Code Correctness & Efficiency

Robert B. Gramacy
Virginia Tech Department of Statistics

bobby.gramacy.com

Most built-in subroutines in R make judicious checks for the validity of input, and correctness of output.

- The better add-on packages do too.
- This is an under-appreciated feature of a well-designed programming language,
- and an easily overlooked aspect of software engineering, even for the simplest of tasks.

Compared to other languages, R is pretty good in this respect.

- It can be cryptic with errors, but it is not overly pedantic (think object precedence).

That said, its built-in debugger and profiler features pale in comparison to other languages (e.g., MATLAB).

Still R has many features/functions to help you

- prevent misuse of your own code (by others)
- diagnose errors in your own code, or in your use of code written by others
- automate recovery when errors are encountered,

and otherwise facilitate the practice of good (conservative) software engineering.

Sanity Checks

Peppering your code with checks,

- especially the validity of arguments passed to functions,

is *so* important. It should be habit, even when prototyping.

- It will pay dividends in time saved later.
- Otherwise its garbage-in garbage-out and none the wiser.

Recall the preamble of our `bisection()` root-finding from lecture 3.

```
bisection <- function(f, xl, xr,
   tol=sqrt(.Machine$double.eps), verb=0)
  {
    ## check inputs
    if(xl > xr) stop("must have xl < xr")

    ## setup and check outputs
    fl <- f(xl)
    fr <- f(xr)
    if(fl * fr > 0) stop("f(xl) * f(xr) > 0")
```

- What if we didn't have those checks?

The function `bisection2()` in the `.R` file is the same as our old `bisection()`, but without the checks.

```
> f <- function(x) log(x) - exp(-x)
> ## results in an error
> fr <- bisection(f, 2, 1)
Error in bisection(f, 2, 1) : must have xl < xr
> ## results in nonsense output
> fr2 <- bisection2(f, 2, 1)
> fr2$ans
[1] 1.5
> ## correct result
> fr3 <- bisection(f, 1, 2)
> fr3$ans
[1] 1.3098
```

Errors

`stop(...)` signals an error with a message to the screen.

It works like `cat(...)`

- accepting character strings that are pasted together to form the error message
- except that it terminates execution, returning control to the calling environment
- and ultimately back to the user at the command prompt (unless *caught*).

`stop()` is one of several exceptions in R.

Here is another example that deploys sanity checks/`stop()`.

```
> doWork <- function(filename) {
+
+   ## checking the type of file
+   if(class(filename) != "character" ||
+      length(filename) != 1)
+     stop("Filenames should be a single character string")
+
+   ## checking that the file exists
+   if(! file.exists(filename))
+     stop("Could not open file: ", filename)
+
+   ## otherwise do something with the file, e.g.,
+   read.delim(filename)
+ }
```

```
> doWork("this")
Error in doWork("this") : Could not open file: this
> doWork(1)
Error in doWork(1) : Filenames should be a single
                     character string
```

Of course, `read.delim()` has its own error checking:

```
> read.delim("this")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") : cannot open file 'this':
                      No such file or directory
> read.delim(1)
Error in read.table(file = file, header = header, ... :
  'file' must be a character string or connection
```

It is not uncommon for high-level/interface functions to have

- many lines of sanity checks,
- possibly combined with light code that *massages* input arguments into a convenient form.

The number of checking lines may be dwarfed by the number of lines of genuine computation.

- Again, garbage-in garbage-out, and authors don't want to be blamed for garbage-out.
- Often the real computation is buried in a subroutine.

E.g., see `blasso()` or `regress()` from the `monomvn` package.

