Debugging and Good Practices

September 15, 2013

Sources:

- Chambers
- Roger Peng's notes on debugging in R
- Murrell, Introduction to Data Technologies, Ch. 2
- Journal of Statistical Software vol. 42: 19 Ways of Looking at Statistical Software
- Wilson et at., Best practices for scientific computing, ArXiv:1210:0530

This unit covers debugging (and practices for avoiding bugs), good coding/software development practices, and doing reproducible research. As in later units of the course, the material is not in any fundamental way specific to R, but some details and the examples are in R.

1 Common syntax errors and bugs

1.1 Tips for avoiding bugs

1. Plan out your code in advance, including all special cases/possibilities.
2. When doing software development, write tests for your code early in the process.
3. Build up code in pieces, testing along the way. Make big changes in small steps, sequentially checking to see if the code has broken on test case(s).
4. Code in a modular fashion, making good use of functions, so that you don't need to debug the same code multiple times. Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion (like the UNIX utilities)

5. Use core R functionality and algorithms already coded. Figure out if a functionality already exists in (or can be adapted from) an R package (or potentially in a C/Fortran library/package): code that is part of standard mathematical/numerical packages will probably be more efficient and bug-free than anything you would write.

6. Remove objects you don't need, to avoid accidentally using values from an old object via the scoping rules.

7. Be careful that the conditions of *if* statements and the sequences of *for* loops are robust when they involve evaluating R code.

8. Don't hard code numbers - use variables (e.g., number of iterations, parameter values in simulations), even if you don't expect to change the value, as this makes the code more readable and reduces bugs when you use the same number multiple times:

   ```
   speedOfLight <- 3e+08
   nIts <- 1000
   ```

9. Check that inputs to and outputs from functions (either functions you call or functions you write) are valid and use *warning()* and *stop()* to give a warning or stop execution when something unexpected happens (see Section 4.5).

10. Use *try()* with functions that may fail (see Section 4.5) in cases where you don't want overall execution to fail because a single piece of the execution fails.

11. Write code for clarity and accuracy first; then worry about efficiency. Write an initial version of the code in the simplest way, without trying to be efficient (e.g., you might use *for* loops even if you're coding in R); then make a second version that employs efficiency tricks and check that both produce the same output.

1.2 Common syntax errors and bugs

Some of these are not specific to R, though some are.

1. Parenthesis mis-matches

2. *[[. . .]]* vs. *[. . .]*

3. == vs. =

4. Comparing real numbers exactly using '==' is dangerous (more in a later Unit). Suppose you generate *x* = *0.333333* in some fashion with some code and then check:

   ```
   x == 1/3 # FALSE is the result
   ```

5. Vectors vs. single values:

   (a) || vs. | and *&&* vs. *&*

   (b) You expect a single value but your code gives you a vector

   (c) 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()*

6. Silent type conversion when you don't want it, or lack of coercion where you're expecting it

7. Using the wrong function or variable name

8. Giving unnamed arguments to a function in the wrong order

9. In an if-then-else statement, the *else* cannot be on its own line (unless all the code is enclosed in { }) because R will see the if-then part of the statement, which is a valid R statement, will execute that, and then will encounter the *else* and return an error. We saw this in Unit 4.

10. Forgetting to define a variable in the environment of a function and having the function, 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.

11. R (usually helpfully) drops matrix and array dimensions that are extraneous; which can sometimes confuse later code that expects an object of a certain dimension. The '[' operator takes an additional optional argument that can avoid dropping dimensions.

```
mat <- matrix(1:4, 2, 2)[1, ]

dim(mat)

## [1] 2

print(mat)
```

```
## [1] 1 3

colSums(mat)

## Error: 'x' must be an array of at least two dimensions

mat <- matrix(1:4, 2, 2)[1, , drop = FALSE]
```

2 Debugging Strategies

Debugging is about figuring out what went wrong and where it went wrong.

In compiled languages, one of the difficulties is figuring out what is going on at any given place in the program. This is a lot easier in R by virtue of the fact that R is interpreted and we can step through code line by line at the command line. However, beyond this, there are a variety of helpful tools for debugging R code. In particular these tools can help you step through functions and work inside of functions from packages.

