# Code Chunks: Comparing Sweave and Knitr

**Paul E. Johnson**, Director, CRMDA <pauljohn@ku.edu>

**keywords**: Rmarkdown, Codechunks, R, documents.
See `https://crmda.ku.edu/guides` for updates.

KU
CENTER FOR RESEARCH METHODS & DATA ANALYSIS
College of Liberal Arts & Sciences

**Feb. 19, 2019**

**Abstract**

This is about code chunks and interacting with statistical software via a document.

## Introduction: Terminology

> A code chunk is a runable piece of R code. Re-producing the document will re-run calculations.

Code chunk technology is beneficial because the risk of mismatch between the commentary in a paper and the results being discussed is reduced.

The document family offered by the `stationery` package for R a number of code-chunk worthy document templates. We can write in either `Rmarkdown` (file format: `Rmd`) or `noweb`/LaTeX (file format: `Rnw`) to produce formal report documents or informal guides for educational presentations. There are two chunk management "engines" that can be used with `noweb`/LaTeX documents, `Sweave` and `knitr`, while for `Rmarkdown` documents the only code chunk engine is `knitr`. The output format may be either HTML for Web pages or portable document format (PDF).

As a result of the "mixing and matching" among document preparation formats, output formats, and processing technologies, we have a fairly large family of document types among which to choose. We refer to the in and output formats by the a shorthand, "input2output", as in "rmd2html" to mean an `Rmd` document that will end up as a Web page. The formats that we are preparing at this time are:

1. Guides
   - rmd2html-guide
   - rmd2pdf-guide
   - rnw2pdf-guide-knit
   - rnw2pdf-guide-sweave
2. Reports
   - rmd2pdf-report
   - rnw2pdf-report-knit
   - rnw2pdf-report-sweave
3. Slides
   - rnw2pdf-report-sweave

When a document is prepared in `noweb`/LaTeX or `Rmarkdown`, a document will be converted through several formats to arrive in either HTML or PDF. If everything "just works", then authors might not

worry too much about success at each individual stage. In my experience, more elaborate documents almost never "just work". Understanding the sequence of transitions can help in correcting problems.

This document is about one of the earliest stages in document processing. The transition from `Rnw -> tex`, or from `Rmd -> md`, will scan the document for "code chunks", send them to R, and then the results might be put to use in the document. This is called "weaving" or "knitting", depending on which chunk-processing function is used. The older, traditional method is called `Sweave`, while the newer engine is called `knitr`.

## Enter "code chunks"

Suppose a professor is writing about the psychology of adolescence or social conflict in Uganda. In the "old" paradigm, the professor will do some statistical analysis in SAS or SPSS and the copy/paste output into the document. This is a tedious, error prone process. Documents produced in this way are difficult to keep up-to-date and difficult to proof read.

Instead of cutting and pasting, we instead insert code chunks that are *run during document preparation*. The advantages of this are obvious. The statistical code and results never become "out of step" with the final document. If the raw data is revised somehow, we no longer repeat the "old" fashioned process of re-running the analysis and then "copy/pasting" the results into a revised document. Instead, we run through the document preparation steps again and the results are *automatically* updated and revised.

Code chunks are allowed in both `noweb`/LaTeX and `Rmarkdown` documents intended for either HTML or PDF. In R, the original method for creating code is called "Sweave". A code chunk is created by a somewhat cumbersome notation that begins with "<<>>=" and ends with "@". Here is a code chunk that runs a regression model.

```
<<>>=
lm(y ~ x, data = dat)
@
```

The chunks in `noweb`/LaTeX documents can be processed by `knitr` as well. In `Rmarkdown` documents, `knitr` is the only chunk-processing technology that is available. In `Rmarkdown` documents, the outer boundaries are changed to three back ticks, along with squiggly braces and the letter "r", which designates the language being processed (`knitr` can handle several languages).

```
```{r}
lm(y ~ x, data = dat)
```
```

These are simple, unnamed chunks. In actual usage, the beginning of the chunk usually has several additional arguments, including a name for the chunk. Names are helpful because error messages will later report the name of the failed chunk. For that reason, we suggest, as a general policy, that *chunks should be named*.

```
<<mychunk10>>=
lm(y ~ x, data = dat)
```

```
@
```

```
```{r mychunk10}
lm(y ~ x, data = dat)
```
```

The options, which control if the code is displayed in the document, or if the output is included, and so forth, can be specified after the chunk's name.