Check/conform arguments to be of the right type, and check their dimensions against those of other arguments.

```
X <- as.matrix(X)
y <- as.numeric(y)
if(length(y) != nrow(X))
    stop("must have nrow(X) == length(y)")
```

Check length, sign, and class (with flexibility).

```
if(length(T) != 1 || T <= 1)
    stop("T must be a scalar integer > 1")
if(length(RJ) != 1 || !is.logical(RJ))
    stop("RJ must be a scalar logical")
if(length(lambda2) != 1 || lambda2 < 0)
    stop("lambda2 must be a non-negative scalar")
```

Use `match.arg()` with switch-style function arguments.
E.g., `regress()` has an argument called `method`, with "default":

```
method = c("lsr", "plsr", "pcr", "lasso", "lar",
    "forward.stagewise", "stepwise", "ridge",
    "factor")
```

This serves two purposes.

1. Shows the user the options; the first one is the real default.
2. Allows `match.arg()` to check the argument against a list for (partial) matches.

   ```
   method <- match.arg(method)
   ```

Finally, another option for sanity checking is `stopifnot(...)`.

- These work like `assert` statements in C.
- They evaluate each statement in `...`, and throw an error if *any* evaluate to `FALSE`

```
> x <- -2
> stopifnot(x > 0)
Error: x > 0 is not TRUE
```

They are nice as sanity checks in code development, but they are unhelpful in production code

- because they offer no error message capability.

Warnings

Sometimes you want to alert the user to something that might be troublesome,

- but that does not necessarily bar normal execution or the provision of sensible output.

In that case a `warning(...)` is appropriate; it works just like `stop(...)` with two exceptions.

1. They do not stop execution.
2. They are not immediately printed unless the `immediate.=TRUE` argument is used. Instead, they join a queue (max 50) for printing when control is passed back to the user at the console.

We've seen several examples already.

- `if` statements with multiple conditions.

```
> if(c(TRUE, FALSE)) TRUE else FALSE
[1] TRUE
Warning message:
In if (c(TRUE, FALSE)) TRUE else FALSE :
  the condition has length > 1 and only t
  the first element will be used
```

- When the names of the columns of a data frame being `attach()`ed clash with symbols in the current environment.

```
> x <- 1; attach(data.frame(x=2))
The following object(s) are masked by '.GlobalEnv':
    x
> x
[1] 1
```

  - not technically a `warning()`

`regress()` gives a warning when an OLS (`method="lsr"`) regression fails,

- e.g., when the matrix `X` is not of full rank.
- In that case, it uses one of the penalized methods instead,
- so the output it gives still summarizes a reasonable inference.

Although the output "under warning" is usually sensible, one must be wary that it is a symptom of something more troubling

- a bug in input/data preparation
- or a misunderstanding about appropriate usage.

When developing my own code

- peppered with warnings of my own creation,
- or using add-on packages (peppered with warnings),

I like to elevate warnings so that they are either

- all printed immediately `options(warn=1)`, or
- converted to errors `options(warn=2)`.

The latter is particularly useful for postmortem debugging,

- (in tandem with `options(error=recover)`),

to be discussed in more detail shortly.

If you're really sure that a warning (in an add-on package) is innocuous, and

- you're tired of seeing it
- or don't want your users to see it and become frustrated with your code even though its not your warning

they can be suppressed with `suppressWarnings(expr)`.

E.g., the `regress.pls()` subroutine to `regress()` in the `monomvn` package suppresses warnings coming from `pcr()` and `plsr()` from the `pls` package.

Catching errors

Sometimes it makes sense to automate and recover from an error, deep in a subroutine, without bothering the user.

E.g., maybe you've written code calling a function from an external library, say, which sometimes trips an error

- a circumstance for which you have a backup plan.

`try({expr, ...})` allows you to evaluate any expression and

- return the usual output if successful (no error)
- or catch any errors before they get to the user, allowing an alternate execution path
- which may involve a custom error message or warning.

Here is how `try()` works.

```
> res <- try({ 1/2 }, silent=TRUE)
> res
[1] 0.5
> res <- try({ 1/"2" }, silent=TRUE)
> res
[1] "Error in 1/\"2\" : non-numeric argument
     to binary operator\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in 1/"2": non-numeric argument
     to binary operator>
```

`try()` thus gives us the opportunity of a automating a second chance.

```
> res <- try({ 1/"2" }, silent=TRUE)
> if(class(res) == "try-error") {
+   res <- 1/as.numeric("2")
+ }
> res
[1] 0.5
```

Just as it is good programming practice to add sanity checks to your code, signaling informative errors to users,

- it is equally helpful to protect your users from errors tripped by subroutines in code you did *not* write.

