Effective R Programming

Jacob Colvin

February 21, 2009
1 Introduction
   - Motivation

2 R Concepts
   - Language Details

3 Debuging

4 Profiling
   - Tidying R Code

5 Good Code, Bad Code
   - Vectorize!
   - Cumulative Sum
   - DP Code
   - MCMC without Loops!

6 Conclusion
Motivation

- Dispell the myth that R is slower the C/Fortran.
  - As long as you don’t program in R like you did in C/Fortran.
  - As long as your code is not very serial like `cumsum()`.
- Be more productive by learning how to correctly program & debug in R.
  - How to debug without resorting to `print()` statements.
  - How to profile your code to find out why it is actually slow so you don’t bother optimizing the wrong parts.
R is a scripting language, so it takes a lot of work to go from one command to another compared to a compiled language.

So in R you need to avoid for loops and try and do as much work as possible in each command.

For complex commands, R will call the same fortran code, like BLAS, as a native Fortran program would.

Might be more worth your time to tune R with better BLAS libraries like perhaps the ATLAS ones.

R functions are semantically “call by value”, but are implemented in a “copy on write” fashion.

Hence no penalty for passing large objects in function arguments as long as you don’t modify them.
See “Writing R Extensions” Chapter 4

- traceback() or where did my program die?
- browser() or why did my program die?
  - insert browser commands in code like this
    
    ```
    if( sum( is.na(x) ) > 0 ) browser()
    ```
  - Q for quit
  - [Return] to continue program execution until possibly the next call to browser()
Debug Session Example

```r
> x = matrix(rnorm(12), nrow = 2)
> apply(x, 1, function(y) {browser(); sum(y)})

Called from: FUN(newX[, i], ...)

Browse[1] > x
[1,] 0.3290968 0.6182340 0.4220994 -0.9335046 -1.0767550 0.3365174
[2,] 0.6961244 -1.1485500 1.4818257  1.0544334  0.0786361 1.8111323

Browse[1] > y
[1] 0.3290968 0.6182340 0.4220994 -0.9335046 -1.0767550 0.3365174

Browse[1] >

Called from: FUN(newX[, i], ...)

Browse[1] > y
[1] 0.69612441 -1.14854997 1.48182574  1.05443341  0.07863615

Browse[1] >

[1] -0.3043121  3.9736021
>
```
Profiling: or how to not waste your time

See “Writing R Extensions” Chapter 3

> Rprof("boot.out")
> x = mcmc(Itr=1e6)
> Rprof(NULL)

Followed by this at the command prompt:

:> R CMD Rprof boot.out

A quick and dirty version would be use system.time(), but note the use of the <- operator

> system.time(x <- mcmc(Itr=1e6) )
    user  system elapsed
  3.444   0.008   3.482
Say you are given code that was never indented, and/or you want to remove all the comments.

```r
> options(keep.source = FALSE)
> source("myfuns.R")
> dump(ls(all = TRUE), file = "new.myfuns.R")
```

If you really want to add all of the comments back, you can use a merge tool like Kdiff3 to make it happen.
N=1000000
f=function() {
  x=numeric(N)
  for(i in 1:N)
    x[i]=runif(1)
}
g=function() x=runif(N)

> system.time(f())
  user  system elapsed
   14.345   0.032  14.522
> system.time(g())
  user  system elapsed
    0.108   0.012   0.122

  120X improvement in the vector version!
noloop=function(x){
  gen.iter = function(y=0)
    function(x)
      y <<- x+y
    sapply(x,gen.iter())
}

loop = function(x) {
  for( i in 2:length(x) )
    x[i] = x[i] + x[i-1]
  x
}

> rep(c(-1,1),1e6)->x
> system.time(noloop(x)->x1)
  user  system elapsed
  25.249  0.032  25.350

> system.time(loop(x)->x2)
  user  system elapsed
  13.885  0.060  13.981

> system.time(cumsum(x)->x3)
  user  system elapsed
  0.052  0.000  0.054
\[
\begin{align*}
\omega_1 &= z_1 \\
\omega_i &= z_i \prod_{j=1}^{i-1} (1 - z_j)
\end{align*}
\]

dp.stick.1 = function(y,n=1000,alpha=1) {
  z = rbeta(n,1,alpha)
  w = numeric(length(z))
  w[1] = z[1]
  for( i in 2:length(z) )
    w[i] = z[i] * prod(1-z[1:(i-1)])
  w
}

10000 iterations
  user     system    elapsed
  425.2     0.4      426.5
\[
\begin{align*}
\omega_1 &= z_1 \\
\omega_i &= z_i \prod_{j=1}^{i-1} (1 - z_j)
\end{align*}
\]

dp.stick.2 = function(y,n=1000,alpha=1) {
  z = rbeta(n,1,alpha)
  w = numeric(length(z))
  w[1] = z[1]
  for( i in 2:length(z) )
    w[i] = z[i] * w[i-1] / z[i-1] * (1-z[i-1])
  w
}

10000 iterations
  user  system elapsed
  137.521   0.220  139.714
\begin{align*}
\omega_1 &= z_1 \\
\omega_i &= z_i \prod_{j=1}^{i-1} (1 - z_j)
\end{align*}

dp\_stick.3 = function(y,n=1000,alpha=1) {
    z = rbeta(n,1,alpha)
    z/(1-z)*cumprod(1-z)
}

