Simulation

Roger D. Peng, Associate Professor of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Generating Random Numbers

Functions for probability distributions in R

- `rnorm`: generate random Normal variates with a given mean and standard deviation
- `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- `pnorm`: evaluate the cumulative distribution function for a Normal distribution
- `rpois`: generate random Poisson variates with a given rate

Generating Random Numbers

Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

- `d` for density
- `r` for random number generation
- `p` for cumulative distribution
- `q` for quantile function

Generating Random Numbers

Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

If Φ is the cumulative distribution function for a standard Normal distribution, then `pnorm(q)` = $\Phi(q)$ and `qnorm(p)` = $\Phi^{-1}(p)$.

Generating Random Numbers

```
> x <- rnorm(10)
> x
 [1] 1.38380206 0.48772671 0.53403109 0.66721944
 [5] 0.01585029 0.37945986 1.31096736 0.55330472
 [9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
 [1] 23.38812 20.16846 21.87999 20.73813 19.59020
 [6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.32   19.73   20.55   20.67   21.67   23.39
```

Generating Random Numbers

Setting the random number seed with `set.seed` ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

Always set the random number seed when conducting a simulation!

Generating Random Numbers

Generating Poisson data

```
> rpois(10, 1)
 [1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
 [1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
 [1] 20 11 21 20 20 21 17 15 24 20

> ppois(2, 2)  ## Cumulative distribution
[1] 0.6766764  ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347   ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662  ## Pr(x <= 6)
```

Generating Random Numbers From a Linear Model

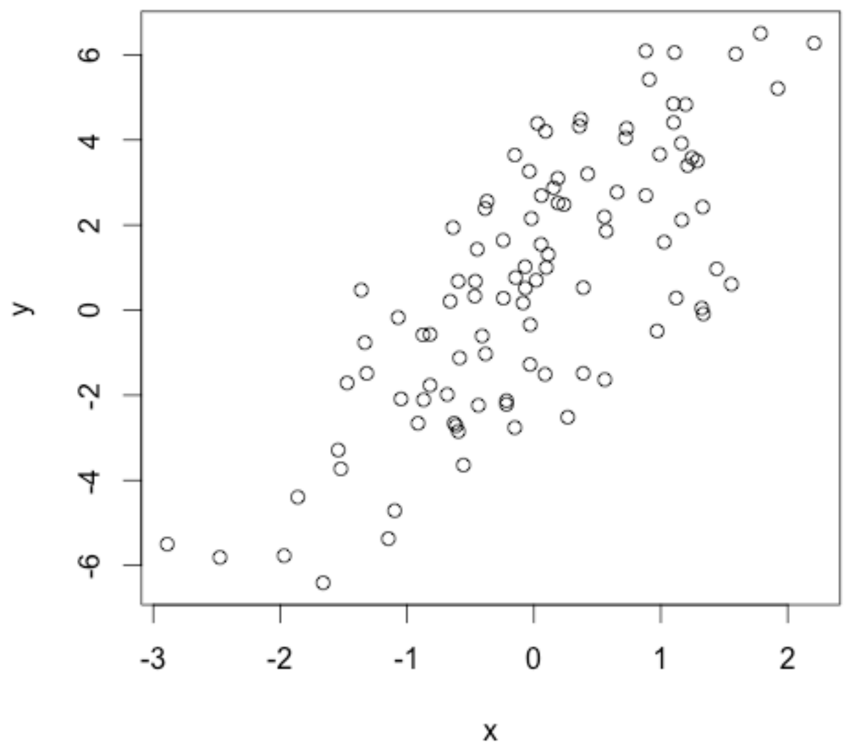
Suppose we want to simulate from the following linear model

$$y = \beta_0 + \beta_1 x + \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```