As one example from my own work, see the function `glmn.hr()` in the accompanying `.R` file.

- It automates a penalized logistic regression via `glmnet()`,
- using `try()` to guard against errors that can arise due the random nature of the choice of λ via CV.

Debugging

Debugging is an essential part of programming, in any language.

- It often represents 90% or more of the effort.
- I'm skeptical when a first pass at coding leads to (apparently) working code.

Write code expecting bugs.

- And take advantage of debugging tools;
- climbing their learning curve is a good investment.

Most of the principles in debugging apply generically.

- Start small.
- Code in a top-down manner (routines comprised of well-defined, small, sub-routines coded as functions)
- but test sub-routines exhaustively (bottom-up too).
- Debug exhaustively: once anything looks wrong, check everything systematically.

Sanity checks are an integral part of the debugging process.

- Your task is vastly simplified when one is tripped.
- If no sanity checks are tripped, and the output is still wrong, then you don't have enough checks.

But sometimes it is not obvious, at first, what to check for.

- That takes some experience, and more infrastructure.
- And we'll come back to that later.

For now, suppose your code is generating an error or warning.

- Start by elevating all warnings to errors with `options(warn=2)`.

Post-mortem debugging

Your best tools for diagnosing/fixing errors in code are

- `traceback()` after code breaks, immediately after an error, and
- `options(error=recover)`, to enter the debugger following an error.

Sometimes `traceback()` is enough, and its the only option if you have not, previously, engaged the debugger

- via `debug()` or `options(error=recover)`.

These features are best explained via demonstration.

The file `mind.R` contains a simple pair of functions which calculate the minimum value of a (symmetric) matrix `d[i,j]`, `i != j`, and returns the row `i` and col `j` of that minimum.

- but the implementation has at least one bug.

```
> source("mind.R")
> m <- rbind(c(0,12,5), c(12,0,8), c(5,8,0))
> mind(m)
Error in (i + 1):(lx - 1) : argument of length 0
```

- Not particularly helpful.

If we were intimately familiar with the code we might recognize the offending statement `(i + 1):(lx - 1)`.

- If not, it might be hard to know where to look.

```
> traceback()
4: which.min(x[(i + 1):(lx - 1)]) at mind.R#22
3: FUN(newX[, i], ...) at mind.R#7
2: apply(dd[-n, ], 1, imin) at mind.R#7
1: mind(m)
```

- So the error was tripped in `mind.R` on line 22.
- But that doesn't necessarily mean that's where the bug is.

A rudimentary diagnostic adds `print()`s the code to inspect the objects before the offending expression.

```
> imin <- function(x) {
+   lx <- nrow(x); i <- x[lx]
+   print(lx); print(i)
+   j <- which.min(x[(i+1):(lx-1)])
+   return(c(j,x[j]))
+ }
> mind(m)
NULL
numeric(0)
Error in (i + 1):(lx - 1) : argument of length 0
```

Now, we could go in and add print statements on `x`, etc.

- We'll get there eventually, but this is not the most direct route.
- We're better off inside the debugger.

Tell R to automatically enter the debugging `browser()` upon an error.

```
> options(error=recover)
```

This augments the information from `traceback()` with an interactive environment.

- We can interact, via the command prompt, anywhere in the call stack maintained at the time of the error.

```
> mind(m)
Error in (i + 1):(lx - 1) : argument of length 0

Enter a frame number, or 0 to exit

1: mind(m)
2: mind.R#7: apply(dd[-n, ], 1, imin)
3: mind.R#7: FUN(newX[, i], ...)
4: mind.R#22: which.min(x[(i + 1):(lx - 1)])

Selection:
```

- To print `x`, `i`, or `lx`, we want to be inside `imin`,
- which is the `FUN=` argument to `apply()`.

