Programming in R

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2017

R as a programming language

- R is a free implementation of a dialect of the S language, the interactive statistics and graphics environment developed at Bell Labs.

- R/S are probably the most widely used software for research in statistical methodology and in genomics, and is popular in financial modelling and medical statistics.

- John Chambers won the 1999 ACM Software Systems award for S, which *will forever alter the way people analyze, visualize, and manipulate data*.

- Ross Ihaka won the Royal Society of New Zealand's 2008 Pickering Medal, recognizing *excellence and innovation in the practical application of technology* for the creation of R.

Programmers: a little prehistory

The design of R is largely based on S version 3, which pre-dates Java, Python, JavaScript, Linux, MacOS X, and usable versions of Windows.

Much of the design was fixed in S version 2, which pre-dates C++, Perl, the ANSI C standard, the IBM PC, the GNU project, and Miami Vice.

The basic graphics system is older than Space Invaders.

Yes, some things would be done differently today.

Simulation

This really is how calculations and simulation studies were done!
Simulations have **always** been part of statistical research.

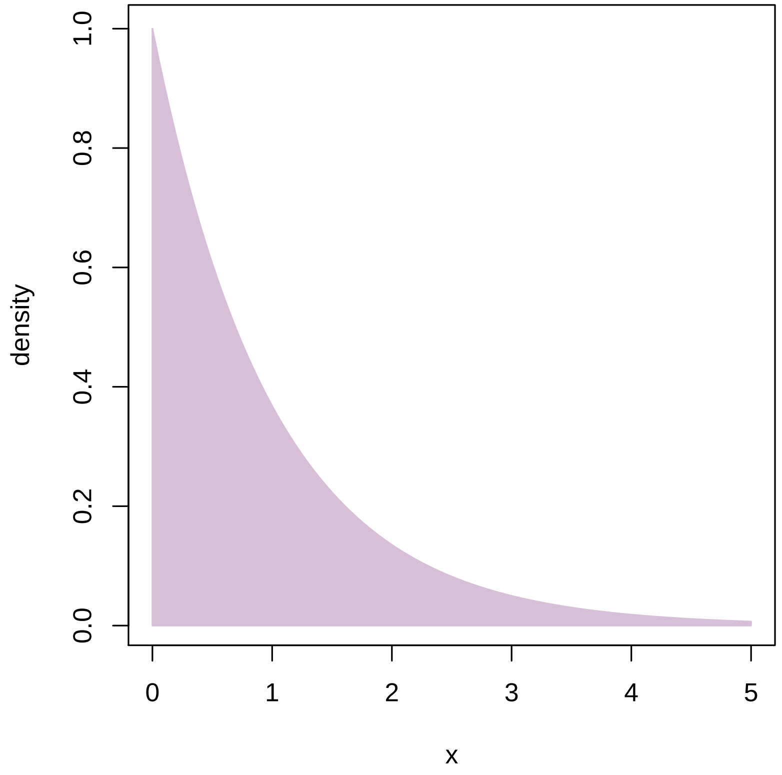
Simulations: a simple example?

Here's a simple problem, for which we *can* work out the exact answer;

For samples of i.i.d $Exp(1)$ data with n=51...
What is the mean value of the sample median?
What is the mean value of the median-squared?

If you had, say, 51 survival times to analyze, from a distribution of times not unlike $Exp(1)$, these are sane questions.

$Exp(1)$ looks like this (right) any guesses?



Simulations: a simple example?

...guessing these would require a lot of luck;

$$\mathbb{E}Y = \frac{2178178936539108674153}{3099044504245996706400}$$

$$\mathbb{E}Y^2 = \frac{2467282316063667967459233232139257976801959}{4802038419648657749001278815379823900480000}$$

- They are 0.70286, 0.51380, to 5 d.p.
- They are *about* 2/3 and 1/2
- 3–4 significant figures is probably enough for most practical purposes. Being *able* compute more accurately is re-assuring
- In the ‘post-genome’ era, being able to compute quickly *is* important (again)

Simulations: a simple example?

