave** for a long time. I am certain that this package will stand out as the goto package for literate programming for a very long time to come because it is elegant, quick and has features that you were always trying to patch in via other packages. Should you read this book? Well, if you have the patience and time to go over the manual and a thousand posts from stackoverflow and other places to know the various features of the package, you don't need this book. However if you are like me who is short of time, values content that is organized and prefers to know the key hacks from the package, this book is definitely worth it.