- ... so choose 2:

```
Selection: 2
Called from: apply(dd[-n, ], 1, imin)
Browse[1]> x
[1] 1
Browse[1]> class(x)
[1] "numeric"
```

- The problem is that `x` is not a matrix, so `lx <- nrow(x)` doesn't make any sense.
- Perhaps we meant `lx <- length(x)`?

Press `Enter` then `0` to leave the `browser()`.

```
Browse[1]>

Enter a frame number, or 0 to exit

1: mind(m)
2: mind.R#7: apply(dd[-n, ], 1, imin)
3: mind.R#7: FUN(newX[, i], ...)
4: mind.R#22: which.min(x[(i + 1):(lx - 1)])

Selection: 0
>
```

Then fix `imin()`.

```
> imin <- function(x) {
+   lx <- length(x)
+   i <- x[lx]
+   j <- which.min(x[(i+1):(lx-1)])
+   return(c(j,x[j]))
+ }
```

and re-run:

- Darn!

```
> mind(m)
Error in mind(m) : subscript out of bounds
```

We see that the error was tripped in `mind()` now, not `imin`

- although the problem could still be in `imin()`.
- It could be returning a bad index.

```
Enter a frame number, or 0 to exit

1: mind(m)

Selection: 1
```

- An easy choice. :)

```
Browse[1]> where
where 1 at mind.R#14: eval(expr, envir, enclos)
where 2 at mind.R#14: eval(substitute(...
    envir = sys.frame(which))
where 3 at mind.R#14: function ()
..
```

Ok, so the problem occurred on line 14. That code is:

```
return(c(d[i,j],i,j))
```

- So we should inspect `i`, `j`, and `d`.

```
Browse[1]> i
[1] 2
Browse[1]> j
[1] 12
Browse[1]> d
     [,1] [,2] [,3]
[1,]    0   12    5
[2,]   12    0    8
[3,]    5    8    0
```

So we're trying to do `d[2,12]` but `d` is `3 x 3`.

`j` is obtained from `wmins` as

```
i <- which.min(wmins[1,])
j <- wmins[2,i]
```

So the problem must be in `wmins`,

```
Browse[1]> wmins
     [,1] [,2]
[1,]    2    1
[2,]   12   12
```

whose k^{th} row (from the output of `imin()` via `apply()`) is supposed to contain information about the minimum value in row `k` of `d`.

`wmins` says that the first row (`k=1`) of `d` is

```
Browse[1]> d[1,]
[1]  0 12  5
```

has a minimum of `12` at index `2`

- but it should be `5` at `3`.

So something is wrong with this line:

```
wmins <- apply(dd[-n,],1,imin)
```

There are several possibilities here.

But since ultimately `imin()` is called, we can check them all from within that function.

- Trouble is, the error was not tripped inside that function,
- so we can't `browse()` it in the current debugging session.

We need to

- quit out and tell the debugger to stop in `imin()` and let us poke around.

```
Browse[1]> Q     ## shorter than Enter+0
> debug(imin)
```

- And then restart the code.

```
> mind(m)
debugging in: FUN(newX[, i], ...)
debug at #1: {
    lx <- length(x)
    i <- x[lx]
    j <- which.min(x[(i + 1):(lx - 1)])
    return(c(j, x[j]))
}
Browse[2]>
```

- Ok, we're in `imin()`.

Lets see if `imin()` properly received the first row of `dd`,

```
Browse[2]> x
[1]  0 12  5  1
```

- Ok, `imin()` getting the right arguments. Stepping thru:

```
Browse[2]> n
debug at #2: lx <- length(x)
Browse[2]> n
debug at #3: i <- x[lx]
Browse[2]> n
debug at #4: j <- which.min(x[(i + 1):(lx - 1)])
Browse[2]> n
debug at #5: return(c(j, x[j]))
```

```
Browse[4]> lx
[1] 4
Browse[4]> i
[1] 1
Browse[4]> j
[1] 2
Browse[4]> x[(i+1):(lx-1)]
[1] 12  5
```

So `j = 2` is indeed the correct index of the minimum of `x[(i+1):(lx-1)]`, but that is not the correct index of the row of `dd`.

