


# Using RStudio's "Compile Report" Command

---

 [aliesdataspace.com/2018/12/using-r-studio-s-compile-report](https://aliesdataspace.com/2018/12/using-r-studio-s-compile-report)

December 1, 2018

2018-12-01 in [R](#)

When I have anything R-related (ex. data, an analysis, some results, etc) that I want/need to discuss with somebody (ex. my supervisor, somebody willing to help me with my stats, etc), I often use the "Compile Report" command in RStudio in order to turn an R script into a an html document that has the code and output all together. This is really handy, because not only do you have your plots right there to show results, but if you want to discuss details about how you calculated those results, you also have the code right there as well. It's also a quick and easy way to compile different types of results, such as plots and simple summary data / counts / etc.

Info about how the "Compile Report" command actually works, and how to call the same command within the script itself / if you're not using RStudio, can be found on [the R markdown website](#). What I want to give you here is just a simple overview of a few key syntax notes that can get you started in turning .R scripts into useful, snazzy-looking reports.

## Grid 3

Now change the grid resolution to be finer...

```
x <- seq(-40,60,by=0.1) #grid resolution set in the 'by=' arg (was by=1)
y <- seq(-60,60,by=0.1) #grid resolution set in the 'by=' arg (was by=1)
xy <- expand.grid(x=x,y=y)
xy.sp <- SpatialPoints(xy)
gridded(xy.sp) <- TRUE

#calculate the ranges
ud <- lapply(sp, kernelUD, grid=xy.sp)

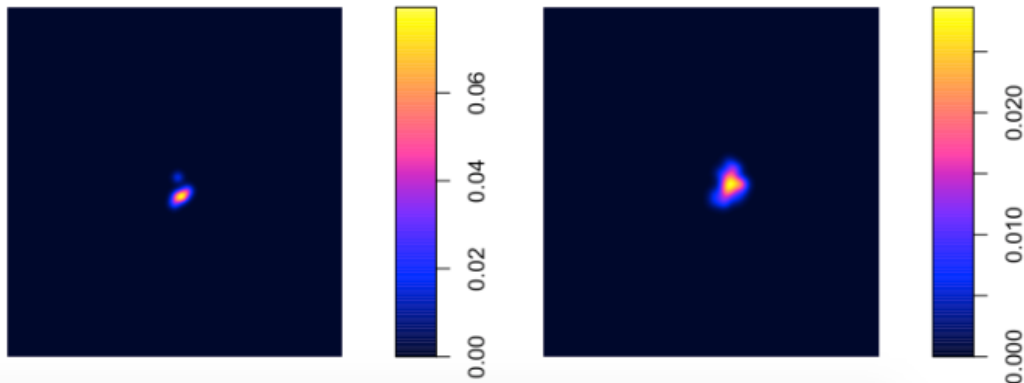
#sum the @data$ud slots
for (i in 1:4) {
  print(sum(ud[[i]]@data$ud))
}
```

```
## [1] 99.98728
## [1] 99.98737
## [1] 99.88627
## [1] 99.98732
```

Using the previous grid extents, the sums were all around 1, now they are all approaching 100. i.e. making the grid resolution finer by a factor of 100 (/10 on each axis), increased the sum of `@data$ud` by a factor of 100.

So now plot these ranges to look at them.

```
for (i in 1:4) {
  plot(ud[[i]], xlim=c(-20,25), ylim=c(-35,50))
  Sys.sleep(3)
}
```



## If you are familiar with R markdown syntax...

... then all you really need to know is - in your plain .R script - add a `#'` (hash apostrophe) to the start of every line that you don't want in a code block (so text, titles, etc), and then after that, just use the regular R markdown syntax. Code chunk (and in-code-chunk comments) should just be written as usual in the .R script (no need for the ````{r}` code chunk start and end syntax).

Code chunk options that would normally go ````{r HERE}` can also be used: just start the line with `#+` (instead of `#'`), and then specify your options, separated by commas.