2.1 Basic strategies

Read and think about the error message. Sometimes it's inscrutable, but often it just needs a bit of deciphering. Looking up a given error message in the R mailing list archive or on Stack Overflow or simply doing a web search with the exact message in double quotes can be a good strategy.

Fix errors from the top down - fix the first error that is reported, because later errors are often caused by the initial error. It's common to have a string of many errors, which looks daunting, caused by a single initial error.

Is the bug reproducible - does it always happen in the same way at at the same point? It can help to restart R and see if the bug persists - this can sometimes help in figuring out if there is a scoping issue and we are using a global variable that we did not mean to.

Another basic strategy is to build up code in pieces (or tear it back in pieces to a simpler version). This allows you to isolate where the error is occurring.

The *codetools* library has some useful tools for checking code, including a function, *findGlobals()*, that let's you look for the use of global variables

```
options(width = 55)
library(codetools)
findGlobals(lm)[1:25]

##  [1] "<-"             "=="             "-"
##  [4] "!"              "!="             "["
##  [7] "[[<-"           "{"              "$<-"
## [10] "*"              "&&"             "as.name"
## [13] "as.vector"      "attr"           "c"
## [16] "class<-"        "eval"           "gettextf"
## [19] ".getXlevels"    "if"             "is.empty.model"
## [22] "is.matrix"      "is.null"        "is.numeric"
## [25] "length"

f <- function() {
    y <- 3
    print(x + y)
}
findGlobals(f)

## [1] "<-"    "{"     "+"     "print" "x"
```

If you've written your code modularly with lots of functions, you can test individual functions. Often the error will be in what gets passed into and out of each function.

You can have warnings printed as they occurred, rather than saved, using `options(warn = 1)`. This can help figure out where in a loop a warning is being generated. You can also have R convert warnings to error using `options(warn = 2)`.

At the beginning of time (the 1970s?), the standard debugging strategy was to insert print statements in one's code to see the value of a variable and thereby decipher what could be going wrong. We have better tools nowadays.

2.2 Interactive debugging via the browser

The core strategy for interactive debugging is to use *browser()*, which pauses the current execution, and provides an interpreter, allowing you to view the current state of R. You can invoke *browser()* in four ways

- by inserting a call to *browser()* in your code if you suspect where things are going wrong

- by invoking the browser after every step of a function using *debug()*
- by using `options(error = recover)` to invoke the browser when *error()* is called
- by temporarily modifying a function to allow browsing using *trace()*

Once in the browser, you can execute any R commands you want. In particular, using *ls()* to look at the objects residing in the current function environment, looking at the values of objects, and examining the classes of objects is often helpful.

2.3 Using *debug()* to step through code

To step through a function, use `debug(nameOfFunction)`. Then run your code. When the function is executed, R will pause execution just before the first line of the function. You are now using the browser and can examine the state of R and execute R statements.

In addition, you can use "n" or return to step to the next line, "c" to execute the entire current function or current loop, and "Q" to stop debugging. We'll see an example in the demo code.

To unflag the function so that calling it doesn't invoke debug, use `undebug(nameOfFunction)`. In addition to working with functions you write you can use debug with standard R functions and functions from packages. For example you could do `debug(glm)`.

2.4 Tracing errors in the call stack

traceback() and *recover()* allow you to see the call stack (the sequence of nested function calls – see Unit 4) at the time of an error. This helps pinpoint where in a series of function calls the error may be occurring.

If you've run the code and gotten an error, you can invoke *traceback()* after things have gone awry. R will show you the call stack, which can help pinpoint where an error is occurring.

More helpful is to be able to browse within the call stack. To do this invoke `options(error = recover)` (potentially in your *.Rprofile* if you do a lot of programming). Then when an error occurs, *recover()* gets called, usually from the function in which the error occurred. The call to *recover()* allows you to navigate the stack of active function calls at the time of the error and browse within the desired call. You just enter the number of the call you'd like to enter (or 0 to exit). You can then look around in the frame of a given function, entering an empty line when you want to return to the list of calls again.

You can also combine this with `options(warn = 2)`, which turns warnings into errors to get to the point where a warning was issued.