- we forgot to add in the row number `i`:

```
Browse[2]> Q
> imin <- function(x) {
+   lx <- length(x)
+   i <- x[lx]
+   j <- which.min(x[(i+1):(lx-1)])
+   k <- j + i
+   return(c(k,x[k]))
+ }
> mind(m)
Error in mind(m) : subscript out of bounds
```

- Oh no! Another bounds error!

```
Enter a frame number, or 0 to exit

1: mind(m)

Selection: 1
Called from: top level
Browse[1]> i
[1] 1
Browse[1]> j
[1] 5
```

- `i` is correct, but the value of `j` is still wrong, it should be `<= 3`.
- `where` indicates line 14 again. Argh!

`wmins` looks correct

```
Browse[1]> wmins
     [,1] [,2]
[1,]    3    3
[2,]    5    8
```

- Ah, but we're using it wrong.
- The first row has indices and the second has values, and we were using them the other way around.

```
i <- which.min(wmins[1,])
j <- wmins[2,i]
```

```
Browse[1]> Q
> mind <- function(d) {
+   n <- nrow(d)
+   dd <- cbind(d,1:n)
+   wmins <- apply(dd[-n,],1,imin)
+   i <- which.min(wmins[2,])
+   j <- wmins[1,i]
+   return(c(d[i,j],i,j))
+ }
> mind(m)
[1] 5 1 3
```

- Golden!

That's about all there is to it.

- More of an art than a science, but good tools help.
- You can set breakpoints at particular lines in files too (see`? setBreakpoint`).
- This calls `trace()` on a particular function, so unsetting breakpoints involves `untrace()`.
- Some prefer this to `debug(f)ing` a whole function `f()`.

Also see:

- The `debug` package
- The `edtdbg` package which works inside Vim and Emacs.
- RStudio's debugging enhancements.

Profiling for speed

If you think your R code is running unnecessarily slowly,

- a handy tool for finding the culprit is `Rprof()`.
- It gives you a report of (approximately) how much time your code is spending in each of the functions it calls.

This is important, as it may not be wise to optimize *every* section of your program.

- Optimization may come at the expense of coding time and code clarity,
- so it is of value to know where optimization would reap the largest dividends.

Consider the following code which computes powers of a vector.

```
> powers1 <- function(x, dg)
+   {
+     pw <- matrix(x, nrow=length(x))
+     prod <- x
+     for(i in 2:dg) {
+       prod <- prod * x
+       pw <- cbind(pw, prod)
+     }
+     return(pw)
+   }
```

Timing the whole function is straightforward, but does not provide a granular report.

```
> x <- runif(1000000)
> system.time(p1 <- powers1(x, 16))
   user  system elapsed
  1.021   0.706   1.730
```

`Rprof()` provides more detail.

- We have to tell it when to start and stop profiling.

```
> Rprof()
> p1 <- powers1(x, 16)
> Rprof(NULL)
```

- This creates a temporary file called `Rprof.out`, the contents of which we will discuss shortly.
- A summary of that output may be obtained by `summaryRprof()`.

```
> summaryRprof()
$by.self
        self.time self.pct total.time total.pct
"cbind"      1.10     87.3       1.10      87.3
"*"          0.16     12.7       0.16      12.7

$by.total
          total.time total.pct self.time self.pct
"powers1"       1.26     100.0      0.00      0.0
"cbind"         1.10      87.3      1.10     87.3
"*"             0.16      12.7      0.16     12.7
...
```

The `$by.self` list sorts functions by the time spent therein, and therein *only*:

- i.e., not accumulating time spent in other functions called therein.
- `$by.total` shows cumulative timings including callees.

Functions which appear near the top of *both* lists are cause for most concern/best targets for extra optimization effort.

- Roughly, we see that `cbind()` is requiring $> 5\times$ the computing effort of other functions.
- Remember the importance of pre-allocation!