Brute force provides perfectly acceptable answers; the `replicate()` function replicates evaluation of an expression

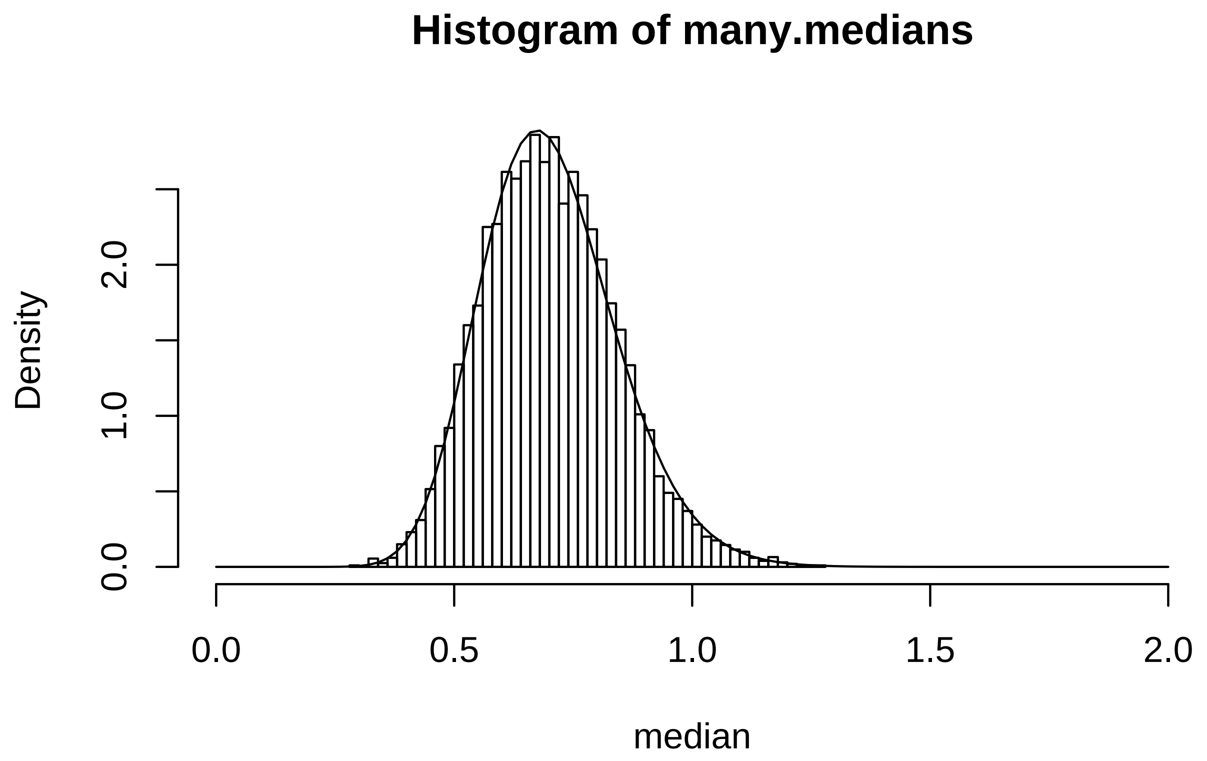
```
> bigB <- bazillion <- 10000
> set.seed(4) # a specific "start" value
> many.medians <- replicate(bigB, { median(rexp(51)) } )
> round( mean(many.medians), 3)
[1] 0.702
> round( mean(many.medians^2), 3)
[1] 0.513
```

The ‘right’ answers averages over an infinite number of replicat-sions. `bigB=10,000` here, which $\approx \infty$.

This calculation takes < 2 seconds, on my desktop

Simulations: a simple example?

Our simulations get us very close to the true distribution of the median;



Having done the 'hard work' of simulation, we can also compute skewness, kurtosis, quantiles, etc – all for 'free'. This technique is very powerful – and often under-rated by statisticians.

Simulations: a simple example?