## If you are not familiar with R markdown syntax...

---

... then the most important thing to know is that you'll need to play around with this a bit to really get it, but once you do, it's super simple and easy. Basically, write your code as usual. But any comments, section breaks, etc, should be formatted as follows...

```
#' turns that text into plain text
```

```
#' # Header 1
```

```
#' ## Header 2
```

```
#' ### Header 3
```

```
#' using stars around words makes them italics or bold
```

```
# a regular hash comment, without an apostrophe, will appear as a comment in the code block
```

```
#' *** horizontal break
```

The above syntax, in a regular R script, will - in the final 'Compile Report' command's html output - look like...

turns that text into plain text

### Header 1

---

### Header 2

---

### Header 3

---

using stars around words makes them *italics* or **bold**

```
# a regular hash comment, without an apostrophe, will appear as a comment in the code block
```

---

You can also make a YAML header, just like in R markdown, by starting each line with `#'` ...

```
#' ---  
# title: "This is an Example"  
# author: "Alie"  
# date: "November 1, 2018"  
# ---
```

The above syntax, in a regular R script, will - in the final 'Compile Report' command's html output, will look like...

# This is an Example

---

*Alie*

*November 1, 2018*

In the header, you can specify other optional arguments, such as the output format ( `output` - html, pdf, or word doc), if there should be a table of contents ( `toc` ), and what theme you want ( `theme` - this will change the colours of your code chunks background and text).

```
#' ---
#' title: "This is an Example"
#' author: "Alie"
#' date: "November 1, 2018"
#' output:
#'   html_document:
#'     toc: true
#'     highlight: haddock
#' ---
```

Note the indent spacing of the 3 last code lines in the YAML header above - this indentation matters.

You can also specify options that are applied to different parts of your code and that affect how it is run and/or displayed in the compiled report. If you read up on R markdown (for example, check out this [Rmarkdown reference guide](#)), you can find many many different options that you can use. All you need to do to implement these options, is start the line with `#+` (instead of `#'` ). For now, here are a few of the options I find I use the most...

```
##+ messages = FALSE <<this will suppress the printing of any messages (for example,
those that you get
#                               when you load a package) from printing in your output

##+ echo = FALSE <<this will run the code, and show the output, but it won't show the
code itself in the
#                               final report (for example, if you want to show a plot, but not
show the code used to generate
#                               the plot)

##+ eval = FALSE <<this will not run the code, not show the output, but it will show
the code (for
#                               example, if you want to show some code that takes a long time to
compute, so you
#                               don't want the compile report command to run it every time)

##+ fig.height = 3, fig.width = 3 <<these options set the output figure height and
width
```

A couple of notes about options:

1. As was done in the `fig.height` and `fig.width` example, several options can be passed at the same time, just separate them with commas.
2. The option(s) that you set will apply to all subsequent code until the next line that starts with `#'` or `#+` .

## Some final thoughts

---

What I love about generating html reports using this Compile Report command (versus just straight up writing an Rmarkdown document) is the versatility and flexibility of working with just a simple R script. While .Rmd is great for a final report, especially something with a lot of text, just using a plain .R script file (with a few extra characters for output formatting) feels less restrictive - especially when I want to generate an html report of a script that is still a work-in-progress. I use this format the most for meetings with my supervisor and/or collaborators - when I want to show preliminary results (i.e. the script itself is a work in progress), and I'm there myself to actually explain things (i.e. not a lot of text needed anyways). The syntax is quick and simple and doesn't require much more thought than any other .R script, so it's easy to include in any script, even if you're not sure if you'll ever want/need to compile it as a report.

## Resources

---

[R markdown website](#)

[Official Spin Documentation](#)

## Appendix: A Proper Example

---

The following is part of a script I wrote when a collaborator and I were trying to figure out the best way to scale and combine orangutan utilization distributions (home ranges) for an analysis. I was trying to gain a better understanding of how exactly `adehabitatHR`'s `kernelUD` function works by working with some simple simulated data. These documents can hopefully give you an idea of the syntax needed for certain aspects of basic formatting in an html output.

Here is the compiled html report (or open in its own window using [this link](#)).