Here is a new version, with smart pre-allocation.

```
> powers2 <- function(x, dg)
+   {
+     pw <- matrix(nrow=length(x), ncol=dg)
+     prod <- x ## current product
+     pw[,1] <- prod
+     for(i in 2:dg) {
+       prod <- prod * x
+       pw[,i] <- prod
+     }
+   }
> system.time(p2 <- powers2(x, 16))
   user  system elapsed
  0.390   0.109   0.507
```

- ... and it takes less than one-third of the time. (.R file)

What is `Rprof()` doing?

Every 0.02 seconds (the default value), R

- inspects the call stack to determine which function calls are in effect at that time.
- It writes the result of each inspection to a file (`Rprof.out` by default).
- `summaryRprof()` aggregates those timings (by self and total) for each unique function in the `Rprof.out` file, displaying results for those functions with the largest proportion of times.

This type of profiling technique is called sampling.

For a second, more involved, example consider the code in `ar.R` that provides MCMC samples from the posterior distribution of an auto-regressive model.

- The file `ar_test.R` provides a stress test/demo of the implemented routines.

This illustration will focus on the `ar.gibbs()` method, which requires the following minimal setup.

```
> source("ar.R")
> x <- ts(scan("ar3.txt", quiet=TRUE))
> lx <- length(x)
> XKs <- make.XKs(x, 4)
```

Now for the profiling exercise.

```
> Rprof("ar3.Rprof")
> theta.3 <- ar.gibbs(1000, x[5:1x], Xk=XKs[[3]])
> Rprof(NULL)
> ar3.Rprof <- summaryRprof("ar3.Rprof")
> ar3.Rprof$by.total[1:10,]
  ## output suppressed
> ar3.Rprof$by.self[1:10,]
  ## output suppressed
```

- (See R session)

`rmvnorm()` appears highest, *jointly*, in both lists.

- Some of its subroutines `*all.equal*`, `is.Symmetric*` also appear high in the list(s).

Inspecting `rmvnorm()`, from the `mvtnorm` library, reveals that these functions are used in a sanity checking capacity.

- This is a *very* conservative implementation.
- Checking for symmetry, e.g., is $O(n^2)$.

If we want to be cowboys, and throw caution to the wind, we might get a faster piece of code with a less conservative implementation.

Lets try

```
> rmvnorm <- function(n,mu,sigma)
+   {
+     p <- length(mu)
+     z <- matrix(rnorm(n * p),nrow=n)
+     ch <- chol(sigma,pivot=T)
+     piv <- attr(ch,"pivot")
+     zz <- (z%*%ch)
+     zzz <- 0*zz
+     zzz[,piv] <- zz
+     zzz + matrix(mu,nrow=n,ncol=p,byrow=T)
+ }
```

A new profiling session:

```
> Rprof("ar3.Rprof2")
> theta.3 <- ar.gibbs(1000, x[5:lx], Xk=XKs[[3]])
> Rprof(NULL)
> ar3.Rprof2 <- summaryRprof("ar3.Rprof2")
```

And a comparison:

```
> ar3.Rprof$sampling.time
[1] 0.72
> ar3.Rprof2$sampling.time
[1] 0.28
```

- Woah! Almost 3$\times$ faster!
- Hard to imagine finding that without `Rprof()`.

Profiling for memory usage

`Rprofmem()` does for memory what `Rprof()` does for time.

It works exactly the same way.

1. Call `Rprofmem()` with an optional filename.
2. Run code to profile for memory usage.
3. Call `Rprofmem(NULL)`.
4. Get a summary with `noquote(readLines("Rprofmem.out", n = 5))`.

Unfortunately, however, you need to compile a special R from scratch or you'll get an error message:

```
> Rprofmem()
Error in Rprofmem() : memory profiling is not
   available on this system
```

Also, there is no equivalent to `summaryRprof()`.

- Needless to say, this is not a commonly used technique.

Another option, `tracemem(x)`, causes a message to be printed to the screen whenever `x` is copied.

- But it too requires a custom R compilation.