Troubleshooting for R Functions

Fixing Mistakes, Preventing Them

Paul E. Johnson[1][2]

[1]University of Kansas, Department of Political Science [2]Center for Research Methods and Data Analysis

2015

Outline

Outline

"Debugger", Generally Speaking

- Most programming languages have debuggers.
  - "gdb" is the GNU debugger for C, C++, etc
- Key idea: Run a program "inside" the debugger
- "break point": halt a program at a place in the code to inspect the condition of the variables.
- "back trace": When a program crashes, recover the last few commands that were run.
- Many programs can create "core files" that can inspected after a program crash

R has Interactive Debugging

debug(func) An R function that asks the interpreter to stop whenever a function named "func" is called.

Caution: The function must be "in" the R session.

Example

```
debug(lm)
mod1 <- lm(y1 <- x1 + x2, data = mydata)
```

browser() Inserting the function "browser()" into your code creates a break point. R interpreter will act as if we had run debug().

```
myGiantFn <- function(x, y, z, a, b){
  qq <- x + y
  browser()
  }
```

To Learn More

- R Core Team, Writing R Extensions (run "help.start()")
- John Chambers, *Software for Data Analysis*
- Hadley Wickham, *Advanced R*, Chap 9

What to do inside the debugger

- Type into the R console

n <Ret>	next. Steps "into" the next level of code
<Ret>	Similar to next, but does not step "into" a for loop
c<Ret>	continue (starts R running from there)
where	recent commands (usu. after trouble)
Q	quit debugging

- Important feature: User can Run R commands

ls()	displays currently available object
print(x)	displays contents of x (same as typing "x")

- In ESS tracebug mode, use these keys in the source code file (M means the Meta (Alt) key. All are capitalized):

M-N	Next step
M-C	continue (starts R running from there)
M-U	Step up one frame (get out of for loop)
M-Q	continue (starts R running from there)

  - see `http://ess.r-project.org/Manual/ess.html#ESS-tracebug`

debug

- Try running

```
set.seed(234234)
y <- rnorm(100)
x <- rnorm(100)
debug(lm)
m1 <- lm( y ~ x )
```

- In the R terminal, computation will stop at the start of lm
- The whole function will print out, then we step into it

debug ...

```
> m1 <- lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset", "
        weights", "na.action",
        "offset"), names(mf), 0L)
    ## SNIP 100s of lines cut
    z
}
```

Interact with lm in the Debugger/Browser

- "n" steps into a lower level of detail (inside "if" or "for")
- Just hit Enter after first "n" to stay in same level

```
Browse[2]> n
debug at <tmp>#5: ret.x <- x
Browse[2]> n
debug at <tmp>#6: ret.y <- y
Browse[2]>
debug at <tmp>#7: cl <- match.call()
Browse[2]>
debug at <tmp>#8: mf <- match.call(expand.dots
    = FALSE)
Browse[2]>
debug at <tmp>#9: m <- match(c("formula", "data
    ", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
```

Interact with lm in the Debugger/Browser ...

```
Browse[2]>
debug at <tmp>#11: mf <- mf[c(1L, m)]
Browse[2]>
debug at <tmp>#12: mf$drop.unused.levels <-
    TRUE
%$
```

- Inspect objects inside the running function

```
Browse[2]> ls()
 [1] "cl"             "contrasts"      "data"
     "formula"        "method"
 [6] "mf"             "model"          "na.action"
     "offset"         "qr"
[11] "ret.x"          "ret.y"          "singular.ok"
     "subset"         "weights"
[16] "x"              "y"
Browse[2]> model
```

Interact with lm in the Debugger/Browser ...

```
[1] TRUE
Browse[2]> mf
lm(formula = y ~ x)
Browse[2]> formula
y ~ x
Browse[2]> contrasts
NULL
Browse[2]> method
[1] "qr"
```

How do you insert "browser()" into the middle of a function?

- Insert "browser()" in the middle of your function, re-load the function, and run the command that calls the function.
- If you want to insert browser into a function from a package, can do in a few ways.
- R allows "in memory" changes to a function. Consider `fix()`.

```
fix(lm)
```

- The program you are using to interact with R will decide how to open that function.
  - In an R console in Linux, the function opens in vi. Edit, then close to launch it back into the session.
  - Emacs 24.4 refuses to edit the file until I run

```
M-x server start
```

After that, a new buffer pops up with the code.

How do you insert "browser()" into the middle of a function? ...

- If you want to edit functions in this way, Emacs-ESS suggests instead
  - `C-c C-e C-d` to "dump" a function as a text file
  - `C-c C-l <Ret>` to load the edited function back.
- Rstudio pops open a new editor frame. The "save" button at the bottom loads the function into the R session.
- Changes we make in this way are purely temporary.

Outline

try and tryCatch

- Suppose a command inside your function causes a function to fail. Ordinarily, R processing will stop
- Most programming have some "try" "catch" logic so that commands may fail without destroying the progress of the larger function.
- Examples
  - Try to run something, ignore error entirely

```
try(file.symlink(normalizePath("../ppimages")
    ), tmpdir), silent = TRUE)
```

  - Try to run something, check for class of result

```
h11a <- try(solve(g11),silent=TRUE)

if (inherits(h11a,"try-error")) {
            h11a <- mpinv(g11)
}
```

try and tryCatch ...

- tryCatch can put those chores together. This runs lrm, if that fails it returns a vector of NA

```
out1 <- tryCatch( lrm( ...whatever ...),
    error = function(x) {rep(NA, 6)})
```

Outline

Advice 1: checkUsage

A Description of the Problem

- R searches from inside out for variables and functions
- If a function accidentally uses an object from the larger environment, you may have a hard-to-find bug.
- Best practice: isolate function from "accidental" access to environment variables.