Generating Random Numbers From a Linear Model

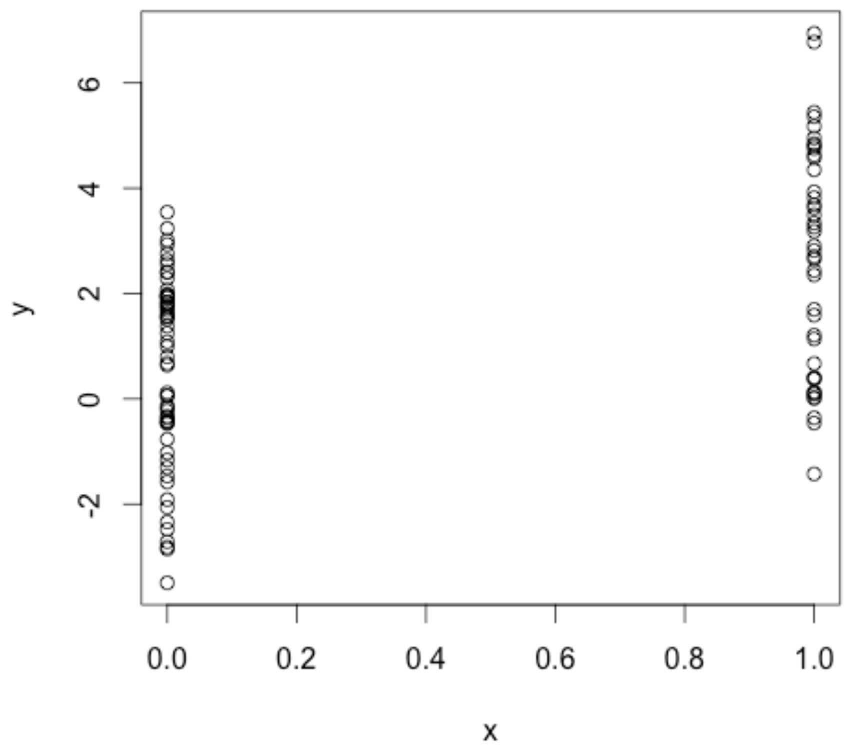


Generating Random Numbers From a Linear Model

What if `x` is binary?

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```

Generating Random Numbers From a Linear Model



Generating Random Numbers From a Generalized Linear Model

Suppose we want to simulate from a Poisson model where

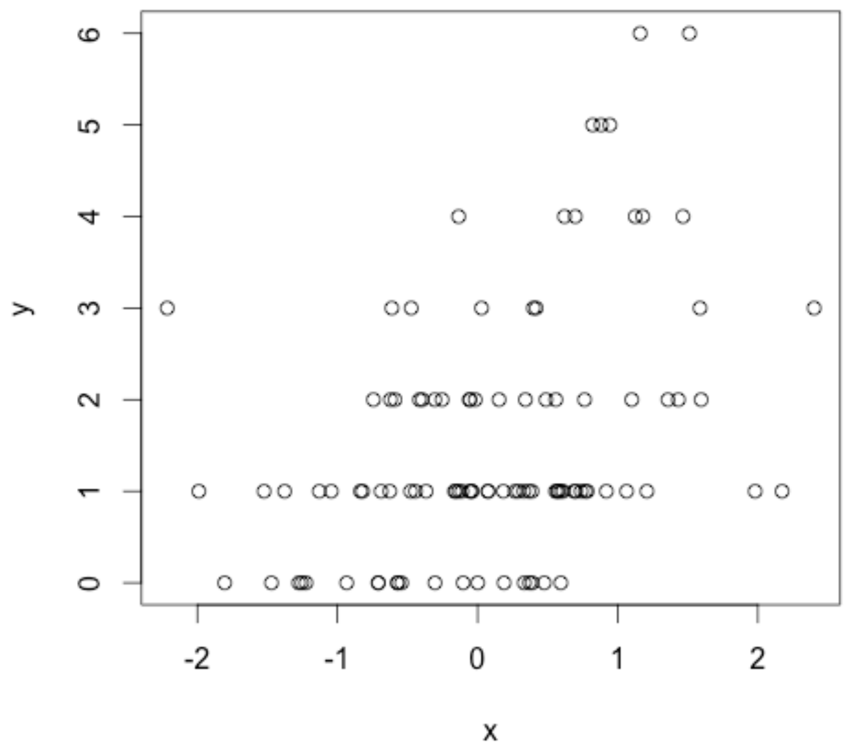
Y ~ Poisson(μ)

log μ = $\beta_0 + \beta_1 x$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$. We need to use the `rpois` function for this

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   0.00    1.00    1.00    1.55    2.00    6.00 
> plot(x, y)
```

Generating Random Numbers From a Generalized Linear Model



Random Sampling

The `sample` function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
> sample(1:10)  ## permutation
 [1] 4 710 6 9 2 8 3 1 5
> sample(1:10)
 [1]  2  3  4  1  9  5 10  8  6  7
> sample(1:10, replace = TRUE)  ## Sample w/replacement
 [1] 2 9 7 8 2 8 5 9 7 8
```

Simulation

Summary

- Drawing samples from specific probability distributions can be done with `r*` functions
- Standard distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc.
- The `sample` function can be used to draw random samples from arbitrary vectors
- Setting the random number generator seed via set.seed is critical for reproducibility

Profiling R Code

Roger D. Peng, Associate Professor of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Why is My Code So Slow?

