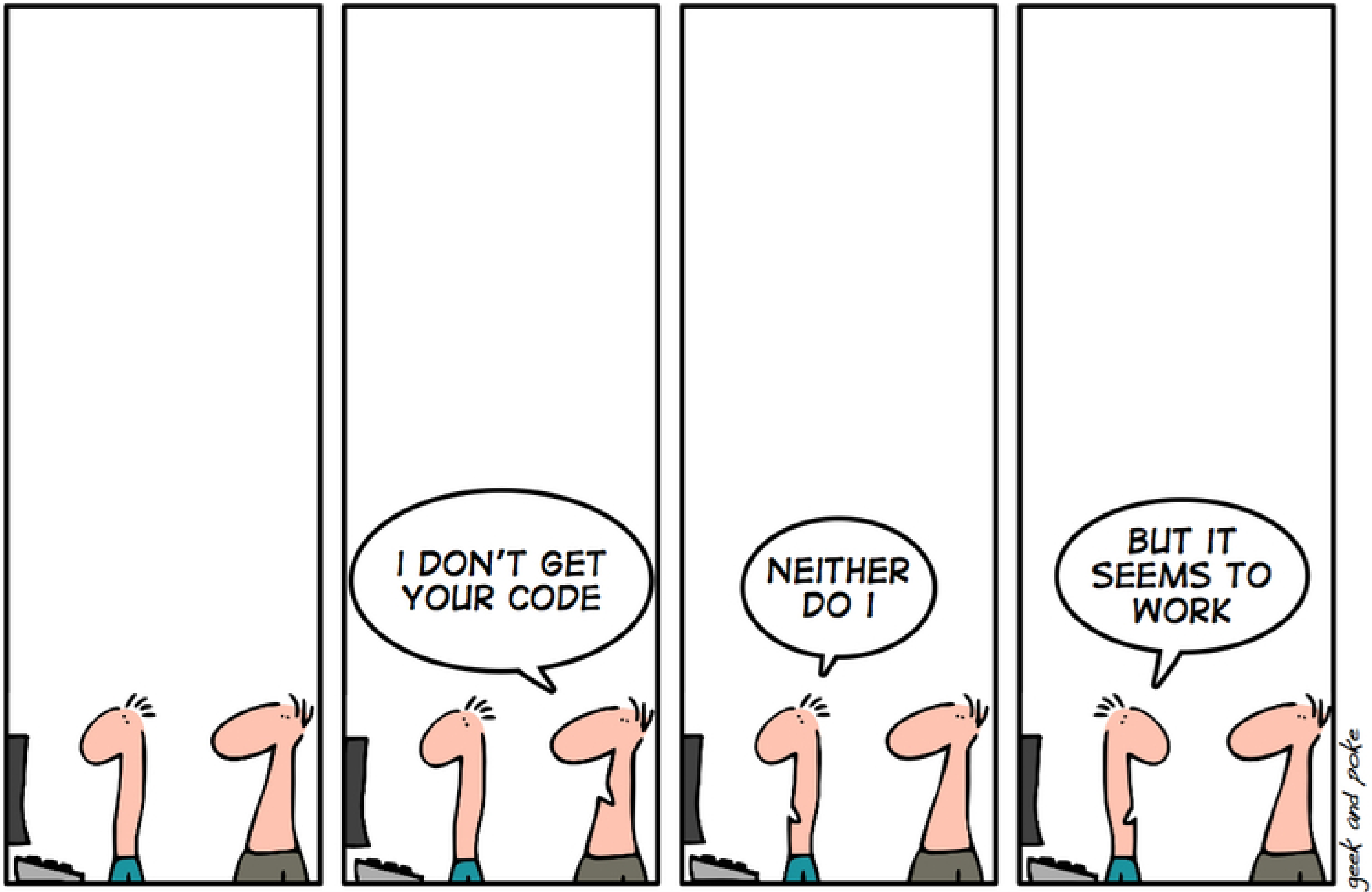

THE ART OF PROGRAMING

geekandpoke.typepad.com

Programming style

(viewed generally)

Karl W Broman

Biostatistics & Medical Informatics
University of Wisconsin – Madison

http://www.biostat.wisc.edu/~kbroman

What statisticians do

- (Think deeply.)
- Write computer programs.
- Analyze data.
- Conduct computer simulations.
- Write papers.