#### If you don't want code chunks

Some documents may not use the code chunks. But it is nice to know they are available when we need them.

## Chunk options in common between knitr and Sweave

Whether we use knitr or Sweave to handle code chunks, the same chores must be handled. The key options available in both frameworks are

1. echo: if TRUE, the R code will be displayed along with the output
2. include: the chunk is evaluated, but neither the echo nor the results are displayed
3. eval: if FALSE, the chunk is not sent to the R session, but it is checked for correctness of code.
4. fig: if TRUE, the chunk creates a graphic to be saved (usually pdf or png)
5. results: indicate if results should be included, possibly specifying details

In the documentation for Sweave, the original chunk processor, the options are declared by short names `T` or `F`, while in the time since then the R Core generally suggests writing the full names `TRUE` and `FALSE`. In documentation about `knitr`, authors are much more likely to write out the full names `TRUE` and `FALSE`.

`knitr` differs from `Sweave` in a few interesting ways that we will explore as we go. `knitr` has different codes for document-wide options. It also has a richer set of options to control functions to handle the document processing. One benefit of knitr is that it can be used to write about other kinds of programs. I've used it to write about the BASH shell, for example.

Rather than going through all possible chunk arguments, we now survey (and give examples) of the chunk variations that we require in documents, for any frontend or backend. We need the ability to create chunks that:

1. are not evaluated, but are displayed "beautifully" to the reader, with syntax highlighting.
2. are not displayed, but are evaluated, and the results may (or may not) be displayed for the reader.
3. create graphics, which are automatically included in the document.
4. create graphics (or other files) that are not automatically included in the document. Graphics are saved in image files that can be inserted at a different part in the document. I call this the "save something for later" approach.
5. import previous chunks for re-analysis.

This document is prepared with `Rmarkdown`. The chunks are converted from `Rmd` to `md` by the `knitr` functions.

Because this document is using the PDF backend, we are able to use the LaTeX `listings` package to display the results. `listings` is a highly customizable framework that can beautify the code and output displays.

The listings display is set to show code chunks in a light gray background with a monospace typewriter font. I prefer to not display the R prompt ">" in the listings display for code input. In addition, contrary to the `Rmarkdown` default, I do not want the R comment symbol, `##`, at the beginning of every line of output.

## Tour of knitr code chunks

1. A chunk that is evaluated, echoed, both input and output. This is a standard chunk, no chunk options are used:

```
   ```{r chunk10}
   set.seed(234234)
   x <- rnorm(100)
   mean(x)
   ```
```

   The user will see both the input code and the output, each in a separate box:

```
set.seed(234234)
x <- rnorm(100)
mean(x)
```

```
[1] -0.1004232
```

   Notice the code highlighting is not entirely successful, as the function `set.seed` is only half-highlighted.

2. A chunk with commands that are echoed into the document, but not evaluated (`eval=F`).

```
   ```{r chunk30, echo=F}
   set.seed(234234)
   x <- rnorm(100)
   ```
```

   The user will not see any code that runs, but only a result box:

```
   ```{r chunk20, eval=F}
   set.seed(234234)
   x <- rnorm(100)
   mean(x)
   ```
```

When the document is compiled, the reader will see the depiction of the code, which is (by default) beautified and reformatted:

```
set.seed(234234)
x <- rnorm(100)
mean(x)
```

3. A chunk that is evaluated, with output displayed, but code is not echoed (`echo=F`). It is not necessary to specify `eval=T` because that is a default.

````
   ```{r chunk30, echo=F}
   set.seed(234234)
   x <- rnorm(100)
   ```
````

The user will not see any code that runs, but only a result box:

```
[1] -0.1004232
```

4. A hidden code chunk. A chunk that is evaluated, but neither is the input nor output displayed (`include=F`)

````
   ```{r chunk40, include=F}
   set.seed(234234)
   x <- rnorm(100)
   mean(x)
   ```
````

Here's what happens when that is processed:

What is the grammatically correct way to say "did you see nothing?" You should not even see an empty box? After that, the object `x` exists in the on-going R session, it can be put to use.