Here are some other statistical concepts, interpreted in the same way;

- *[If] we simulated data, a bazillion times* $(B \approx \infty)$...”
- *...and applied our procedure to each dataset – and recorded the output*

- Does our estimate usually get close? [consistency]
- How close does our estimate typically get? [bias]
- How variable is our estimate? [standard error, efficiency]
- How often does our interval cover the truth? [coverage]
- How often does our test make a Type I/Type II error? [size/power]

Effective coding

We need to be able to program simulations effectively. A good default for any simulation study follows this ‘pseudo-code’;

```
do.one <- function(n, beta, f){
   ... commands to do one analysis
   ... last command spits out what you want
}

many.sim <- replicate(bigB, do.one(my.n, my.beta, my.f))
   ... commands to work out observed coverage, bias, etc
```

Once this works, wrap it inside further loops, e.g.

```
n.vals        <- c(10, 20, 30, 40, 50, 1000)
coverage.vals <- sapply(n.vals, function(n){
   ... commands to do the replication, with my.n=n
   })
```

At each stage, you must first write a function, *then check it*. This requires a bit of sanity-checking (i.e. trying it where you know *at least roughly* what should happen) and debugging.

Effective coding

The use of `sapply()` (and `apply()`, `lapply()`) can be unfamiliar – many programmers have used `for()` loops elsewhere. `R` does have `for()` loops (see `?Control`) but;

- 'Growing' the dataset is a terrible idea;
  ```
  for(i in 1:n){
  mydata <- cbind(mydata, rnorm(1000, mean=i)) # nooooo!
  }
  ```
  Always set up blank output first, then 'fill it in'
- Use of `replicate()`, `apply()` etc means slightly faster interpretation of code than `for()` – but not by much. `for()` loops are not *intrinsically* evil
- `for()` requires more typing than `replicate()` etc, and is often more work to edit
- Using functions makes your ultimate `R` package easier to produce... right?

Debugging

A ‘handy’ hint from the Apple Corporation;

Take 20 seconds to debug yourself.
A reminder from Apple Wellness

Debugging

Beyond the level of spotting missed commas and mis-matching parentheses, debugging is difficult.

We'll discuss use of `traceback()` and `recover()`, which can help;

```
> # a trite example of traceback()
> f1 <- function(x){ print(x); f2(x) }
> f2 <- function(x){ x + i.dont.exist }
> f1(10) # gives this strange error;
[1] 10
Error in f2(x) : object 'i.dont.exist' not found
> traceback()
2: f2(x)
1: f1(10)
```

The error occurred inside the execution of `f2()`

Debugging

If the error's not obvious, try using `recover()`;

```
options(error=recover) # enter c to close
set.seed(4)
replicate(1000, {
y <- rnorm(10)
x <- rbinom(10, 1, 0.5)
lm1 <- lm(y~x) # regress Y on X
c(coef(lm1)[2], vcov(lm1)[2,2]) # terms of interest
})

# Hint: look at the highest number frame first

#turn it off! turn it off!
options(error=NULL)
```

Use `ls()` to list local objects; the highest frame number is a good place to start

Debugging

`trace()` adds instrumentation to a function

- `trace(rnorm)` prints a message when `rnorm` is started/ended
- `trace(rnorm, recover)` calls the debugger when `rnorm()` is entered.
- `trace(lm, quote(if(all(mf$x==1)) recover()), at=12)` calls the debugger if `mf$x` is all 1s at line 12 of `lm()`

Use `untrace(rnorm)` to remove tracing from `rnorm()`

Exceptions

While you might never see them in practice (due to data cleaning) in simulation studies your replications may produce 'pathological' data, e.g. all X are identical, or all minor allele-carriers smoke. If your regressions estimate differences per allele-copy, adjusting for smoking, it *should* complain.

If this is just too tedious (and rare) to bother fixing, you can use `tryCatch()`;

```
one.glm <- function(outcome, x){
   tryCatch(
      {model <- glm(outcome~x, family=binomial())
       coef(summary(model))[2,]
      },
   error=function(e){rep(NA, 4)} # puts 4 NAs in output
   )
}
```

... but check your simulation output's rates of `NA`-ness. It's better to pre-empt these problems – but this is not easy

Timing

Premature optimization is the root of all evil

Donald Knuth

If you **already have** the capacity to generate reasonably accurate results within a sane time limit, optimizing code is a *waste of effort*

If you need to do things an **order of magnitude** faster, or use your code again (repeatedly) then optimizing your code **may** be worthwhile

To optimize, you need to know;

- What's the bottleneck?
- How much faster can I make that step?