The html output

And here is the .R script that made it...

```

#' ---
#' title: "Playing with Utilization Distribution (UD) scaling"
#' author: "Alie"
#' date: "2018-10-09"
#' output:
#'   html_document:
#'     toc: true
#'     highlight: haddock
#' ---
#'
#+ setup, include=FALSE
require(adehabitatHR)
require(raster)
require(scales)

resetPar <- function() {
  dev.new()
  op <- par(no.readonly = TRUE)
  dev.off()
  op
}

#' ## Start by generating some data
#'
#' Generate SpatialPoints data for 4 different UD.
set.seed(100)
sp <- list(
  #small sample, small spread
  SMLn_SMLsd = SpatialPoints(cbind(rnorm(n = 10, mean = 4, sd=2),
                                   rnorm(n = 10, mean = 4, sd=2))),

  #large sample, small spread
  LGn_SMLsd = SpatialPoints(cbind(rnorm(n = 30, mean = 7, sd=2),
                                   rnorm(n = 30, mean = 7, sd=2))),

  #small sample, large spread
  SMLn_LGsd = SpatialPoints(cbind(rnorm(n = 10, mean = -2, sd=10),
                                   rnorm(n = 10, mean = -2, sd=10))),

  #large sample, large spread
  LGn_LGsd = SpatialPoints(cbind(rnorm(n = 30, mean = 10, sd=10),
                                   rnorm(n = 30, mean = 10, sd=10)))
)

#' Take a look at the points.
#'
#+ echo = FALSE
par(mfrow=c(1,1))
plot(sp[[1]], col=1,
      xlim=c(-5,20), ylim=c(-25, 30),
      asp=1,
      pch=16,
      axes=T, xlab="X", ylab="Y")
for (i in 2:4) {

```

```

    plot(sp[[i]], col = i, pch=16, add=T)
  }
  legend("topright", c("SMLn_SMLsd", "LGn_SMLsd", "SMLn_LGsd", "LGn_LGsd"),
        col=c(1:4), pch=16)

#' ## Playing around with the grid
#' ### Grid 1

#make the grid
x <- seq(-40,40,by=1.5)
y <- seq(-40,40,by=1.5)
xy <- expand.grid(x=x,y=y)
xy.sp <- SpatialPoints(xy)
gridded(xy.sp) <- TRUE

#calculate the ranges
ud <- lapply(sp, kernelUD, grid=xy.sp)

#sum the @data$ud slots
for (i in 1:4) {
  print(sum(ud[[i]]@data$ud))
}
#' All UDs sum to the almost exactly the same value, regardless of sample size (n)
and
#' spread (sd).
#'
#'
#' ### Grid 2
#'
#' Now change the grid extent and see what happens...
x <- seq(-40,60,by=1) #grid x min and x max
y <- seq(-60,60,by=1) #grid y min and y max
xy <- expand.grid(x=x,y=y)
xy.sp <- SpatialPoints(xy)
gridded(xy.sp) <- TRUE

#calculate the ranges
ud <- lapply(sp, kernelUD, grid=xy.sp)

#sum the @data$ud slots
for (i in 1:4) {
  print(sum(ud[[i]]@data$ud))
}
#' Again, all UDs sum to the almost exactly the same value, regardless of spread and
#' sample size. But, changing the grid has changed what value the UDs sum to (bigger
grid,
#' higher value, it would seem).
#'
#' ### Grid 3
#'
#' Now change the grid resolution to be finer...
x <- seq(-40,60,by=0.1) #grid resolution set in the 'by=' arg (was by=1)
y <- seq(-60,60,by=0.1) #grid resolution set in the 'by=' arg (was by=1)
xy <- expand.grid(x=x,y=y)

```

```

xy.sp <- SpatialPoints(xy)
gridded(xy.sp) <- TRUE

#calculate the ranges
ud <- lapply(sp, kernelUD, grid=xy.sp)

#sum the @data$ud slots
for (i in 1:4) {
  print(sum(ud[[i]]@data$ud))
}
#' Using the previous grid extents, the sums were all around 1, now they are all
#' approaching 100. i.e. making the grid resolution finer by a factor of 100 (/10 on
each
#' axis), increased the sum of `@data$ud` by a factor of 100.

#' So now plot these ranges to look at them.
#+ fig.width=3.5, fig.height=2.5
for (i in 1:4) {
  plot(ud[[i]], xlim=c(-20,25), ylim=c(-35,50),
       main = names(ud)[i])
}

#' Note how the scale bar is diff for each UD! Thus, from range 1 to range 4, the
high
#' density areas are "worth" less and less (shallower peaks) in order to maintain the
#' equal sum of `@data$ud` across wider home ranges. Therefore, this is NOT a good
value
#' to use for UD comparisons, because larger UDs will end up having smaller values
(as an
#' artifact of the math, not the actual density).
#'
#' To illustrate this even more clearly, force the colour scales to be the same
(using the
#' `zlim` arg), then more wider-spread (`LGsd`) ranges almost dissappear...
#+ fig.width=3.5, fig.height=2.5
for (i in 1:4) {
  plot(ud[[i]], xlim=c(-20,25), ylim=c(-35,50), zlim=c(0,.08),
       main = names(ud)[i])
}

#' ## Comparing UD pixel values

#' Generate random points in the grid that cluster around where the ranges overlap
the
#' most (for the sake of having nice plots, without too many (0,0) points).
pts <- SpatialPoints(cbind(rnorm(n = 100, mean = 5, sd=5), rnorm(n = 100, mean = 5,
sd=5)))
#' Try some different methods for extracting and comparing pixel values...
#'
#' #### Using raw UD data
#'
#' Look at some UD pixel value comparisons - eventhough this method is prob not the
right
#' one, but just to see what happens...