Basic principles

- Code that works
  - No bugs
  - Efficiency is secondary
- Reuseable
  - Modular
  - Reasonably general
- Readable
  - Fixable, extendable
- Reproducible
  - Re-runnable
- Think before you code
  - More thought $\Longrightarrow$ fewer bugs/re-writes
- Learn from others' code
  - R itself
  - R packages

Distributing code

- The only way that your ideas will be used.
- Don't embarass yourself.
- More documentation $\Longrightarrow$ fewer stupid questions ("RTFM").
- People really like tutorials (vignettes in R).
- Learn about software licenses.
  - e.g., GNU Public License

Software for yourself and for others

- Two years from now, “you” become more like an “other”.
- As your number of simultaneous projects increases, the need to make code understandable increases.
- Comment, document (e.g., manual, examples)

  $\Longrightarrow$ Create an R package.

  – Data starts in R; results immediately in R.
  – Make use of R’s functions and math library.
  – Quick and useful documentation plus sample data and demos/vignettes.
- Don’t make things too specific.
  – Functions taking relatively general input.
  – Don’t want to have to edit the code.

“User interface” example

Bad code:

- Referring directly to data in one’s workspace.
- Magic numbers:
  - maximum number of iterations
  - tolerance for convergence
  - other fudge factors

Good code:

```
npmix.em <-
function(y, start, maxk=30, maxit=1000,
         tol=1e-6, trace=FALSE)
{
    ...
}
```

Error/warning messages

- Explain what's wrong (and where).
- Suggest corrective action.
- Give details.
- Don't let the user do something stupid without warning them.

Check data integrity

Check that the input is as expected, or give warnings/errors.

```
npmix.em <-
function(y, start, maxk=30, maxit=1000,
         tol=1e-6, trace=FALSE)
{
    # check the input
    if(!is.list(y))
        stop("y should be a list.")
    if(length(y) < 2)
        stop("y should have length >= 2.")
    if(length(start) != length(y)+3)
        stop("length(start) should = length(y)+3.")

    # get rid of NA's
    n.na <- sum(is.na(unlist(y)))
    if(n.na > 0) {
        warning("Omitting ", n.na, " NAs from the input, y.")
        y <- lapply(y, function(a) a[!is.na(a)])
    }
    ...
}
```

Program organization

- Break code up into separate files (generally $<$2000–3000 lines).
- Files include related functions.
- Files named meaningfully.
- Start each file with a comment saying who wrote it and when, what it contains, and how it fits into the larger program.
- End each file with a comment like "`end of myfile.R`".

No. lines per file in R/qtl

argmax.geno.R	188
calc.genoprob.R	372
calc.pairprob.R	212
discan.R	323
effectplot.R	765
effectscan.R	349
errorlod.R	289
est.map.R	307
est.rf.R	437
fitqtl.R	686
makeqtl.R	1575
plot.R	447
plot.scantwo.R	404
qtlcart_io.R	468
read.cross.R	287
read.cross.csv.R	372
read.cross.gary.R	182
read.cross.karl.R	168
read.cross.mm.R	359
read.cross.qtx.R	226
ripple.R	388
scanone.R	1087
scanqtl.R	437
scantwo.R	1694
sim.geno.R	191
simulate.R	810
summary.cross.R	572
util.R	4117
vbscan.R	194
write.cross.R	430
xchr.R	912

discan.c	268
fitqtl_imp.c	478
hmm_4way.c	596
hmm_bc.c	202
hmm_f2.c	379
hmm_main.c	1163
info.c	94
ripple.c	590
scanone_em.c	245
scanone_em_covar.c	468
scanone_hk.c	292
scanone_imp.c	545
scanone_mr.c	213
scanone_np.c	114
scantwo_em.c	1009
scantwo_hk.c	720
scantwo_imp.c	651
scantwo_mr.c	578
util.c	905
vbscan.c	301