## Chunks with graphics in knitr

5. A chunk that creates a graph, and allows it to be inserted into the document, but the code is not echoed.

````
   ```{r chunk50, fig=T, fig.height=3, fig.width=4,
      fig.show="hold", echo=F}
    hist(x, main = "One Histogram Displayed Inline")
   ```
````

The result of that should simply be a graph inserted into the final document, with no signal to the reader that it was produced by a code chunk.

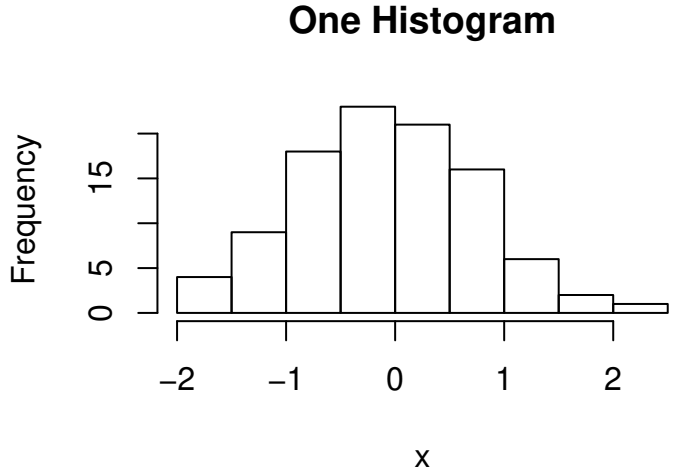
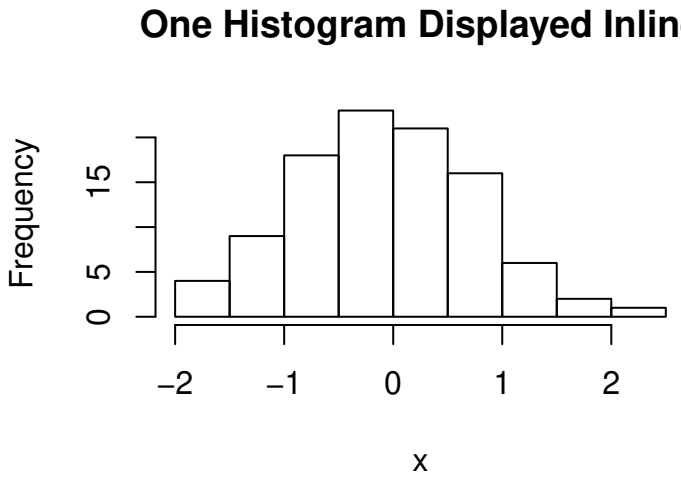



Figure 1: A Floating Histogram




6. Save a graph in a file and display it at another point in the document.

   A. First, I demonstrate feature unique to the PDF backend. PDF documents have features for "floating" tables and figures and these can be used. A chunk that creates a graph and provides a caption will cause a float to be created.

```
   ```{r chunk60, fig=T, fig.height=3, fig.width=4,
      fig.show="hold", echo=F, fig.cap = "\\label{fig:hfloat}A
      Floating Histogram"}
   hist(x, main = "One Histogram")
   ```
```

   Observe in the output we have a numbers, floating figure. Because we inserted a label with the caption, we can cross-reference Figure 1. Figure numbers will be adjusted automatically as new figures are added before and after this chunk. This feature is not available in HTML.

   B. For HTML output, the best we can do is save a graph, but does not allow it to be displayed immediately.

```
   ```{r chunk65, fig=T, fig.height=3, fig.width=4,
      fig.show="hide", dev=c("pdf", "png")}
   hist(x, main = "Another Histogram")
   ```
```

Why do this? In HTML, we don't have the option to create figure floats, but the next-best option is to show code for a figure, but prevent its inclusion in the document by setting `fig.show="hide"`.