```



```

#'
#plot the points
par(resetPar(), bg="white")
plot(sp[[1]], col=1, xlim=c(-5,20), ylim=c(-20, 30), asp=1, pch=16, axes=T)
for (i in 2:4) {
  plot(sp[[i]], col = i, pch=16, add=T)
}
plot(pts, col="orange", pch=4, add=T)
legend("topright", c("SMLn_SMLsd", "LGn_SMLsd", "SMLn_LGsd", "LGn_LGsd", "100 random
points"),
      col=c(1:4, "orange"), pch=c(rep(16, times=4), 4))

#' Now extract the value of each UD at the random `pts`...
rst <- lapply(ud, raster)
vals <- lapply(rst, extract, pts)
df.raw <- as.data.frame(matrix(unlist(vals), 100, 4, byrow=FALSE))
names(df.raw) <- c("SMLn_SMLsd", "LGn_SMLsd", "SMLn_LGsd", "LGn_LGsd")
plot(df.raw, pch=16, cex=0.7, asp=1)
#' Super teeny tiny numbers on the axes, if set asp=1, can barely see any variation
in
#' relationships between ranges with different degrees of spread (`SMLsd` vs `LGsd`).
#' Ok, try something else....
#'
#' ### Using inverted volume UD data
#'
#' Continue with the UDs from the last section, but now use the volume function
before
#' extracting values...
vud <- lapply(ud, getvolumeUD)

#take a look at it
#+ fig.width=3.5, fig.height=2.5
for (i in 1:4) {
  plot(vud[[i]], xlim=c(-20,25), ylim=c(-35,50),
      main = names(vud)[i])
}

for (i in 1:4) {
  print(sum(vud[[i]][[1]]))
}
#' The scale bar doesn't change (so peaks are 'equal' even with different sample
sizes (n)
#' and spreads (sd). Notice how the sum of the volume is less for wider spread data
#' (because there are fewer 100 cells), but is not so affected by sample size. This
is
#' good... now we are getting somewhere....
#' To make these values more intuitive (for me with my current goals, higher use =
higher
#' value), change all cell values to their inverse.

inverse <- function(x) {
  max(x) - x
}

inv.vud <- vud

