

# Debugging in R

Zoe Vernon

23 September, 2020

## Useful links

- General advice for debugging
  - [Efficient Debugging](#) by Goldspink
  - [Debugging for Beginners](#) by Brody
- R specific debugging
  - [Advanced R](#) by Wickham
  - [Debugging in Rstudio](#) by Gadrow
- The material for this document is based on the SCF tutorial linked below.
  - [Berkeley-SCF tutorial](#) by Chris Paciorek
  - [Debugging demo](#) by Chris

## Learning objectives

The goal for this section is to become familiar with the debugging tools available in R as well as provide additional information on online forums and common errors in R. Note that the debugging tools in R are difficult to illustrate in a Rmd document, so I recommend watching this [screencast](#) from the SCF tutorial. I will also do a live demo at the beginning of section if that you all would find that useful.

Also, note that the material in this PDF is a summary of SCF tutorial on debugging, so if you want to see more detail please visit the [tutorial GitHub](#). The tutorial also has some good tips for defensive programming that you may find useful for preventing and catching errors in your code.

## R's debugging tools

Below is a list of the debugging tools available in R. I took screenshots illustrating how some of the tools work in R Studio in the example below.

### Tools

- Use `traceback` to view the call stack, which can help pinpoint where an error is occurring.
- Use `recover` to navigate the stack of active function calls at the time of the error and browse within the desired call. If you set `options(error = recover)` then `recover` is invoked whenever an error occurs. You can revert the options to the default with `options(error = NULL)`.
- `browser()`: pauses current execution, provides an interactive interpreter. You can now step through a function line-by-line to find errors.
- `debug(someFunc)`: sets a `browser()` statement at the first line of `someFunc`
  - `undebug(someFunc)` removes the `debug()` statement. Or close the R session

- `debugonce(someFunc)` lets you debug only once, no need to run `undebug()`
- `trace()`: allows you to temporarily modify a function without saving the modifications
  - Edits will be removed when session ends
  - Alternatively, you can use `untrace()` to remove temporary edits.