2.5 Using *trace()* to temporarily insert code

trace() lets you temporarily insert code into a function (including standard R functions and functions in packages!) that can then be easily removed. You can use trace in a few ways - here's how you would do it most simply, where by default the second argument is invoked at the start of the function given as the first argument, but it is also possible to invoke just before exiting a function:

```
  trace(lm, recover) # invoke recover() when the function starts
  trace(lm, exit = browser) # invoke browser() when the function ends
  trace(lm, browser, exit = browser) # invoke browser() at start and
end
```

Then in this example, once the browser activates I can poke around within the *lm()* function and see what is going on.

The most flexible way to use *trace()* is to use the argument *edit = TRUE* and then insert whatever code you want whereeever you want. If I want to ensure I use a particular editor, such as *emacs*, I can use the argument *edit = "emacs"* . A standard approach would be to add a line with *browser()* to step through the code or *recover()* (to see the call stack and just look at the current state of objects). Alternatively, you can manually change the code in a function without using *trace()*, but it's very easy to forget to change things back and hard to do this with functions in packages, so *trace()* is a nice way to do things.

You call *untrace()*, e.g., `untrace(lm)`, to remove the temporarily inserted code; otherwise it's removed when the session ends.

Alternatively you can do `trace(warning, recover)` which will insert a call to *recover()* whenever *warning()* is called.

3 Getting help online

3.1 Searching for help

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

- R help: R mailing lists archive
- 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. Unfortunately these are not easily searchable, but can often be found by simple web searchs, potentially including the name of the SIG in the search.

- Simple web searches: You may want to include "in R", with the quotes in the search. 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

If you are searching you often want to search for a specific error message. Remember to use double quotes around your error message so it is not broken into individual words by the search engine.

Just searching the R mailing list archive often gives you a hint of how to fix things. An example occurred when I was trying to figure out how fix a problem that was reporting a "*invalid multibyte string*" error in some emails in a dataset of Spam emails. I knew it had something to do with the character encoding and R not interpreting the codes for non-ASCII characters correctly but I wasn't sure how to fix it. So I searched for *"invalid multibyte string*". Around the 8th hit or so there was a comment about using *iconv()* to convert to the UTF-8 encoding, which solved the problem.

Note: of course the various mailing lists 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.

3.2 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):

1. Search the archives and look through relevant R books or manuals first.
2. Boil your problem down to the essence of the problem, giving an example, including the output and error message
3. Say what version of R, what operating system and what operating system version you're using. Both *sessionInfo()* and *Sys.info()* can be helpful for getting this information.
4. Read the posting guide.

The mailing list is 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". I think it's a pretty good tradeoff - where else do you get the foremost experts in a domain actually helping you?

4 Good coding practices

Some of these tips apply more to software development and some more to analyses done for specific projects; hopefully it will be clear in most cases.

4.1 Editors

Use an editor that supports the language you are using (e.g., *Emacs/Aquamacs*, *vim, TextMate, WinEdt*, *Tinn-R*, or the built-in editors in *RStudio* or the Mac R GUI). Some advantages of this can include: (1) helpful color coding of different types of syntax and of strings, (2) automatic indentation and spacing, (3) code can often be run or compiled from within the editor, (4) parenthesis matching, (5) line numbering (good for finding bugs).

4.2 Coding syntax

- Header information: put metainfo on the code into the first few lines of the file as comments. Include who, when, what, how the code fits within a larger program (if appropriate), possibly the versions of R and key packages that you wrote this for

- Indentation: do this systematically (your editor can help here). This helps you and others to read and understand the code and can help in detecting errors in your code because it can expose lack of symmetry.

- Whitespace: use a lot of it. Some places where it is good to have it are (1) around operators (assignment and arithmetic), (2) between function arguments and list elements, (3) between matrix/array indices, in particular for missing indices.

- Use blank lines to separate blocks of code and comments to say what the block does

- Split long lines at meaningful places.

- Use parentheses for clarity even if not needed for order of operations. For example, `a/y*x` will work but is not easy to read and you can easily induce a bug if you forget the order of ops.

- Documentation - add lots of comments (but don't belabor the obvious). Remember that in a few months, you may not follow your own code any better than a stranger. Some key things to document: (1) summarizing a block of code, (2) explaining a very complicated piece of code - recall our complicated regular expressions, (3) explaining arbitrary constant values.