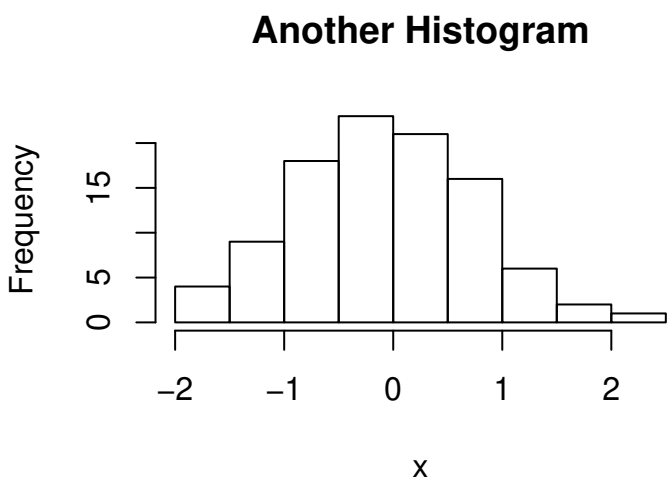
In this code chunk, we asked for output in 2 formats, one as png and one as pdf.

Now that the image file exists, we can insert it manually, because the pdf was saved in a file named "tmpout/p-chunk60-1.pdf". I chose the name "tmpout" in the template, that is configurable. But I like that name pretty much. Here, for example, is an HTML table in which I have embedded that image

If we want to insert the figure with markdown tools, we could. This is the code that would typically be used. `![Another Histogram](tmpout/p-chunk65-1.pdf)`

This is inflexible because the image cannot be rescaled. Also, note that markdown does not have a concept of "centering" or "alignment" of a figure, so to a significant extent we have to focus on tools in the backend document language. Here we suggest using backend-specific code to include–and scale–the figure. This inserts the figure using LaTeX terminology in a PDF document—

```
\includegraphics[width=3.5in]{tmpout/p-chunk65-1.pdf}
```




Here we are happy to see that the *raw LaTeX* code for inserting graphics does work (but do not understand why).

If we are targeting an HTML backend, we would write a similar request in HTML code, using the file `tmpout/p-chunk65-1.png`:

```
<img src="tmpout/p-chunk65-1.png" alt="a floating histogram"
   width="308"
height = "216">
```

Here, we are guessing that the end user's screen will tolerate an image that is 308 pixels wide. If I were writing in HTML at the moment, I'd try harder on sizing that appropriately. But this is a PDF document and this is just an example of what an HTML author might do.

7. One chunk that shows a series of commands. This is an example of a feature in the `knitr` chunk-processing framework. It is not directly accessible from Sweave, but it can be achieved by some careful coding. The `knitr` code chunk option will . It is possible to display the whole graph created by the series of commands.

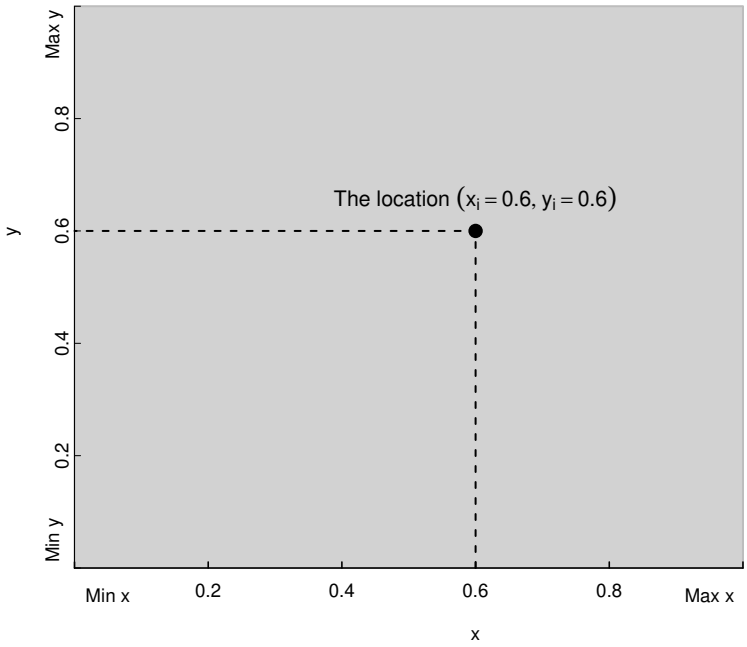
In this case, we demonstrate the usual R plotting exercise in which a "blank" plot is created, and then lines, points, and labels are added one by one.

```
plot(c(0, 1), c(0, 1), xlim = c(0,1), ylim = c(0,1), type = "n",
   ann = FALSE, axes = F)
rect(0, 0, 1, 1, col = "light grey", border = "grey")
axis(1, tck = 0.01, pos = 0, cex.axis = 0.6, padj = -3, lwd = 0.8,
     at = seq(0, 1, by = 0.2), labels = c("", seq(0.2,0.8, by=0.2),
        ""))
axis(2, tck = 0.01, pos = 0, cex.axis = 0.6, padj = 3, lwd = 0.3,
     at = seq(0, 1, by = 0.2), labels = c("", seq(0.2,0.8, by=0.2),
        ""))
mtext(expression(x), side = 1, line = 0.5, at = .6, cex = .6)
mtext(expression(y), side = 2, line = 0.5, at = .6, cex = .6)
mtext(c("Min x", "Max x"), side = 1, line = -0.5, at = c(0.05,
   0.95), cex = .6)
mtext(c("Min y", "Max y"), side = 2, line = -0.5, at = c(0.05,
   0.95), cex = .6)
lines(c(.6, .6, 0), c(0, .6, .6), lty = "dashed")
text(.6, .6, expression(paste("The location ",
                              group("(",list(x[i] == .6, y[i] ==
                                 .6),")"))), pos = 3, cex = .7)
points(.6, .6, pch = 16)
```