## Example of debugging

We will use the `jackknife.R` code to understand the debugging tools.

```
library(MASS)

gamma_est <- function(data) {
  # this fits a gamma distribution to a collection of numbers
  m <- mean(data)
  v <- var(data)
  s <- v/m
  a <- m/s
  return(list(a=a,s=s))
}

calc_var <- function(estimates){
  var_of_ests <- apply(estimates, 2, var)
  return(((n-1)^2/n)*var_of_ests)
}

gamma_jackknife <- function(data) {
  ## jackknife the estimation

  n <- length(data)
  jack_estimates = gamma_est(data[-1])
  for (omitted_point in 2:n) {
    jack_estimates = rbind(jack_estimates, gamma_est(data[-omitted_point]))
  }

  jack_var = calc_var(jack_estimates)

  return(sqrt(jack_var))
}

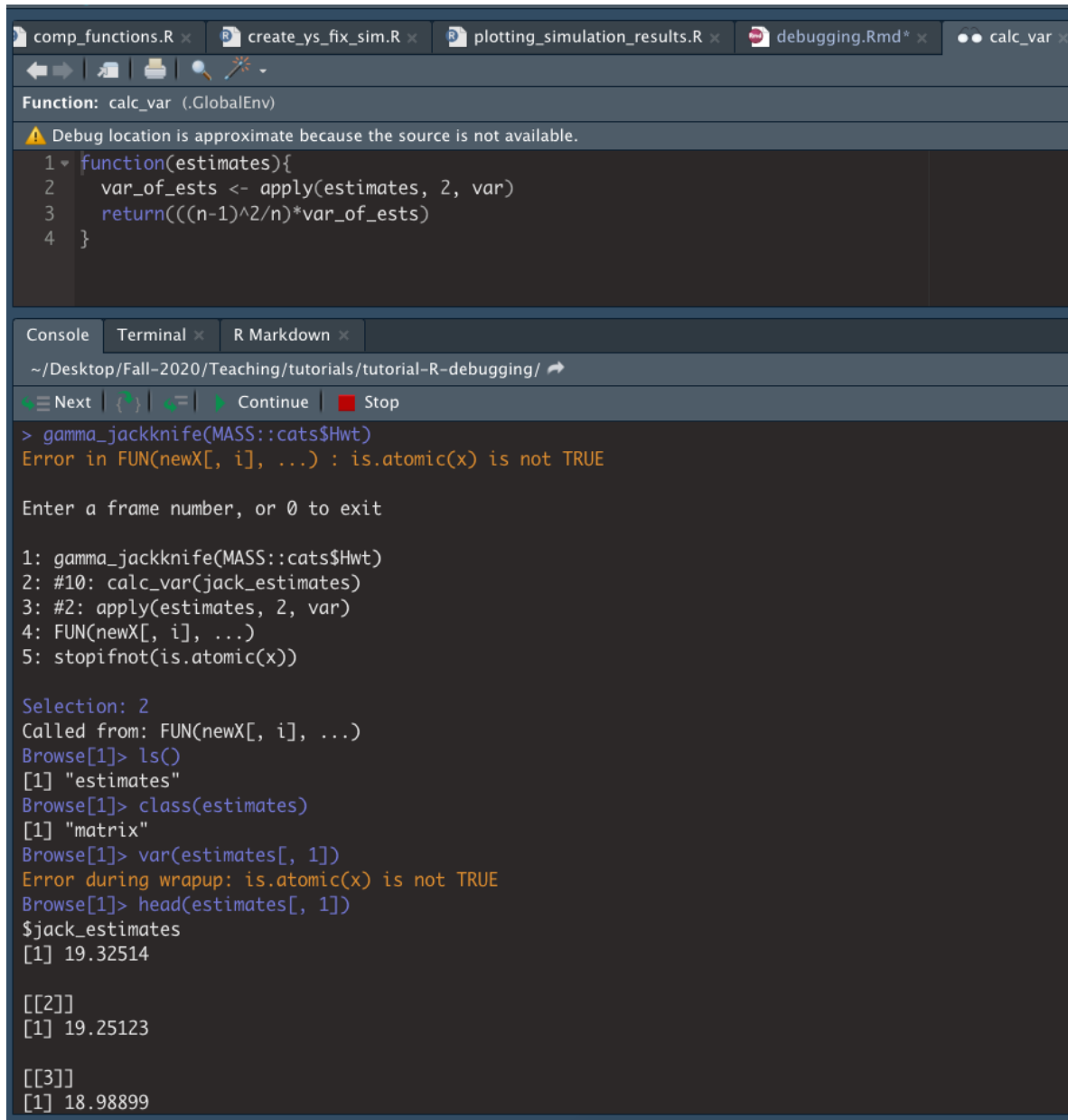
# jackknife gamma dist. estimates of cat heart weights
gamma_jackknife(MASS::cats$Hwt)
```

Notice that there is an error returned by the function, but it is unclear what is producing the error. We can start by calling `traceback()` to see what may have gone wrong.

```
> traceback()
6: stop(simpleError(msg, call = if (p <- sys.parent(1L)) sys.call(p)))
5: stopifnot(is.atomic(x))
4: FUN(newX[, i], ...)
3: apply(estimates, 2, var) at #2
2: calc_var(jack_estimates) at #10
1: gamma_jackknife(MASS::cats$Hwt)
```

`traceback()` shows us the set of calls leading up to the error. We see that the error is produced at 5, and thus came from the call at 4 `FUN(newX[, i], ...)` which occurred after calling `calc_var()` function and attempting to execute the `apply` statement.