- For software development, break code into separate files (<2000-3000 lines per file) with meaningful file names and related functions grouped within a file.
- Choose a consistent naming style for objects and functions: e.g. *nIts* vs. *n.its* vs *numberOfIts* vs. *n_its*
  - This R journal articlesummarizes the state of naming styles on CRAN.
  - Adler and Google's R style guide recommend naming objects with lowercase words, separated by periods, while naming functions by capitalizing the name of each word that is joined together, with no periods.
  - On the other hand, programmers who use other languages dislike R code with periods in it except in the context of object-oriented programming. E.g., *summary.lm* is clear in that the period distinguishes the method from the class. Naming a method something like *special.summary.lm* or an object *my.summary* then confuses things.
- Try to have the names be informative without being overly long.
- Don't overwrite names of objects/functions that already exist in R. E.g., don't use 'lm'.

  ```
  > exists("lm")
  ```
- Use active names for functions (e.g., *calc.loglik*, *calcLogLik*)
- Learn from others' code

This semester, someone will be reading your code - Tessa and me when we look at your assignments. So to help us in understanding your code and develop good habits, put these ideas into practice in your assignments.

4.3 Coding style

This is particularly focused on software development, but some of the ideas are useful for data analysis as well.

- Break down tasks into core units
- Write reusable code for core functionality and keep a single copy of the code (w/ backups of course) so you only need to change it once
- Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion (like the UNIX utilities)

- Write functions that take data as an argument and not lines of code that operate on specific data objects. Why? Functions allow us to reuse blocks of code easily for later use and for recreating an analysis (reproducible research). It's more transparent than sourcing a file of code because the inputs and outputs are specified formally, so you don't have to read through the code to figure out what it does.
- Functions should:
  - be modular (having a single task);
  - have meaningful name; and
  - have a comment describing their purpose, inputs and outputs (see the help file for an R function for how this is done in that context).
- Object orientation is a nice way to go
- Don't hard code numbers - use variables (e.g., number of iterations, parameter values in simulations), even if you don't expect to change the value, as this makes the code more readable:

  ```
  > speedOfLight <- 3e8
  ```
- Use R lists to keep disparate parts of related data together
- Practice defensive programming
  - check function inputs and warn users if the code will do something they might not expect or makes particular choices;
  - check inputs to *if* and the ranges in *for* loops;
  - provide reasonable default arguments;
  - document the range of valid inputs;
  - check that the output produced is valid; and
  - stop execution based on checks and give an informative error message.
- Try to avoid system-dependent code that only runs on a specific version of an OS or specific OS
- Learn from others' code
- Consider rewriting your code once you know all the settings and conditions; often analyses and projects meander as we do our work and the initial plan for the code no longer makes sense and the code is no longer designed specifically for the job being done.

4.4 Version control

Some basic ideas of version control are as follows:

- Whenever you make changes to the structure of your code/program, make it a new version, numbering sequentially (or with more sophisticated version numbering used by software developers).

- Keep a text file that documents what changes from version to version (or do this in the document header). I find this easier than trying to name my code files informatively.

- DO NOT get rid of old code versions (text is cheap to store) and DO NOT get rid of blocks of code that work.

- Keep a running file (or comments in the code) of what you'd like to change in the future.

Even better than doing this manually, both for data analysis and your own project work, you can use a formal version control system (VCS), in particular git. *cvs* and *svn* are other VCS. You can host a *git* repository at *github.com* for free (provided you don't mind having others see it).

4.5 Dealing with run-time errors

When writing functions, and software more generally, you'll want to warn the user or stop execution when there is an error and exit gracefully, giving the user some idea of what happened. The *warning()* and *stop()* functions allow you to do this; in general they would be called based on an if statement. To stop code if a condition is not satisfied, you can use *stopifnot()*, e.g.,

```
> x <- 3
> stopifnot(is(x, "matrix"))
```

These approaches allow you to catch errors that can be anticipated.