```

```

for (i in 1:4) {
  inv.vud[[i]]@data$u <- inverse(vud[[i]]@data$u)
}

##+ fig.width=3.5, fig.height=2.5
#now look at the plots again
for (i in 1:4) {
  plot(inv.vud[[i]], xlim=c(-20,25), ylim=c(-35,50),
       main = names(inv.vud)[i])
}
#' Now the volume values are inverted - so a higher value means higher use - and the
range
#' of values is independent of the spread (sd) of the points, i.e. high peaks in the
#' different ranges are 'equally high'.
#'
#' Now extract the values at our random `pts` again...
rst.vol <- lapply(inv.vud, raster)
vals.vol <- lapply(rst.vol, extract, pts)
df.vol <- as.data.frame(matrix(unlist(vals.vol), 100, 4, byrow=FALSE))
names(df.vol) <- c("SMLn_SMLsd", "LGN_SMLsd", "SMLn_LGsd", "LGN_LGsd")
plot(df.vol, pch=16, cex=0.7, asp=1)
#' Values extracted from the volume UD's yield easier to interpret plots and visible
#' relationships between values in ranges with different spreads (`SMLsd` vs `LGsd`).
#'
#' ### Using scaled raw UD values
#'
#' Using the original UD's (before the volume conversion) `@data$u` slot, scale the
#' values in this slot for each range so that the values range from 0 to 1...
#'
#duplicate the UD objects, then replace the @data$u with a rescaled version of
itself
ud.rscl <- ud
for (i in 1:4) {
  ud.rscl[[i]]@data$u <- rescale(ud[[i]]@data$u, to=c(0,1))
}

##+ fig.width=3.5, fig.height=2.5
#take a look at it...
for (i in 1:4) {
  plot(ud.rscl[[i]], xlim=c(-20,25), ylim=c(-35,50),
       main = names(ud.rscl)[i])
}
#' Notice how the scale bar is the same for all ranges. i.e. high peaks in the
different
#' ranges are 'equally high'.
#'
#' Now extract the values at the random `pts` and plot them...
rst.rscl <- lapply(ud.rscl, raster)
vals.rscl <- lapply(rst.rscl, extract, pts)
df.rscl <- as.data.frame(matrix(unlist(vals.rscl), 100, 4, byrow=FALSE))
names(df.rscl) <- c("SMLn_SMLsd", "LGN_SMLsd", "SMLn_LGsd", "LGN_LGsd")
plot(df.rscl, pch=16, cex=0.7, asp=1)
#' Again, these rescaled values are easier to interpret, and the relationships
between
#' ranges with different spreads are visible and not artificially flattened.

```

```

#'
#' ## Zoom in on UD comparisons
#'
#' Now subset these final data frames to just look at points that are essentially
within
#' the first (small sample size, small spread, `SMLn_SMLsd`) range's 99%iso (but for
the
#' sake of simplicity, I won't actually calculate the range and put points in there,
I'll
#' just use points for which the values are already calculated, and the `SMLn_SMLsd`
#' values are >0).
#'
df.ls <- list(raw = subset(df.raw, SMLn_SMLsd > 0),
              vol = subset(df.vol, SMLn_SMLsd > 0),
              rscl = subset(df.rscl, SMLn_SMLsd > 0))
#'
par(resetPar())
#' And look at the relationship between **`SMLn_SMLsd` and `LGn_SMLsd`**
#+ fig.width=9, fig.height=3.5
par(mfrow=c(1,3))
for (i in 1:3) {
  plot(SMLn_SMLsd~LGn_SMLsd, data=df.ls[[i]],
       pch=16, cex=0.7,
       asp=1, main=names(df.ls)[i])
}
#' Note here how, because we're comparing two ranges with similar spread (both
`SMLsd`),
#' it doesn't really matter too much which values we use, the relationship always
looks
#' the same.
#'
#' And look at the relationship between **`SMLn_SMLsd` and `SMLn_LGsd`**
#+ fig.width=9, fig.height=3.5
par(mfrow=c(1,3))
for (i in 1:3) {
  plot(SMLn_SMLsd~SMLn_LGsd, data=df.ls[[i]],
       pch=16, cex=0.7,
       asp=1, main=names(df.ls)[i])
}
#' But now here, comparing a small spread range (`SMLsd`) to a wide-spread range
(`LGsd`),
#' we see that the raw data loses it's power to show the relationship because the
variance
#' within the wide-spread range is so little since each UD pixel is worth less. The
two
#' different scaling methods, however, both produce similar results
#'
#'
#' And look at the relationship between **`SMLn_SMLsd` and `LGn_LGsd`**
#+ fig.width=9, fig.height=3.5
par(mfrow=c(1,3))
for (i in 1:3) {
  plot(SMLn_SMLsd~LGn_LGsd, data=df.ls[[i]],
       pch=16, cex=0.7,
       asp=1, main=names(df.ls)[i])
}

```

```
}  
# ' Same as the previous one.  
# '  
# ' ## Abbreviated Conclusion  
# ' I think that using the inverted volume (`vol`) or the rescaled `@data$ud` (`rscl`)  
# ' methods are both appropriate and both basically the same, in the end. I like the  
# ' `rscl`  
# ' option because it's fewer steps, and I find the 0 to 1 range to be even more  
# ' intuitive  
# ' than a 0 to 100 range.
```

TAGGED IN

R Reporting RStudio Rmarkdown

- NEXT
- PREVIOUS