I don't get your code.

Neither do I.

But it seems to work.

The Art of Programming.
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 $\Rightarrow$ 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 embarrass yourself.
- More documentation $\implies$ 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)
  ➞ 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:

```r
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.

```r
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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>argmax.geno.R</td>
<td>188</td>
</tr>
<tr>
<td>calc.genoprob.R</td>
<td>372</td>
</tr>
<tr>
<td>calc.pairprob.R</td>
<td>212</td>
</tr>
<tr>
<td>discan.R</td>
<td>323</td>
</tr>
<tr>
<td>effectplot.R</td>
<td>765</td>
</tr>
<tr>
<td>effectscan.R</td>
<td>349</td>
</tr>
<tr>
<td>errorlod.R</td>
<td>289</td>
</tr>
<tr>
<td>est.map.R</td>
<td>307</td>
</tr>
<tr>
<td>est.rf.R</td>
<td>437</td>
</tr>
<tr>
<td>fitqtl.R</td>
<td>686</td>
</tr>
<tr>
<td>makeqtl.R</td>
<td>1575</td>
</tr>
<tr>
<td>plot.R</td>
<td>447</td>
</tr>
<tr>
<td>plot.scantwo.R</td>
<td>404</td>
</tr>
<tr>
<td>qtlcart_io.R</td>
<td>468</td>
</tr>
<tr>
<td>read.cross.R</td>
<td>287</td>
</tr>
<tr>
<td>read.cross.csv.R</td>
<td>372</td>
</tr>
<tr>
<td>read.cross.gary.R</td>
<td>182</td>
</tr>
<tr>
<td>read.cross.karl.R</td>
<td>168</td>
</tr>
<tr>
<td>read.cross.mm.R</td>
<td>359</td>
</tr>
<tr>
<td>read.cross.qtx.R</td>
<td>226</td>
</tr>
<tr>
<td>ripple.R</td>
<td>388</td>
</tr>
<tr>
<td>scanone.R</td>
<td>1087</td>
</tr>
<tr>
<td>scanqtl.R</td>
<td>437</td>
</tr>
<tr>
<td>scantwo.R</td>
<td>1694</td>
</tr>
<tr>
<td>sim.geno.R</td>
<td>191</td>
</tr>
<tr>
<td>simulate.R</td>
<td>810</td>
</tr>
<tr>
<td>summary.cross.R</td>
<td>572</td>
</tr>
<tr>
<td>util.R</td>
<td>4117</td>
</tr>
<tr>
<td>vbscan.R</td>
<td>194</td>
</tr>
<tr>
<td>write.cross.R</td>
<td>430</td>
</tr>
<tr>
<td>xchr.R</td>
<td>912</td>
</tr>
<tr>
<td>discan.c</td>
<td>268</td>
</tr>
<tr>
<td>fitqtl_imp.c</td>
<td>478</td>
</tr>
<tr>
<td>hmm_4way.c</td>
<td>596</td>
</tr>
<tr>
<td>hmm_bc.c</td>
<td>202</td>
</tr>
<tr>
<td>hmm_f2.c</td>
<td>379</td>
</tr>
<tr>
<td>info.c</td>
<td>94</td>
</tr>
<tr>
<td>ripple.c</td>
<td>590</td>
</tr>
<tr>
<td>scanone_em.c</td>
<td>245</td>
</tr>
<tr>
<td>scanone_em_covar.c</td>
<td>468</td>
</tr>
<tr>
<td>scanone_hk.c</td>
<td>292</td>
</tr>
<tr>
<td>scanone_imp.c</td>
<td>545</td>
</tr>
<tr>
<td>scanone_mr.c</td>
<td>213</td>
</tr>
<tr>
<td>scanone_np.c</td>
<td>114</td>
</tr>
<tr>
<td>scantwo_em.c</td>
<td>1009</td>
</tr>
<tr>
<td>scantwo_hk.c</td>
<td>720</td>
</tr>
<tr>
<td>scantwo_imp.c</td>
<td>651</td>
</tr>
<tr>
<td>scantwo_mr.c</td>
<td>578</td>
</tr>
<tr>
<td>util.c</td>
<td>905</td>
</tr>
<tr>
<td>vbscan.c</td>
<td>301</td>
</tr>
</tbody>
</table>
# 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

```r
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.
  \[(\text{e.g., } n\text{.ind vs. nind})\]
- Don’t use names that are quite similar.
  \[(\text{e.g., } \text{total and totals})\]
- Name “magic numbers” in a meaningful way.
  \[(\text{e.g., } \text{maxit} \leftarrow 1000 \text{ rather than just using 1000})\]
- Put the most important word first.
  \[(\text{e.g. } \text{entry.count, entry.min, entry.max})\]
- Use active names for functions.
  \[(\text{e.g. } \text{calc.lnlik})\]
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 \texttt{\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 keywords.
- Consider using \texttt{make} to automate compilation.
## Organizing your directories

<table>
<thead>
<tr>
<th>Directory</th>
<th>Code Subdirectories</th>
<th>Docs Subdirectories</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>~ /Code</td>
<td>~ /Docs</td>
</tr>
<tr>
<td>~ /Code</td>
<td>.... /C</td>
<td>.... /Abstracts</td>
</tr>
<tr>
<td>~ /Data</td>
<td>.... /Perl</td>
<td>.... /Grants</td>
</tr>
<tr>
<td>~ /Docs</td>
<td>.... /Rbroman</td>
<td>.... /Letters</td>
</tr>
<tr>
<td>~ /Play</td>
<td>.... /Rgeesibson</td>
<td>.... /Papers</td>
</tr>
<tr>
<td>~ /Projects</td>
<td>.... /Rqtl</td>
<td>.... /Posters</td>
</tr>
<tr>
<td>~ /Teaching</td>
<td>.... /Rqtl sim</td>
<td>.... /Resumes</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>.... /Talks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Directory</th>
<th>Projects Subdirectories</th>
<th>Projects/PIn Subdirectories</th>
<th>Projects/PIn/July02 Subdirectories</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ /Projects</td>
<td>~ / Projects/PIn</td>
<td>~ / Projects/PIn/July02</td>
<td></td>
</tr>
<tr>
<td>~ /Projects/PIn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>~ /Ari</td>
<td>~ /Apr01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~ /Clarke</td>
<td>~ /Oct01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~ /Db</td>
<td>~ /Jan02</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~ /DeMaio</td>
<td>~ /May02</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~ /Fuchs</td>
<td>~ /July02</td>
<td></td>
</tr>
<tr>
<td></td>
<td>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|              |                       |                             |.../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.