Example file header

```
##############################################################################
# npmix.R
#
# Karl W Broman, Johns Hopkins University
# 29 Nov 2003
#
# Code for getting MLEs via the EM algorithm for a normal/Poisson
# mixture model.  Written as an illustration for the course
# 140.776 (Statistical computing).
#
# The problem:
#
# m groups, n_i values in group i.
# y_ij: quantitative responses (j=1...n_i, i=1...m)
# k_ij: unobserved counts
#
# (k_ij, y_ij) mutually independent
# k_ij ~ Poisson(lambda_i)
# y_ij | k_ij ~ Normal(mean=a + b k_ij, sd = sigma)
#
# Obtain MLEs of a, b, sigma, lambda_1 ..., lambda_m by EM
#
# Contains: npmix.em, npmix.estep, npmix.mstep, npmix.lnlik, npmix.sim
##############################################################################
```

Functions

- Each function should have a single, focused task.
- Give each function a meaningful name.
- If a function starts to get really complicated, consider separating parts out as separate functions. (Think reuse.)

  Ugly example: `scantwo` (4% of R code and 20% of C code in R/qtl)
- Precede each function with a comment regarding its task and the format of the input and output.

The example

```
npmix.em <-
function(y, start, maxk=30,
         maxit=1000, tol=1e-6, trace=FALSE)
{
    ...
    cur <- start
    flag <- 0
    n <- sapply(y, length)

    for(i in 1:maxit) {
        estep <- npmix.estep(y, cur, maxk)
        new <- npmix.mstep(y, estep$ek, estep$eksq, n)

        if(all(abs(new-cur) < tol)) { # converged
            flag <- 1
            break
        }
        cur <- new
    }
    if(!flag) warning("Didn't converge\n")
    ...
}
```

Example function header

```
##############################################################################
# npmix.em: The main function for performing EM.
#
# Input:
#   y:         a list; components are the m groups; values in each
#              component are the y_ij
#
#   start:     a vector of starting values for
#              theta = (a,b,sigma,lambda_1,...,lambda_m)
#
#   maxk:      maximum value of k for sums over poisson counts
#
#   maxit:     maximum number of iterations
#
#   tol:       tolerance for determining convergence
#
#   trace:     FALSE/0: give no tracing info
#              TRUE/1:  print change in ln lik + max change in param
#              2:       print delta lnlik and current param ests
#
# Output: A list containing the following:
#
#   est:       estimate of theta = (a, b, sigma, lambda_1 ... lambda_m)
#   lnlik:     ln likelihood at the estimate
#   ek:        E(k | theta_hat, y)   [a list like "y" in the input]
#   eksq:      E(k^2 | theta_hat, y) [same structure as ek]
#
##############################################################################
```

Clear code

- Find a clear style and stick to it.
- Comment the tricky bits.
- Indent. (4 spaces?)
- Use white space.
- Don't let lines get too long. (72 characters?)
- Use parentheses to avoid ambiguity.
- Conform to traditions.
- Avoid really long statements.
- Balance speed, length, clarity.

Variable/function names

- Descriptive, concise.
- Be consistent.
  (e.g., `n.ind` vs. `nind`)
- Don't use names that are quite similar.
  (e.g., `total` and `totals`)
- Name "magic numbers" in a meaningful way.
  (e.g., `maxit <- 1000` rather than just using 1000)
- Put the most important word first.
  (e.g. `entry.count`, `entry.min`, `entry.max`)
- Use active names for functions.
  (e.g. `calc.lnlik`)

Commenting

- Comment the tricky bits and the major sections.
- Don’t belabor the obvious.
- Don’t comment bad code; re-write it.
- Don’t contradict the code.
- Clarify; don’t confuse.
- Comment code as you are writing it (or even before).
- Plan to spend 1/4 of your time commenting.

Complex data objects

- Keep disparate bits of inter-related data together in a more complex structure.
  - In R, use a `list`.
  - In C, use a `struct`.
- Consider object-oriented stuff.

