03 Random Numbers and Functions

Generating Random Data

- `R` has several functions for generating random data,
  - Built-in statistical distribution functions
  - `sample()` function

```
> sample(x, size, replace=FALSE, prob=NULL)
```

- The `sample()` function takes a sample of size `size` from a vector `x` either with or without replacement. Optionally a vector of probabilities for obtaining the elements of `x`, `prob`, can be supplied.
- If `replace` is `FALSE`, the probabilities are updated after each sample is drawn, that is the probability of choosing the next item is proportional to the weights for the remaining items.

```
> set.seed(1234)
> sample(1:10)
 [1] 10 6 5 4 1 8 2 7 9 3
> sample(1:10, replace=TRUE)
 [1] 10 6 4 8 4 4 5 8 4 8
> sample(1:10, 5)
 [1] 3 4 7 8 5
> sample(1:10, 15)

Error in sample.int(length(x), size, replace, prob) :
  cannot take a sample larger than the population when
  'replace = FALSE'

> sample(1:10, 15, replace=TRUE)
 [1] 10 5 2 8 4 3 7 9 3 6 4 8 10 2 5
> sample(1:5,10, replace=T, prob=c(.6,.2,.1,.05,.05))
 [1] 2 1 3 3 1 1 1 1 2 1
```

```
> set.seed(1111)
> m <- sample(c("A", "B", "C"), 1000, replace = TRUE,
              prob = c(0.1, 0.3, 0.6))

> table(m)

m
  A   B   C
116 297 587

> s <- sample(1:100, replace=TRUE)
> length(unique(s))
[1] 68
> m <- matrix(1:10, 10, 100)
> t(apply(m, 2, sample))
```

Exercise

- Suppose that a discrete random variable X_i has a following probability distribution,

x	-1	0	1	2
$P(X_i = x)$	$\frac{3}{10}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{5}$

Compute

$$\frac{1}{N} \sum_{i=1}^{N} X_i$$

for $N = 10,000$.

Statistical Distributions

- `R` has several built-in statistical distributions. For each distribution four functions are available,
  - `r` : Random number generator
  - `d` : Density function
  - `p` : Cumulative distribution function
  - `q` : Quantile function
- Each letter can be added as a prefix to the `R` distributions;

R	Distribution	R	Distribution
`beta`	Beta	`norm`	Normal
`exp`	Exponential	`unif`	Uniform
`t`	T	`f`	F
`biom`	Binomial	`chisq`	Chi-square
`gamma`	Gamma	`pois`	Poisson
`hyper`	Hypergeometric	`cauchy`	Cauchy

```
> # The density, cumulative distribution, quantile
> # and random number generator functions for
> # the standard normal distribution
> dnorm(1.96, mean=0, sd=1) # Density
[1] 0.05844094
> # Distribution (lower tail)
> pnorm(1.96, mean=0, sd=1)
[1] 0.9750021
> # Distribution (upper tail)
> pnorm(1.96, mean=0, sd=1, lower.tail=FALSE)
[1] 0.02499790
> qnorm(0.975, mean=0, sd=1) # Quantile
[1] 1.959964
> rnorm(4, mean=0, sd=1) # Random Number
[1] -0.3108813 -0.1518240 -0.4871378 -0.6400816
```

Exercise

- Suppose that we have, probability density function of x_i is given by

$$f(x_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$$

for $i = 1, \ldots, n$. Generate 100,000 random values x_i when $\mu = 0$ and $\sigma = 1$

1. Compute five number summary of 10,000 x_i values
2. Compute sample mean and sample median of 10,000 x_i values
3. Compute sample variance and sample standard deviation.
4. Compute

$$\frac{1}{N}\sum_{i=1}^{N} I(x_i < 1.965)$$

where $N = 100,000$.

Seed Number for Random Data

- For a simulation to be repeatable we need to specify the type of random number generator and the initial state of the generator.
- R has several kinds of generators.
- The simplest way to specify the initial state or seed is to use,

  ```
  > set.seed(seed)
  ```

  - The argument seed is a single integer value
  - Different seeds give different pseudo-random values
  - Calling `set.seed()` with the same seed produces the same results, if the sequence of calls is repeated exactly.
- If a seed is not specified then the random number generator is initialized using the time of day.

```
> set.seed(17632)
> runif(4)
[1] 0.03288684 0.88861821 0.21466744 0.32907126
> rnorm(4)
[1] 0.7797193 -0.6107181 -0.2786504 -0.6763935
>
> set.seed(89432)
> runif(4)
[1] 0.175990418 0.258567422 0.007370616 0.286017248
>
> set.seed(17632)
> runif(4)
[1] 0.03288684 0.88861821 0.21466744 0.32907126
> rnorm(4)
[1] 0.7797193 -0.6107181 -0.2786504 -0.6763935
```

```
> set.seed(17632)
> rnorm(4)
[1] -1.8399628 -0.7903303 0.7797193 -0.6107181
> runif(4)
[1] 0.3902566 0.7163214 0.2493954 0.8665002
```

```
> set.seed(123456)
> m <- sample(c("A", "B", "C"), 1000, replace = TRUE,
              prob = c(0.1, 0.3, 0.6))
> table(m)


m
  A   B   C
 84 309 607
```