An alternative to `traceback()` is `recover()`. If we have set `options(error = recover)` and call `gamma_jackknife(MASS::cats$Hwt)` again we will see the call stack (in reverse order of `traceback`), but now we have the option to select a number in the stack that we would like to enter. I selected 2 and entered the `calc_var` function. Typing `ls()` showed me that the only object in the function environment is `estimates`, which is a matrix. However, I see the `is.atomic(x)` error when I try to compute the variance of a column. When we look at the column, we can now see that we output a list, instead of a vector and we know exactly where the error is occurring. To exit we type `Q` and hit enter.



```
comp_functions.R x create_ys_fix_sim.R x plotting_simulation_results.R x debugging.Rmd* x calc_var x
Function: calc_var (.GlobalEnv)
⚠ Debug location is approximate because the source is not available.
1 function(estimates){
2   var_of_ests <- apply(estimates, 2, var)
3   return(((n-1)^2/n)*var_of_ests)
4 }

Console Terminal x R Markdown x
~/Desktop/Fall-2020/Teaching/tutorials/tutorial-R-debugging/
Next | | | Continue | Stop
> gamma_jackknife(MASS::cats$Hwt)
Error in FUN(newX[, i], ...) : is.atomic(x) is not TRUE

Enter a frame number, or 0 to exit

1: gamma_jackknife(MASS::cats$Hwt)
2: #10: calc_var(jack_estimates)
3: #2: apply(estimates, 2, var)
4: FUN(newX[, i], ...)
5: stopifnot(is.atomic(x))

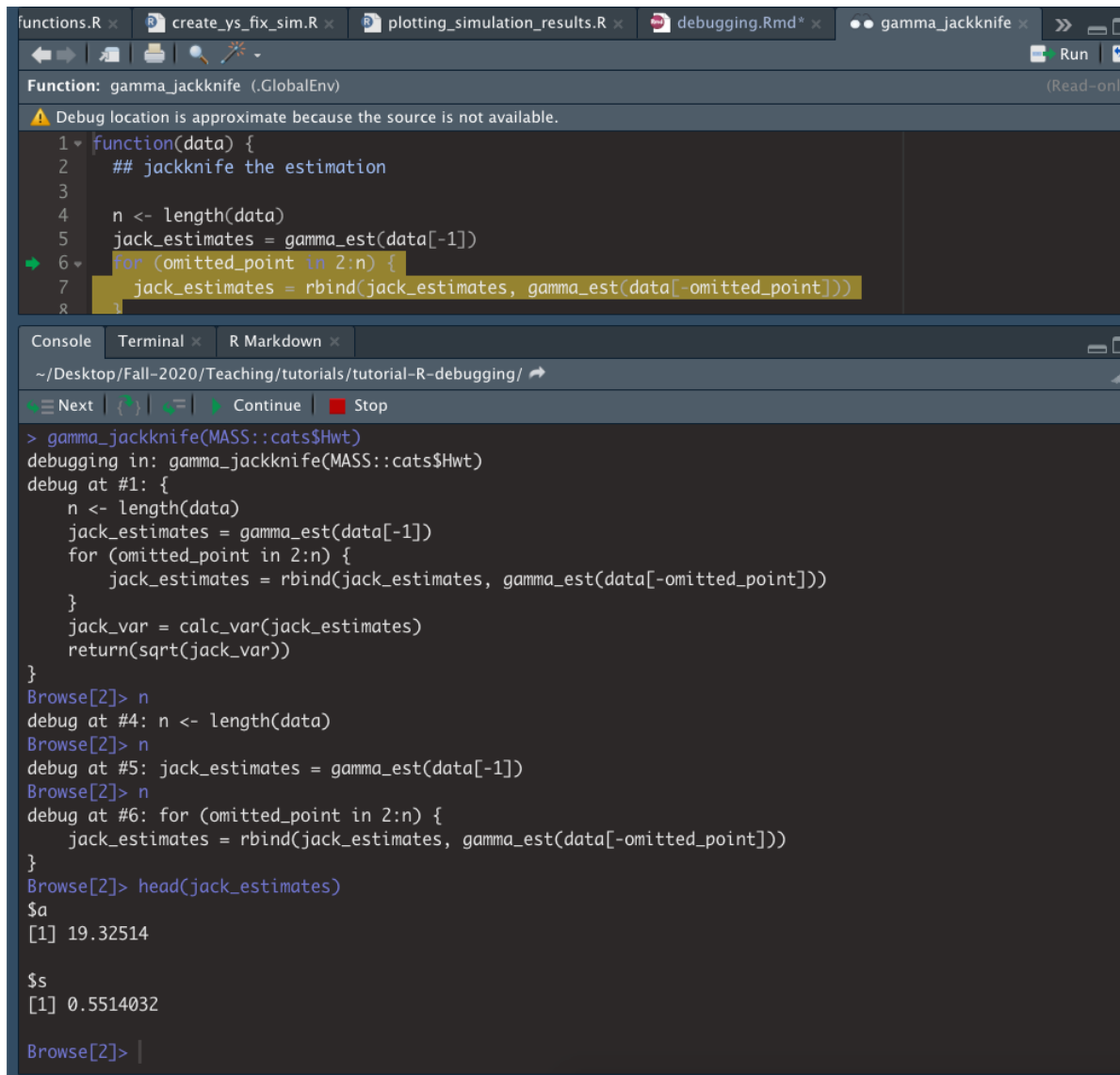
Selection: 2
Called from: FUN(newX[, i], ...)
Browse[1]> ls()
[1] "estimates"
Browse[1]> class(estimates)
[1] "matrix"
Browse[1]> var(estimates[, 1])
Error during wrapup: is.atomic(x) is not TRUE
Browse[1]> head(estimates[, 1])
$jack_estimates
[1] 19.32514

[[2]]
[1] 19.25123

[[3]]
[1] 18.98899
```