An Example of the Problem

- Here's a function with an undefined variable "x"

```
myf <- function(z){
    log(z) + 2*x
}
```

- It will fail if no x exists

```
rm(x)
myf(1)
```

```
Error in myf(1) : object 'x' not found
```

- But if x happens to be floating about, it succeeds

```
x <- c(1,2,3)
myf(1)
```

```
[1] 2 4 6
```

- Dangers are obvious, yes? Other people will get errors if you give them myf().

Use codetools::checkUsage

- install codetools (by Luke Tierney)
- Run checkUsage

> *'checkUsage' checks a single R closure. Options control which possible problems to report. The default settings are moderately verbose. ...*
>
> *'checkUsageEnv' and 'checkUsagePackage' are convenience functions that apply 'checkUsage' to all closures in an environment or a package.*

```
library(codetools)
rm(x)
checkUsage(myf)
```

```
<anonymous>: no visible binding for global
  variable 'x' (functions-2.Rnw:435)
```

Outline

Optimization Intuitions and Facts

- New programmers sometimes fixate on simple ideas about what is "fast"
- Experienced programmers develop
  - intuitions about what might be "fast"
  - suspicion that their intuitions are probably wrong
- General Advice
  - Design code that is clear, well organized, separates chores understandably.
  - If it really is SLOW, run a profiler to find out why.
- Donald Knuth's comment widely cited "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Faster R

- I started a webpage "Faster R: Things to not forget" about ways to make R go faster `http://pj.freefaculty.org/blog/?p=122`
- Coming away from this, we have a few clear principles and a lot of examples.

Clear Principles

- Allocating memory is slow, so do it infrequently if possible
- Take advantage of optimizations that the R Core Team has built into the R code base
- Some functions that "seem" harmless are unnecessary and can slow down a function.

Vectorizing

- This is a way of pushing a lot of calculations out of the R interpreter over to the C functions
- Lets review the example `http://pj.freefaculty.org/R/WorkingExamples/efficiency-vectorize.html`

Matrix algebra

- If you need $X^T X$, don't write R like

```
t(X) %*% X
```

- DO use crossprod. It can be called with either

```
crossprod(X)
```

```
crossprod(X, X)
```

- Why faster? Note the result is symmetric. Hence it is only necessary to calculate one triangle
- similar function tcrossprod for Xy^T.

Similarly, rowSum

- Novices might write

```
rowtotal <- X[ ,1] + X[ ,2] + X[ ,3]
```

- Better to ruse

```
rowtotal <-  rowSum(X)
```

- Why? rowSum sends all of the work to the faster C side in one block, wherease the first one calls the "+" function over and over.

Smarter matrix algebra

- Wood, Generalized Additive Models, observes that this is very slow:

```
A %*% B %*% y
```

  and this is much faster:

```
A %*% (B %*% y)
```

  A and B are NxN, while y is Nx1.

- Each matrix calculation requires a certain number of floating point operations. (By) reduces to an $Nx1$ thing, thus eliminating any need to do an $N \times N$ calculation.

Examples: Removing columns from a data frame

- Example: Removing columns from a data frame.
  - This is slow if there are many columns to remove:

```
dat$A <- NULL
dat$B <- NULL
dat$C <- NULL
dat$D <- NULL
```

  - That's slow because data structure has to be repeatedly copied.
  - Much faster to find columns you want to keep, and keep them in one step

```
datnames <- colnames(dat)
keepnames <- setdiff(datnames, c("A", "B", "
    C", "D"))
dat <- dat[ , keepnames]
```

Examples: appending rows on bottom of a data frame

- Many R users are drawn to an idiom like this:

```
dat <- matrix(NA, nrow = 1, ncol = 6)
for(j in 1:100){
    ## some huge calculation creates a row
    x <- massive(y, z)
    dat <- rbind(dat, x)
}
```

- We find that the repeated use of rbind is poor
- One should either
  1. Create a big empty matrix and fill in the rows, or
  2. Create a list of result vectors, and rbind them together all at once.

Lets discuss the example here: `http://pj.freefaculty.org/R/WorkingExamples/efficiency-stackListItems-01.R`

Notice what they do in BLAS

- Try to grab memory in big chunks, not little bites.
- BLAS functions will Avoid creating new matrices
- They DO re-use matrices, writing results on top of input

Examples: appending rows on bottom of a data frame

- Matrices generally faster than data frames or lists
- An R object with "names" will be processed more slowly than the same thing with names
- Vectorization is Good. The [function is bad
- Warning against "for loops" is mostly driven by concern about use of "[". (That is, apply or lapply may not actually be faster).

Using the R Profiler

- Begin with a chunk of R code that seems slow
- Before that chunk, insert code like this

```
Rprof("Jones_Project-2015-1.out")
```

- Your R code chunk should run for a "while" so the profiler has a chance to study it. Give it a big enough sample of the slow function's use so it can diagnose the problem.
- After your chunk, add this

```
Rprof(NULL)
summaryRprof("Jones_Project-2015-1.out")
```

Chasing Amelia

- December 2012, we noticed Amelia (an MI) package became very slow
- Alex Schoemann made a test case and we profiled it: `http://pj.freefaculty.org/scraps/profile/prof-puzzle-1.Rout`

system.time() is not the greatest

but it is the easiest.
Wrap any code inside
system.time( )
and you'll get back a clock time estimate:

```
  > system.time(
+       impA <- amelia(datM, m = 5, idvars="group",
    p2s = 0)
+       )
   user  system elapsed
141.805   0.360 142.151
```