I want to run that code from top to bottom, but I don't want to retype it. Both Sweave and knitr allow one to retrieve code from a chunk and put it to use again. This demonstrates the knitr method, which uses the chunk option `ref.label`.

```
```{r chunk75, ref.label='chunk71', echo=F, fig=T, fig.keep="last",
   collapse=T, fig.width=5, fig.height=5}
```
```

This produces one figure, which happens to illustrate the Cartesian plane (between 0 and 1 on both axes):




However, we might be in teaching mode and we need to demonstrate the effect of each successive R function in the process of creating the figure. If one is using `Sweave`, one will have to write many separate chunks, each to accomplish each stage. In comparison, the `knitr` engine has a special option, `fig.keep`, which instructs the system to keep a snapshot of each separate image in the creation of this figure.

```
```{r chunk76, ref.label='chunk71', echo=F, fig=T, include=F,
   fig.keep="all", collapse=T, fig.width=4, fig.height=4}
```
```

A quick check of the tmpout directory shows that this code created several graphs. Observe:

```
list.files("tmpout", pattern="p-chunk76.*pdf")
```

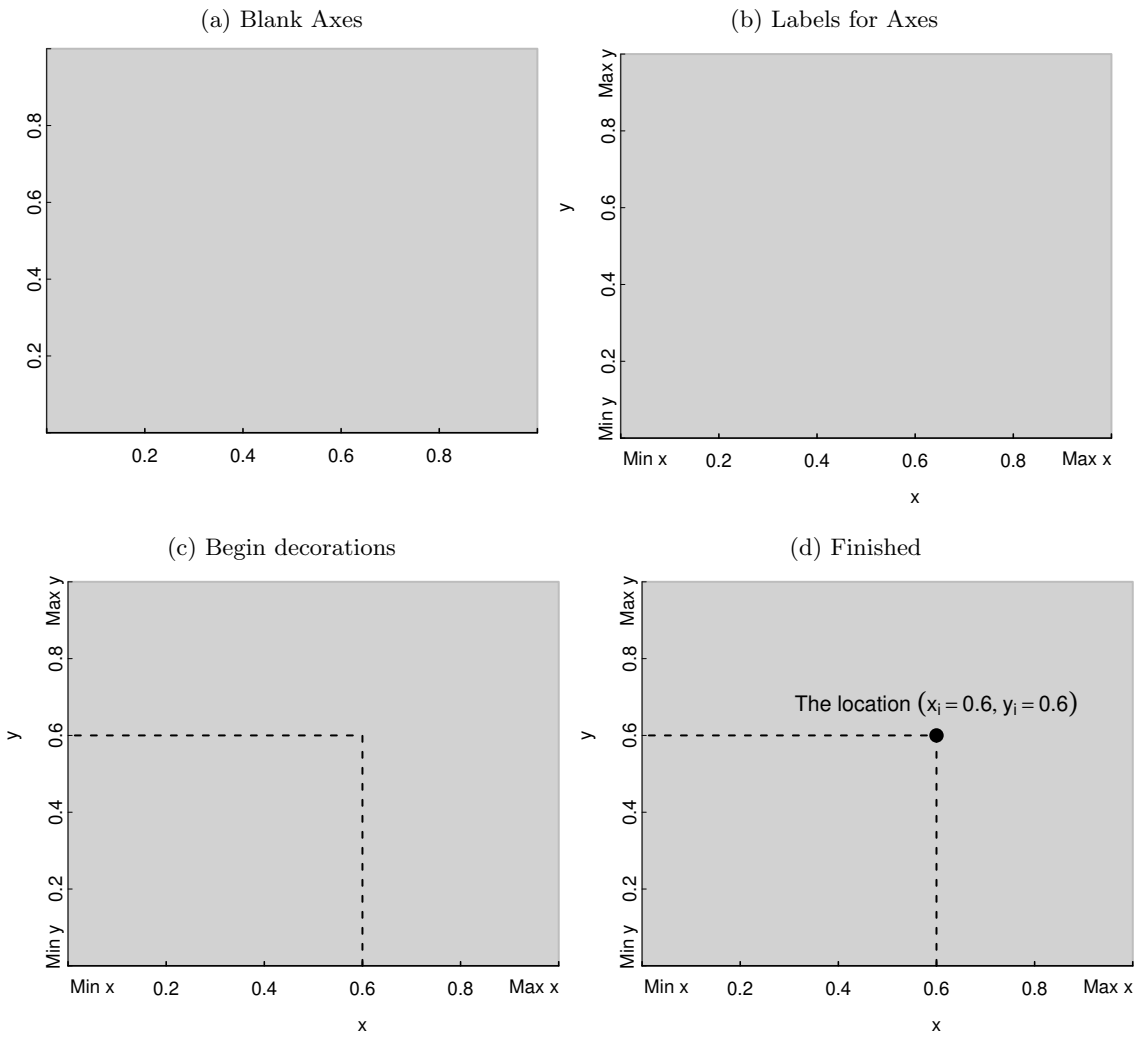
```
 [1] "p-chunk76-1.pdf"  "p-chunk76-10.pdf" "p-chunk76-11.pdf"
 [4] "p-chunk76-2.pdf"  "p-chunk76-3.pdf"  "p-chunk76-4.pdf"
 [7] "p-chunk76-5.pdf"  "p-chunk76-6.pdf"  "p-chunk76-7.pdf"
[10] "p-chunk76-8.pdf"  "p-chunk76-9.pdf"
```

There is a separate output file for each stage in the transition, `p-chunk-76-1.pdf` through `p-chunk-76-11.pdf`. It is a bit tricky to display a matrix of figures "automatically", but I'll use a LaTeX
structure to do this work.

## Chunks with tables

8. "Made to fit" analysis tables.

Figure 2: Four snapshots



If the backend is HTML, we probably need R functions that will write HTML code that can slide into our document *as is*. If the backend is PDF, we probably need a table in LaTeX markup. I say *probably* because sometimes the compilation fails, even if we have the expected formats, because there are little wrinkles