Now let's say we want to browse in the `gamma_jackknife()` function to figure out why we are passing a list to `calc_var` we can utilize the `debug()` function, which will allow us to step through `gamma_jackknife` one line at a time. We first call `debug(gamma_jackknife)` and then when we attempt to

run `gamma_jackknife(MASS::cats$Hwt)`, because an error is produced, we will enter the browser mode.



```
functions.R x create_ys_fix_sim.R x plotting_simulation_results.R x debugging.Rmd x gamma_jackknife x Run
Function: gamma_jackknife (.GlobalEnv) (Read-only)
⚠ Debug location is approximate because the source is not available.
1 function(data) {
2   ## jackknife the estimation
3
4   n <- length(data)
5   jack_estimates = gamma_est(data[-1])
6   for (omitted_point in 2:n) {
7     jack_estimates = rbind(jack_estimates, gamma_est(data[-omitted_point]))
8   }
9   jack_var = calc_var(jack_estimates)
10  return(sqrt(jack_var))
11 }

Console Terminal x R Markdown x
~/Desktop/Fall-2020/Teaching/tutorials/tutorial-R-debugging/
Next { } Continue Stop
> gamma_jackknife(MASS::cats$Hwt)
debugging in: gamma_jackknife(MASS::cats$Hwt)
debug at #1: {
  n <- length(data)
  jack_estimates = gamma_est(data[-1])
  for (omitted_point in 2:n) {
    jack_estimates = rbind(jack_estimates, gamma_est(data[-omitted_point]))
  }
  jack_var = calc_var(jack_estimates)
  return(sqrt(jack_var))
}
Browse[2]> n
debug at #4: n <- length(data)
Browse[2]> n
debug at #5: jack_estimates = gamma_est(data[-1])
Browse[2]> n
debug at #6: for (omitted_point in 2:n) {
  jack_estimates = rbind(jack_estimates, gamma_est(data[-omitted_point]))
}
Browse[2]> head(jack_estimates)
$a
[1] 19.32514

$s
[1] 0.5514032
Browse[2]> |
```

We can use the graphical interface in R Studio or the command line, with the command `n` to step through lines of the code and see what it outputs. Here we see that `gamma_est` is returning a list and that is likely the source of our issues.

For more details on these functions, as well as how to use `trace` to temporarily add edits see the SCF tutorial and the screencast. Also, as I stated above if there is enough interest I can do a live demo at the beginning of section.

## Common errors

- Parenthesis mis-matches
- `[[...]]` vs. `[...]`

```
# example list
myList <- list("A"=1:10,
```

```

"B"=11:20)

# one set
cat("Type: ", typeof(myList[1]), "\nLength: ", length(myList[1]), sep = "")

## Type: list
## Length: 1

# two sets
cat("Type: ", typeof(myList[[1]]), "\nLength: ", length(myList[[1]]), sep = "")

## Type: integer
## Length: 10

```

- == vs. =
- Comparing real numbers exactly using == is dangerous because numbers on a computer are only represented to limited numerical precision.

```

# exact comparison
1/3 == 4*(4/12 - 3/12)

## [1] FALSE

# approximate comparison
# default tolerance is sqrt(.Machine$double.eps)
all.equal(target = 1/3 ,current = 4*(4/12 - 3/12))

## [1] TRUE

```

- You expect a single value but execution of the code gives a vector
- You want to compare an entire vector but your code just compares the first value (e.g., in an if statement)
  - consider using identical() or all.equal()

```

x <- 1:10
y <- 1:5
if (x == y) {
  print("Equal")
}else {
  print("Not equal")
}

```

```

## Warning in if (x == y) {: the condition has length > 1 and only the first
## element will be used

```

```

## [1] "Equal"

if (identical(x, y)) {
  print("Equal")
}else {
  print("Not equal")
}

```

```

## [1] "Not equal"

```

- Silent type conversion when you don't want it, or lack of coercion where you're expecting it
  - eg., read.csv() and the stringsAsFactors argument

