

Efficient *R* Programming

Martin Morgan

Fred Hutchinson Cancer Research Center

17-18 February, 2011

Motivation

Challenges

- ▶ Long calculations: bootstrap, MCMC,
- ▶ Big data: genome-wide association studies, re-sequencing,
- ▶ Long \times big: ...

Solutions

- ▶ Avoid *R* programming pitfalls – *very* significant benefits
- ▶ Large data management
- ▶ Parallel evaluation, especially ‘embarrassingly parallel’ (not discussed in this course)

Programming pitfalls: easy solutions

- ▶ Input only required data

```
> colClasses <-  
+   c("NULL", "integer", "numeric", "NULL")  
> df <- read.table("myfile", colClasses=colClasses)
```

- ▶ Preallocate-and-fill, not copy-and-append

```
> result <- numeric(nrow(df))  
> for (i in seq_len(nrow(df)))  
+   result[[i]] <- some_calc(df[i,])
```

- ▶ Vectorized calculations, not iteration

```
> x <- runif(100000); x2 <- x^2  
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

- ▶ Avoid unnecessary character creation operations, e.g.,
USE.NAMES=FALSE in sapply, use.names=FALSE in unlist.

Programming pitfalls: moderate solutions

- ▶ Use appropriate functions, often from specialized packages.

```
> library(limma) # microarray linear models  
> fit <- lmFit(eSet, design)
```
- ▶ Identify appropriate algorithms, e.g., %in% is $O(N)$, whereas naive might be $O(N^2)$

```
> x <- 1:100; s <- sample(x, 10)  
> inS <- x %in% s
```
- ▶ Use C or Fortran code. Requires knowledge of other programming languages, and how to integrate these into R

Measuring performance: timing

- ▶ Use `system.time` to measure total evaluation time
 - ▶ `gcFirst=TRUE` for 'garbage collection'
- ▶ Use `replicate` to average over invocations

```
> m <- matrix(runif(200000), 20000)
> replicate(5, system.time(apply(m, 1, sum))[[1]])
```

```
[1] 0.144 0.160 0.148 0.160 0.148
```

```
> replicate(5, system.time(rowSums(m))[[1]])
```

```
[1] 0.000 0.000 0.000 0.004 0.000
```

- ▶ Cautionary tale: <http://tinyurl.com/29bd6xv>

Measuring performance: comparison

- ▶ `identical` and `all.equal` ensure that 'optimizations' produce correct results!

```
> res1 <- apply(m, 1, sum)
```

```
> res2 <- rowSums(m)
```

```
> identical(res1, res2)
```

```
[1] TRUE
```

```
> identical(c(1, -1), c(x=1, y=-1))
```

```
[1] FALSE
```

```
> all.equal(c(1, -1), c(x=1, y=-1),
```

```
+           check.attributes=FALSE)
```

```
[1] TRUE
```

Measuring execution time: Rprof

```
> tmpf = tempfile()
> Rprof(tmpf)
> res1 <- apply(m, 1, sum)
> Rprof(NULL); summaryRprof(tmpf)
```

\$by.self

	self.time	self.pct	total.time	total.pct
"apply"	0.16	80	0.20	100
"FUN"	0.02	10	0.02	10
"lapply"	0.02	10	0.02	10
"unlist"	0.00	0	0.02	10

\$by.total

	total.time	total.pct	self.time	self.pct
"apply"	0.20	100	0.16	80
"FUN"	0.02	10	0.02	10
"lapply"	0.02	10	0.02	10
"unlist"	0.02	10	0.00	0

Measuring memory use: `tracemem`

► Enable memory profiling

```
> ~/src/R-devel/configure --help  
> ~/src/R-devel/configure --enable-memory-profiling  
> make -j
```

► Copy-on-change semantics

```
> x <- 1:10; tracemem(x)  
[1] "<0x1b1a8f8>"  
> y <- x          # no change, so no copy  
> x[1] <- 2L      # x, y now differ, so copy  
tracemem[0x1b1a8f8 -> 0x1b1a8a0]:
```


Measuring memory use: `tracemem`

► Copying in *R* functions

```
> l <- list(a=1:10, b=1:10); tracemem(l$a)
[1] "<0x1131ce0>"
> df0 <- as.data.frame(l)
tracemem[0x1131ce0 -> 0x1131bd8]: eval as.data.frame.list a
tracemem[0x1131bd8 -> 0x1131a20]: data.frame eval eval as.o
tracemem[0x1131a20 -> 0x11318c0]: as.data.frame.integer as.
> df1 <- data.frame(a=l$a, b=l$b)
tracemem[0x1131ce0 -> 0x11332c0]: data.frame
tracemem[0x11332c0 -> 0x1133160]: as.data.frame.integer as.
> identical(df0, df1)
[1] TRUE
```

Debugging: browsing and tracing

`browser` start the *browser*

`debug` enter browser when
function invoked

`trace` more flexible
variant of `debug`

`setBreakpoint` arbitrary action
at particular line.

`traceback` see the call stack at
time of last error.

`recover` select call stack
location to invoke
browser.

```
> f <- function() {  
  x = 1  
  browser()  
}  
> f()  
Called from: f()  
Browse[1]> ls()  
[1] "x"  
Browse[1]> x  
[1] 1  
Browse[1]> c
```

See Roger Peng's *An
Introduction to the Interactive
Debugging Tools in R*

Debugging: browsing and tracing

`browser` start the *browser*

`debug` enter browser when
function invoked

`trace` more flexible
variant of `debug`

`setBreakpoint` arbitrary action
at particular line.

`traceback` see the call stack at
time of last error.

`recover` select call stack
location to invoke
browser.

See Roger Peng's *An
Introduction to the Interactive
Debugging Tools in R*

```
> f <- function(x) log(x)
> g <- function(i) f(i-1)
> trace(f,
+ quote(if (x < 0) recover()))
> g(0)
Tracing f(i - 1) on entry
```

Enter a frame number,
or 0 to exit

```
1: g(0)
2: f(i - 1)
```

Selection: 1

```
Called from: eval.parent(expr0
Browse[1]>
```

Debugging: responding to errors

```
options(...)
```

- ▶ warn=2 promote warnings to errors; reset with warn=0
- ▶ error=recover enter recover on error; reset with error=NULL

```
> f <- function() warning("hmm")  
> options(warn=2, error=recover)  
> f()
```

```
Error in f() :  
  (converted from warning) hmm
```

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

```
1: f()  
2: warning("hmm")
```

```
...
```

```
> options(warn=0, recover=NULL)
```

Case study: GWAS

- ▶ Subset of genome-wide association study data
- ▶ Manage sample and SNP annotations in SQL
- ▶ Manage SNP genotypes in netCDF
- ▶ Coordinate access using S4 classes
- ▶ Perform calculations (sliding window composite linkage disequilibrium) in C.
- ▶ Organize as *R* package *StudentGWAS*