10000 iterations

    user  system elapsed
      7.392   0.248   7.671
\[ \omega_1 = z_1 \]

\[ \omega_i = z_i \prod_{j=1}^{i-1} (1 - z_j) \]

dp.stick.4 = function(y,n=1000,alpha=1) {
  z = rbeta(n,1,alpha)
  z * c(1, cumprod(1-z[-length(z)]))
}

10000 iterations
  user  system elapsed
  7.613  0.124   7.7
$$\omega_1 = z_1$$

$$\omega_i = z_i \prod_{j=1}^{i-1} (1 - z_j)$$

dl \. stick. 5 = function(y,n=1000,alpha=1) {
    z = rbeta(n,1,alpha)
    z/(1-z)*exp(cumsum(log(1-z)))
}

10000 iterations

user  system elapsed
10.253  0.104  10.524
$$\omega_1 = z_1$$

$$\omega_i = z_i \prod_{j=1}^{i-1} (1 - z_j)$$

dp.stick.6 = function(y,n=1000, alpha=1) {
  zz = rbeta(n, alpha, 1)
  (1-zz)/zz*exp(cumsum(log(zz)))
}

10000 iterations
  user    system elapsed
  10.117   0.108   10.24
gibbs.loop.1 = function (Itr=1e5, rho=0.5) {
  mat <- matrix(ncol = Itr, nrow = 2)
  x0 <- 0; y0 <- 0; mat[,1] <- c(x0, y0)
  for (i in 2:Itr) {
    mat[1,i] <- rnorm(1, rho * mat[2,i-1], sqrt(1 - rho^2))
    mat[2,i] <- rnorm(1, rho * mat[1,i ], sqrt(1 - rho^2))
  }
  mat
}

> system.time(gibbs.loop.1()->g1)
  user    system   elapsed
 4.956    0.000    4.981
```r
gibbs.loop.2 = function (Itr=1e5, rho=0.5) {
  mat <- matrix(ncol = Itr, nrow = 2)
  x0 <- 0
  y0 <- 0
  mat[,1] <- c(x0, y0)
  for (i in 2:Itr) {
    x0 <- rnorm(1, rho * y0, sqrt(1 - rho^2))
    y0 <- rnorm(1, rho * x0, sqrt(1 - rho^2))
    mat[,i] = c(x0,y0)
  }
  mat
}
> system.time(gibbs.loop.2() -> g2)
   user  system elapsed
   3.764   0.004   3.779
```
Fastest MCMC Without Loops

gibbs.noloop = function(Itr=1e5, rho=0.5)
{
  gen.gibbs.iter = function(x=0, y=0) # x and y are used as "closures"
    function(t) { # defines what happens inside a MCMC iteration
      y <<- rnorm(1,rho*y, sqrt(1-rho^2)) # basically <- is for dynamic scoping
      x <<- rnorm(1,rho*x, sqrt(1-rho^2)) # and <<- is for static scoping
      c(x,y)
    }
  sapply(integer(Itr),gen.gibbs.iter())
}

> system.time(gibbs.noloop()->g3)
    user  system elapsed
  3.444   0.008   3.482
big example

`:~/R.prog.tutorial$ R CMD Rprof mcmc.out

Each sample represents 0.02 seconds.
Total run time: 3.86 seconds.

Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

<table>
<thead>
<tr>
<th>% total</th>
<th>% self</th>
<th>total seconds</th>
<th>self seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.41</td>
<td>3.76</td>
<td>0.00</td>
<td>0.00</td>
<td>&quot;gibbs.noloop&quot;</td>
</tr>
<tr>
<td>97.41</td>
<td>3.76</td>
<td>0.00</td>
<td>0.00</td>
<td>&quot;sapply&quot;</td>
</tr>
<tr>
<td>96.89</td>
<td>3.74</td>
<td>5.70</td>
<td>0.22</td>
<td>&quot;lapply&quot;</td>
</tr>
<tr>
<td>91.19</td>
<td>3.52</td>
<td>14.51</td>
<td>0.56</td>
<td>&quot;FUN&quot;</td>
</tr>
<tr>
<td>76.68</td>
<td>2.96</td>
<td>70.47</td>
<td>2.72</td>
<td>&quot;rnorm&quot;</td>
</tr>
<tr>
<td>5.18</td>
<td>0.20</td>
<td>0.52</td>
<td>0.02</td>
<td>&quot;unlist&quot;</td>
</tr>
<tr>
<td>4.66</td>
<td>0.18</td>
<td>0.00</td>
<td>0.00</td>
<td>&quot;unique&quot;</td>
</tr>
</tbody>
</table>

...
R is amazing, and if you do something else you are probably wasting your time.

Use R to prototype your projects, and later, if necessary, reimplement the slow functions in C/Fortran

How to call C/Fortran code from R would be a good talk for the future, if I ever find a pressing reason to learn how myself.

Learn how to use the apply family of functions.

Consider this...

What is the ratio of the time spent programming over time spent running the program?

I bet it is over 10, maybe more like 100.

So who cares how slow R is if you can cut programming time in half?