Timing

Obvious bottleneck/easy solution;

Timing

...What's the bottleneck?

Experienced useRs may be able to 'eyeball' this from code; measurement is an **easier and more reliable** approach (!)

To find out how long operations are taking;

- `proc.time()` returns the current time. Save it before a task and subtract from the value after a task.
- `system.time()` times the evaluation of a given expression
- `R` has a **profiler**; this records which functions are being run, many times per second. `Rprof(filename)` turns on the profiler, `Rprof(NULL)` turns it off. `summaryRprof(filename)` reports how much time was spent in each function.

Remember: A 1000-fold speedup in a function used 10% of the time is **less helpful** than a 2-fold speedup in a function used 50% of the time.

Timing

A small example of this in action;

```
# what is taking all the time?
Rprof("deleteme.txt")
   many.sim <- replicate(1000, {
      y <- rnorm(10)
      x <- rbinom(10, 1, 0.5)
      if( all(x==0)|all(x==1))  return(c(NA,NA))
      lm1 <- lm(y~x)
      c(coef(lm1)[2], vcov(lm1)[2,2])
   })
Rprof(NULL) # turn it off! turn it off!
summaryRprof("deleteme.txt")
```

Timing

...How much faster can I make that step?

Some simple tips;

- Pre-process/'clean' your data before analysis; e.g. `sum(x)/length(x)` doesn't error-check like `mean(x)`
- Similarly, use `glm.fit` not `glm` – use matrix calculations in place of `lm()`
- Use vectorized operations, where possible
- Store data as matrices, not data frames (or use the data.table package)
- Delete objects you are finished with

Worked (toy) example

What is the actual p-value distribution under the null hypothesis?

- Small sample size
- Weird distributions
- Multiple (correlated) tests

```
my.sims <- replicate(10000, {
    # make the data
    # analyse the data
})
```

More advanced profiling

- `Rprof()` can also record memory usage
- profvis package displays sampled time and memory use
- `Rprofmem()` gives complete (not sampled) memory allocations
- `tracemem()` reports duplications of an object (eg a large data matrix)

More advanced methods

- Write **small but important** pieces of code in `C`, and call these from `R`
- Run multiple batches. Store your commands in one script file (which you should do anyway) and call it with e.g.

  ```
  R CMD BATCH myscript.R myconsoleoutput.txt &
  ```

  ... and finally assemble all the (saved) results

The second option applies when there is no available speedup; if your `R` session is mostly waiting for `C` to do matrix work, writing the whole thing in `C` offers no important benefit

More advanced: short cuts to C

For a limited number of jobs, it may be worth getting `R` to send a (large) number of generated datasets to `C` simultaneously.

- For example, instead of looping over datasets with $n = 20$ outcomes Y and $n = 20$ covariates X , generate $B \times 20$ matrices $\mathbf{Y}$ and $\mathbf{X}$; using `rowSums(X)`, `rowSums(X*Y)` etc to construct $\hat{\beta}$ avoids `replicate()` or similar
- For large n or large B one can quickly run out of memory
- This is a massive pain! I have only used it productively for one real job – doing 2.5 million cookie-cutter meta-analyses
- Less of a pain is `cor(large.matrix)` – for all pairwise correlations of columns of `large.matrix`, where all the looping is done in `C`

For complex methods, this approach may not help.

Bonus tracks: how big?

Q. What's the 'Monte Carlo' error in my estimates?

One quick-and-dirty measure of uncertainty is given by these intervals;

```
many.thetahat <- replicate(bigB, {...calculate an estimate...} )
lm1 <- lm(many.thetahat~1)
confint.default(lm1)
```

For binary outcomes, (i.e. when you want coverage, size, power)

```
z <- replicate(bigB, {... calculate theta.hat/est.std.err ...})
mean( z^2 < 1.96^2 ) # how many give p>0.05?
lm2 <- lm( I(z^2 < 1.96^2) ~ 1 )
confint.default(lm2)
```

For GWAS-style levels of e.g. 5×10^{-8}, simulations with e.g. $B = 10^{10}$ may be needed; efficient coding of them can save many days of processor time.