Avoiding bugs

- Learn to type well.
- Think before you type.
- Consider commenting before coding.
- Code defensively (handle cases that “can’t happen”).
- Write a prototype in a simpler language (e.g. R vs C).
- Code simply and clearly.
- Use modularity to advantage.
- Think through all special cases.
- Don’t be in too much of a hurry.

Finding bugs

- Learn to use debugging tools (and print/cat statements).
- Look for familiar patterns.
- Examine the most recent change.
- Don’t make the same mistake twice.
- Debug now, not later.
- Read before typing.
- Make the bug reproducible.
- Divide and conquer. (Display output to localize your search.)
- Study the numerology of failures.
- Write self-checking code.
- Write trivial programs to test your understanding.
- Keep records of what you’ve done to find the bug.
- Try an independent implementation.
- Consider that your algorithm may be garbage.

Testing

- Test as you write.
- Test code at “the boundaries”.
- Check error returns.
- Consider automation.
- Test simple parts first.
- Get an interested friend to help.

Versions

- Don't destroy what works.
- Don't destroy anything.
- Keep comments up to date.
- Keep track of what you've changed.
- Keep track of what you want to change.
- My approach:
  - An R package numbered like `0.97-21`
  - Update the build number frequently.
  - Files `STATUS.txt`, `TODO.txt`, `BUGS.txt`.
- Adopt a version control system: subversion, git, mercurial

  (I use git and github.com)

Data analysis

- Write functions (and even a package) to automate things.
- Keep track of versions (of data, of functions).
- Keep track of precisely what you did and what you learned.
- Use care regarding commenting out code.
- Use Sweave to automate data analysis and report writing.

  (I also like asciidoc.)

Computer simulations

- You're creating new data; conform to the standards of experimental scientists.
  - Keep track of what you've done (electronically or on paper?)
  - Can you reproduce the results?
- Consider saving random number seeds.
- Save the exact code and its input.
- Create a package.

Writing papers

- Comment the tricky bits in your LaTeX file.
- Use just one file for the text?
- Keep track of versions.
- Organize your references.
  - Number each article sequentially.
  - Enter each into Mendeley (or equivalent) with a numeric label and key-words.
- Consider using `make` to automate compilation.

Organizing your directories

```
~
    ~/Code
    ~/Data
    ~/Docs
    ~/Play
    ~/Projects
    ~/Teaching
```

```
~/Code
    ..../C
    ..../Perl
    ..../Rbroman
    ..../Rgeesibsor
    ..../Rqtl
    ..../Rqtlsim
        ⋮
```

```
~/Docs
    ..../Abstracts
    ..../Grants
    ..../Letters
    ..../Papers
    ..../Posters
    ..../Resumes
    ..../Talks
```

```
~/Projects
    ..../Ari
    ..../Clarke
    ..../Db
    ..../DeMaio
    ..../Fuchs
        ⋮
```

```
~/Projects/PIn/
    ..../Apr01
    ..../Oct01
    ..../Jan02
    ..../May02
    ..../July02
```

```
~/Projects/PIn/July02
    ..../Data
    ..../Notes
    ..../Perl
    ..../R
    ..../Rawdata
    ..../Summary
```

Summary

- Get the correct answers.
- Plan for the future.
- Be organized.
- Don’t be too hurried.
- Learn from others.

Suggested reading

Oliveira and Stewart (2006) Writing scientific software: A guide to good style. Cambridge University Press.

Kernighan and Pike (1999) *The Practice of Programming*. Addison-Wesley.

Kernighan and Plauger (1978) *The Elements of Programming Style*. McGraw-Hill.

Oualline (1992) *C Elements of Style*. M&T Publishing.

Bentley (2000) *Programming Pearls*, 2nd edition. Addison-Wesley.

McConnell (2004) *Code Complete*, 2nd edition. Microsoft Press.

McGuire (1993) *Writing Solid Code*. Microsoft Press.

Martin (2009) *Clean Code*. Prentice Hall.

Hunt and Thomas (2000) *The Pragmatic Programmer*. Addison Wesley.