### Other special features of `knitr`

#### Chunk caching

This is "SLOW!" That's a common complaint about compiling a document that has substantial code chunks. It is SLOW. The delay discourages authors from following my advice to "compile early, compile often."

When I am writing in LaTeX, I have found 2 strategies to deal with the slowness caused by processing R chunks. First, I have sometimes created "document branches". If we put all of the R code in a document branch, and then flag that branch as "inactive", then the document can be compiled without re-processing all of the R code. Second, I have used a simpler "two document" strategy. I have one `noweb`/LaTeX
file with code chunks. It has nothing else, no commentary. Using the `split=T` flag, that document can be compiled to run the R code. The run creates separate code files for each of the chunks. Then I have a second document that inputs those chunks where desired. In order to compile the document, then, it is not necessary for me to re-run all of the calculations. Both of these approaches require some planning and it is important to run the code-producing branch (or file) when generating a final version of the document.

Neither of my workaround features are available for Rmarkdown authors, in large part because the `split=T` document option is not allowed in Rmarkdown. In fairness, I admit that the "homemade" document workarounds that I use may be difficult to administer (especially for novices). The `Rmarkdown` and `knitr` authors have employed an alternative strategy known as "code caching". When a chunk is processed, a copy of the calculations can be saved and used next time. The "magic trick", if it works perfectly, will never re-run a chunk unless it is necessary. In the implementation, my experience has been somewhat mixed, but quite a bit of effort has been made by the package authors.