- Profiling is a systematic way to examine how much time is spend in different parts of a program
- Useful when trying to optimize your code
- Often code runs fine once, but what if you have to put it in a loop for 1,000 iterations? Is it still fast enough?
- Profiling is better than guessing

On Optimizing Your Code

- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time
- This cannot be done without performance analysis or profiling

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil
--Donald Knuth

General Principles of Optimization

- Design first, then optimize
- Remember: Premature optimization is the root of all evil
- Measure (collect data), don’t guess.
- If you're going to be scientist, you need to apply the same principles here!

Using `system.time()`

- Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression
- Computes the time (in seconds) needed to execute an expression
  - If there’s an error, gives time until the error occurred
- Returns an object of class `proc_time`
  - **user time**: time charged to the CPU(s) for this expression
  - **elapsed time**: "wall clock" time

Using `system.time()`

- Usually, the user time and elapsed time are relatively close, for straight computing tasks
- Elapsed time may be *greater than* user time if the CPU spends a lot of time waiting around
- Elapsted time may be *smaller than* the user time if your machine has multiple cores/processors (and is capable of using them)
  - Multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL)
  - Parallel processing via the **parallel** package

Using `system.time()`

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
   user  system elapsed
  0.004   0.002   0.431

## Elapsed time < user time
hilbert <- function(n) {
        i <- 1:n
        1 / outer(i - 1, i, "+”)
}
x <- hilbert(1000)
system.time(svd(x))
   user  system elapsed
  1.605   0.094   0.742
```

Timing Longer Expressions

```
system.time({
    n <- 1000
    r <- numeric(n)
    for (i in 1:n) {
        x <- rnorm(n)
        r[i] <- mean(x)
    }
})
```

```
##    user  system elapsed
##   0.097   0.002   0.099
```

Beyond `system.time()`

- Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amounts of time
- Assumes you already know where the problem is and can call `system.time()` on it
- What if you don’t know where to start?

The R Profiler

- The `Rprof()` function starts the profiler in R
  - R must be compiled with profiler support (but this is usually the case)
- The `summaryRprof()` function summarizes the output from `Rprof()` (otherwise it's not readable)
- DO NOT use `system.time()` and `Rprof()` together or you will be sad

The R Profiler

- Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spend in each function
- Default sampling interval is 0.02 seconds
- NOTE: If your code runs very quickly, the profiler is not useful, but then you probably don't need it in that case

R Profiler Raw Output

```
## lm(y ~ x)

sample.interval=10000
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
```

Using `summaryRprof()`

- The `summaryRprof()` function tabulates the R profiler output and calculates how much time is spend in which function
- There are two methods for normalizing the data
- "by.total" divides the time spend in each function by the total run time
- "by.self" does the same but first subtracts out time spent in functions above in the call stack

By Total

```
$by.total
                        total.time total.pct self.time self.pct
"lm"                          7.41    100.00      0.30     4.05
"lm.fit"                      3.50     47.23      2.99    40.35
"model.frame.default"         2.24     30.23      0.12     1.62
"eval"                        2.24     30.23      0.00     0.00
"model.frame"                 2.24     30.23      0.00     0.00
"na.omit"                     1.54     20.78      0.24     3.24
"na.omit.data.frame"          1.30     17.54      0.49     6.61
"lapply"                      1.04     14.04      0.00     0.00
"[.data.frame"                1.03     13.90      0.79    10.66
"["                           1.03     13.90      0.00     0.00
"as.list.data.frame"          0.82     11.07      0.82    11.07
"as.list"                     0.82     11.07      0.00     0.00
```

By Self

```
$by.self
                        self.time self.pct total.time total.pct
"lm.fit"                     2.99    40.35       3.50     47.23
"as.list.data.frame"         0.82    11.07       0.82     11.07
"[.data.frame"               0.79    10.66       1.03     13.90
"structure"                  0.73     9.85       0.73      9.85
"na.omit.data.frame"         0.49     6.61       1.30     17.54
"list"                       0.46     6.21       0.46      6.21
"lm"                         0.30     4.05       7.41    100.00
"model.matrix.default"       0.27     3.64       0.79     10.66
"na.omit"                    0.24     3.24       1.54     20.78
"as.character"               0.18     2.43       0.18      2.43
"model.frame.default"        0.12     1.62       2.24     30.23
"anyDuplicated.default"      0.02     0.27       0.02      0.27
```

`summaryRprof()` Output

```
$sample.interval
[1] 0.02

$sampling.time
[1] 7.41
```

Summary

- `Rprof()` runs the profiler for performance of analysis of R code
- `summaryRprof()` summarizes the output of `Rprof()` and gives percent of time spent in each function (with two types of normalization)
- Good to break your code into functions so that the profiler can give useful information about where time is being spent
- C or Fortran code is not profiled