Here's an example of building a robust square root function using *stop()* and *warning()*. Note you could use stopifnot(is.numeric(x)) in place of one of the checks here.

```
mysqrt <- function(x) {
    if (is.list(x)) {
        warning("x is a list; converting to a vector")
        x <- unlist(x)
    }
    if (!is.numeric(x)) {
        stop("What is the square root of 'bob'?")
    } else {
        if (any(x < 0)) {
            warning("mysqrt: found negative values; proceeding anyway")
            x[x >= 0] <- (x[x >= 0])^(1/2)
            x[x < 0] <- NaN
            return(x)
        } else return(x^(1/2))
    }
}
mysqrt(c(1, 2, 3))

## [1] 1.000 1.414 1.732

mysqrt(c(5, -7))

## Warning:  mysqrt:  found negative values; proceeding anyway

## [1] 2.236   NaN

mysqrt(c("asdf", "sdf"))

## Error:  What is the square root of 'bob'?

mysqrt(list(5, 3, "ab"))

## Warning:  x is a list; converting to a vector
## Error:  What is the square root of 'bob'?
```

You can control what happens when a warning occurs with *options()$warning*. This can be helpful for debugging - e.g., you can force R to stop if a warning is issued rather than continuing so you can delve into what happened.

Also, sometimes a function you call will fail, but you want to continue execution. For example, suppose you are fitting a bunch of linear models and occasionally the design matrix is singular. You can wrap a function call within the *try()* function (or *tryCatch()*) and then your code won't stop. You can also evaluate whether a given function call executed properly or not. Here's an example of fitting a model for extreme values:

```
library(ismev)

## Loading required package:  mgcv
## This is mgcv 1.7-22.  For overview type 'help("mgcv-package")'.

library(methods)
n <- 100
nDays <- 365
x <- matrix(rnorm(nDays * n), nr = nDays)
x <- apply(x, 2, max)
x <- cbind(rep(0, 100), x)
params <- matrix(NA, nr = ncol(x), nc = 3)
for (i in 1:ncol(x)) {
    fit <- try(gev.fit(x[, i], show = FALSE))
    if (!is(fit, "try-error"))
        params[i, ] = fit$mle
}
params

##        [,1]   [,2]     [,3]
## [1,]     NA     NA       NA
## [2,] 2.765 0.3202 -0.03077
```

Challenge: figure out how to use *tryCatch()* to deal with the error above. Note that I haven't used it and it seemed somewhat inscrutable on quick look.

5 Tips for running analyses

Save your output at intermediate steps (including the random seed state) so you can restart if an error occurs or a computer fails. Using *save()* and *save.image()* to write to *.RData* files work well for this.

Run your code on a small subset of the problem before setting off a job that runs for hours or days. Make sure that the code works on the small subset and saves what you need properly at the end.

6 Reproducible research

The idea of "reproducible research" has gained a lot of attention in recent years because of the increasing complexity of research projects, lack of details in the published literature, failures in being able to replicate or reproduce others' work, fraudulent research, and for other reasons.

We've seen a number of tools that can help with doing reproducible research, including version control systems such as git, the use of scripting such as bash and R scripts, and literate programming tools such as knitr, Sweave and R Markdown.

Provenance is becoming increasingly important in science. It basically means being able to trace the steps of an analysis back to its origins. *Replicability* is a related concept - the idea is that you or someone else could replicate the analysis that you've done. This can be surprisingly hard as time passes even if you're the one attempting the replication.

Open question: What is required for something to be replicable? What are the challenges in doing so?

6.1 Some basic strategies

- Have a directory for each project with meaningful subdirectories: e.g., *code*, *data*, *paper*
- Keep a document describing your running analysis with dates in a text file (i.e., a lab book)
- Note where data were obtained (and when, which can be helpful when publishing) and pre-processing steps in the lab book. Have data version numbers with a file describing the changes and dates (or in lab book).
- Have a file of code for pre-processing, one or more for analysis, and one for figure/table preparation.
  - Have the code file for the figures produce the EXACT manuscript figures, operating on an RData file that contains all the objects necessary to run the figure-producing code; the code producing the RData file should be in your analysis code file (or somewhere else sensible).
  - Alternatively, use *Sweave* or *knitr* for your document preparation.
- Note what code files do what in the lab book.

6.2 More formal tools

1. In some cases you may be able to carry out your complete workflow in a knitr/Sweave/R Markdown document.

2. Or in Python, you may be able to use the iPython Notebook.

3. You might consider using the UNIX utility *make*, which is generally used for compiling code, as a tool for reproducible research: see http://kbroman.github.io/minimal_make/ for more details.