In an Rmarkdown document that reports on extensive simulations, I would almost certainly avoid embedding the code in the document and I'd write a separate document that can conduct calculations and export tables and graphs. This is more difficult in Rmarkdown than it is in PDF documents, but it can be done.

## Sweave

One document can be handled only with one chunk-handling framework. I have much more experience with Sweave and generally prefer it, but recognize the fact that the young crowd likes `knitr`, so this was prepared with `knitr`.

At the beginning of an Sweave document, we can specify document-wide paramters. Here is a standard piece that I use in noweb/LaTeX
documents.

```
\SweaveOpts{prefix.string=tmpout/t, split=TRUE, ae=FALSE, height=5,
   width=6}
```

Here are the highlights:

1. The `prefix.string` parameter specifies that output files will be placed in a folder `tmpout` and the first letter will be `t`. The user can, of course, change the prefix to any letter. (Because each document can have a different prefix, it is possible then to have several document that output files into the same output folder.)
2. `split=TRUE` asks Sweave to keep each chunks input and output results in a separate file. This is very handy, as will be explained below.
3. Turn off the `ae` feature set. Those features were created long ago and they interfere with more modern packages.

After the R session is started, one of the first chunks will create the `tmpout` directory, the collector of output graphs and code chunks:

```
<<echo=F>>=
if(!dir.exists("tmpout"))dir.create("tmpout", recursive = TRUE)
@
```

Some R users may not have noticed, but all R sessions include a large array of default settings that control display of warnings, the command prompt, reporting of decimals, and so forth. The `options` function is used to control the line length of output and the display of the command prompt. There is also an object called par, which sets defaults for plots. I almost always have a chunk that adjusts both `par` and `opt`.

```
<<Roptions, echo=F, include=F>>=
opts.orig <- options()
options(device = pdf)
options(width=160, prompt=" ", continue="  ")
options(useFancyQuotes = FALSE)
set.seed(12345)
par.orig <- par(no.readonly=TRUE)
pjmar <- c(5.1, 5.1, 1.5, 2.1)
options(SweaveHooks=list(fig=function() par(mar=pjmar, ps=10)))
pdf.options(onefile=FALSE, family="Times", pointsize=10)
@
```

Save something for later

Use `include=F` to save something for later. This avoids some output problems that result from the one-size-fits-all tendency of the "completely automatic" document builder. In the usual `weave''documentation, a user is told to type in a chunk and then the output plops into the documentright` there". Sometimes, it is nicer to create the chunk outputs and

figures, and then insert them later. This gives the author much more control over the size and position of graphs (and it is why we turn off `ae`). I learned this trick from Duncan Murdoch in the r-help email list.

This chunk would create a figure named `tmpout/t-chunkfig.pdf`. The chunk will not be displayed in the document, but it will still create a code chunk output file because `echo=T`.

```
<<chunkfig, include=F, echo=T, fig=T>>=
## R code for figure here
 @
```

A file named `tmpout/t-chunkfig.pdf` will be created in the `tmpout` directory. (Recall `t-` was designated as the prefix).

Now that the chunk was created, we are free to insert the graphic wherever we want in the usual LaTeX way. For example,

```
\includegraphics[width=5in]{tmpout/t-chunkfig}
```

Note `.pdf` is not incldued with the file name.

The echo result created a file named `tmpout/t-chunkfig.tex` and we insert that wherever we want with the LaTeX input macro. Interestingly, we need to provide the full name, including `.tex`:

```
\input{tmpout/t-chunkfig.tex}
```

There are many R functions that can create LaTeX markup. This is especially useful for tables. If a chunk's output is text with LaTeX
markup, and we want that tex to go "into the document immediately, right here," then we add the chunk argument `results=tex`. However, it is also possible to use the "save something for later" approach.

```
<<chunktable, include=F, results=tex>>=
# R code here
@
```

That creates a file named `tmpout/t-chunktable.tex`, which I would insert into the document whenever I like with

```
\input{tmpout/t-chunktable.tex}
```

Why do this? Why separate chunk output creation from inclusion in a document? I can put the chunk in wherever I want, but more importantly I might that chunk in a different document. It is very convenient to come along later sort through the output in the tmpout folder. I might need to make a separate slide show document displaying the same tables and/or figures and this makes it easy to do that.