- Using the wrong function or variable name
- Giving unnamed arguments to a function in the wrong order
- In an if-else statement, the `else` cannot be on its own line (unless all the code is enclosed in `{}`) because R will see the `if` part of the statement, which is a valid R statement, will execute that, and then will encounter the `else` and return an error.
- Forgetting to define a variable in the environment of a function and having R, via lexical scoping, get that variable as a global variable from one of the enclosing environments. At best the types are not compatible and you get an error; at worst, you use a garbage value and the bug is hard to trace. In some cases your code may work fine when you develop the code (if the variable exists in the enclosing environment), but then may not work when you restart R if the variable no longer exists or is different.
  - Clear your environment before testing (`rm(list=ls());gc()`)
  - Restart R session and test
- R (usually helpfully) drops matrix and array dimensions that are extraneous. This can sometimes confuse later code that expects an object of a certain dimension.

```
# 3x3 matrix
myMat <- matrix(data = 1:9, nrow = 3, ncol = 3)

# lost dimensions
dim(myMat[1, ])

## NULL

# keep dimensions
dim(myMat[1, , drop = FALSE])

## [1] 1 3
```

## Getting help online

### Online forums / mailing lists

There are online forums that have lots of useful postings. In general if you have an error, others have already posted about it.

- Simple web searches - *a la Google*
  - You may want to include “in R” or preface your question with “R yada yada yada”
- [Stack overflow](#): R stuff will be tagged with ‘R’
  - <http://stackoverflow.com/questions/tagged/r>
- R help special interest groups (SIG) such as r-sig-hpc (high performance computing), r-sig-mac (R on Macs), etc.
  - To search a SIG you might include the name of the SIG in the search string
- [Rseek.org](#) for web searches restricted to sites that have information on R
- R help: [R mailing lists archive](#)

Note: of course these are also helpful for figuring out how to do things, not just for fixing bugs. For example, this [blog post](#) has a guide to R based simply on Stack Overflow posts.

## Asking questions online

If you've searched the archive and haven't found an answer to your problem, you can often get help by posting to the R-help mailing list or one of the other lists mentioned above. A few guidelines (generally relevant when posting to mailing lists beyond just the R lists):

- Search the archives and look through relevant R books or manuals first.
  - [Advanced R](#) by Hadley Wickham
- Boil your problem down to the essence of the problem, giving an example, including the output and error message
  - My first [SO](#) post
    - \* Notice the not-so-polite comments, see the remark below
  - My second [SO](#) question
- Say what version of R, what operating system and what operating system version you're using.
  - Provide `sessionInfo()` and `Sys.info()`. These show the current state of your machine
- Read the [R mailing list posting guide](#).

The R mailing lists are a way to get free advice from the experts, who include some of the world's most knowledgeable R experts - seriously - members of the R core development team contribute frequently. The cost is that you should do your homework and that sometimes the responses you get **may be blunt**, along the lines of "read the [manual](#)". Chris considers it a pretty good tradeoff - where else do you get the foremost experts in a domain actually helping you?

## Group problem for section: `logitBoot()`

- Load `data.csv`
- Fit a logistic regression model  $y \sim x$  called `mod` in the script provided.
- What is the std of the coefficient of `x` from `summary(mod)`?
- Now find the estimate of the same parameter now using bootstrap by simply calling `logitBoot()` as provided in the script.
- Why is this estimate so much larger?
- Use debugging tools to figure out the bug.

A couple hints to help get started - Use `debug(logitBoot)` to step through the function. Determine which entry/entries in `out` are particularly large or small.

- You may also want to use `trace()` on the `myGLM` function to temporarily edit the values it returns to figure out what is going wrong at the entry/entries that you found to be out of sample above.

The code can be found in the `logitBoot.R` script and the data is stored in the `data.csv` file.

```
my_data <- read.csv('./data.csv')

logitBoot <- function(y, x, nBoot = 2000) {
  set.seed(5)
  out <- sapply(seq_len(nBoot), myglm, y, x)
  boot_se <- sd(out)
  return(boot_se)
}

myGLM <- function(i, y, x) {
  n <- length(y)
  ind <- sample(seq_len(n), n, replace = TRUE)
  out <- glm(y[ind] ~ x[ind], family = 'binomial')
  return(out$coef[2])
}
```

```
mod <- glm(y ~ x, data = my_data, family = 'binomial')
summary(mod)
## note that the standard error for the regression coefficient is ~3

logitBoot(my_data$y, my_data$x)
```