The automatic "stick this output in where the chunk is placed" approach works great with lecture notes and guides because these things are easy to update and re-run.

In conclusion, the "automatic" "self-documenting" report is almost never exactly correct. It requires some "finger painting", if only to regain control of the position and size of figures. By saving the chunks, we give ourselves a degree of control.

#Regression Tables

There are many programs for creating regression tables. Every author will have to test to find out if the results are desirable.

Here we test the `outreg` function in the rockchalk package. `outreg` can generate output in either LaTeX or HTML, and the former is needed for a document that is compiled into a PDF document.

```
set.seed(234234)
dat <- data.frame(x1 = rnorm(100), x2 = rnorm(100), y = rnorm(100))
library(rockchalk)
m1 <- lm(y ~ x1, data = dat)
m2 <- lm(y ~ x1 + x2, data = dat)
vl <- c("x1" = "Excellent Predictor", "x2" = "Adequate Predictor")
outreg(list("First Model" = m1, "Second Model" = m2), varLabels =
   vl,
      tight = FALSE, type = "latex")
```

| | First Model | | Second Model | |
|---|---|---|---|---|
| | Estimate | (S.E.) | Estimate | (S.E.) |
| (Intercept) | -0.042 | (0.108) | -0.062 | (0.110) |
| Excellent Predictor | 0.069 | (0.130) | 0.058 | (0.130) |
| Adequate Predictor | _ | | 0.092 | (0.095) |
| N | 100 | | 100 | |
| RMSE | 1.074 | | 1.075 | |
| $R^2$ | 0.003 | | 0.012 | |
| adj $R^2$ | -0.007 | | -0.008 | |

$*p \leq 0.05$** $p \leq 0.01$***$p \leq 0.001$

There are many other regression-table-making functions available today. I made some lecture notes about it for the R summer workshops that we offer at KU (`http://pj.freefaculty.org/guides/Rcourse/regression-tables-1`).

In the CRMDA, we have devoted a good deal of effort to the creation of a program that can create good-looking tables to summarize structural equation models (SEM). Please try the function `semTable` in the `kutils` package (when finished, that function will probably be moved to a package named `semTable`).

#Listings, not Verbatim

The Sweave default system uses the LaTeX Verbatim class to offer input and output chunks in the document. A better approach, pioneered by Frank Harrell (Vanderbilt) uses the LaTeX class `listings`. Harrell prepared a replacement for R's Sweave.sty and called it Sweavel.sty. I've used that for many years. Because we made a few minor customizations, that style file is now called `kureport.sty`.

We are using the listings class to display input and output chunks in this document. In the preamble, we have some special settings that control the color of text, the background, and so forth. The listings class has an elaborate settings framework.

The following is a listings display. It is not Sweaved, it is simply a LaTeX listings environment colored by the settings in the preamble

```
x <- rnorm(100)
y <- rpois(100, lambda = 2)
plot(y ~ x)
```

## Session Info

Session Information is usually not written into a report, but in a guide file we regularly will include it as follows.

```
R version 3.5.2 (2018-12-20)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 18.10

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.8.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.8.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

other attached packages:
[1] psych_1.8.12      pander_0.6.3      rockchalk_1.8.140
[4] stationery_0.98.6

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.0      knitr_1.21      magrittr_1.5    splines_3.5.2
 [5] kutils_1.64     MASS_7.3-51.1   mnormt_1.5-5    lattice_0.20-38
 [9] pbivnorm_0.6.0  xtable_1.8-3    minqa_1.2.4     carData_3.0-2
[13] stringr_1.3.1   highr_0.7       plyr_1.8.4      tools_3.5.2
[17] parallel_3.5.2  grid_3.5.2      nlme_3.1-137    xfun_0.4
```

```
[21] tinytex_0.10     htmltools_0.3.6 yaml_2.2.0       lme4_1.1-19
[25] digest_0.6.18    lavaan_0.6-3    Matrix_1.2-15    zip_1.0.0
[29] nloptr_1.2.1     evaluate_0.12   rmarkdown_1.11   openxlsx_4.1.0
[33] stringi_1.2.4    compiler_3.5.2  stats4_3.5.2     foreign_0.8-71
```