Control Structures: Conditions and Loops

`if()...else`	Determine which set of expressions to run depending on whether or not a condition is `TRUE`
`ifelse()`	Do something to each element in a data structure depending on whether or not a condition is `TRUE` for that particular element
`switch()`	Evaluate different expressions depending on a given value
`for()`	Loop for a fixed number of iterations
`while()`	Loop until a condition is `FALSE`
`repeat`	Repeat until iterations are halted with a call to `break`
`break`	Break out of a loop
`next`	Stop processing the current iteration and advance the looping index

Condition Structures

- The `condition` needs to evaluate to a single logical value.
- Brackets { } are not necessary if you only have one expression and/or the `if() ... else` statement are on one line.

```
> if (condition) expression # if TRUE
> if (condition) {
+     expressions # if TRUE
+ }
> if (condition) .... # if TRUE
  else .... # if FALSE
> if (condition) {
+     expressions # if TRUE
+ } else {
+     expressions # if FALSE
+ }
```

Condition Structures

- Calculate the median of a random sample $X_1, \ldots, X_n$,

$$\text{median}(X) = \begin{cases} \frac{1}{2}X_{(\frac{n}{2})} + \frac{1}{2}X_{(1+\frac{n}{2})}, & \text{if } n \text{ is even} \\ X_{(\frac{n+1}{2})} & \text{if } n \text{ is odd} \end{cases}$$

where $X_{(1)} \leq X_{(2)} \leq \ldots \leq X_{(n)}$ denote order statistics.

```
> x <- c(5,4,2,8,9,10)
> n <- length(x)
> sort.x <- sort(x)
> if(n %% 2 == 0) {
     median <- (sort.x[n/2] + sort.x[1+n/2]) / 2
  }
  else median <- sort.x[(n+1)/2]
> median
```

Condition Structures

- `ifelse()` returns a value with the same structure as test which is filled with elements selected from either yes or no depending on whether the element of test is `TRUE` or `FALSE`.

  ```
  ifelse(test, yes, no)
  ```

  ```
  > x <- seq(0, 2, length.out=6)
  > x
  [1] 0.0 0.4 0.8 1.2 1.6 2.0
  > ifelse(x<=1, "small", "big")
  [1] "small" "small" "small" "big" "big" "big"
  > round(ifelse(x<=1, exp(x), log(x)), 4)
  [1] 1.0000 1.4918 2.2255 0.1823 0.4700 0.6931
  ```

```
> y <- matrix(1:8, nrow=2)
> y

     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

> ifelse(y>3 & y<7, 1, 0)

     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    1    1    0

> ifelse(y<3 | y>7, 1, 0)

     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    1    0    0    1
```

Loops: `for`

- `for` loops repeat command(s) particular times

```
> for (var in seq) {
+ expressions
+ }
```

- `var` is the name of the loop variable that changes with each iteration
- `seq` is an expression evaluating to any type of vector
- With each iteration `var` takes on the next value in `seq`. At the end of the loop `var` will equal the last value of `seq`.
- The number of iterations equals the length of `seq`
- The `{ }` are only necessary if there is more than one expression

```
> # Calculate 10!  using a for loop

> f <- 1
> for (i in 1:10) f <- f * i
> f
[1] 3628800
> factorial(10)
[1] 3628800

> f <- 1
> fac <- NULL
> for (i in 1:10) {
     f <- f * i
     fac[i] <- f
  }
> fac
> matrix(fac, ncol=1)
```

```
> # Calculate the sum of upper triangular of a matrix
> set.seed(1010)
> x <- matrix(rnorm(5 * 5), 5, 5)
> t <- 0
> for (i in 1:4) {
    for (j in ((i + 1):5)) {
       t <- t + x[i, j]
    }
  }
>

> t
[1] -0.7673945
> sum(x[upper.tri(x)])
[1] -0.7673945
```

Loops: `while`

- `while` loops repeat command(s) until condition fails
  ```
  > while (condition) {
  + expressions
  + }
  ```
- `condition` is a single logical value that is not `NA`
- The `{ }` are only necessary if there is more than one expression
- If you are going to use a `while` loop, you need to have an indicator variable `i` and change its value within each iteration. Otherwise you will have an infinite loop.

```
> # Calculate 10! using a while loop
> i <- f <- 1
> fac <- NULL
> while (i <= 10) {
    f <- f * i
    fac[i] <- f
    i <- i + 1
  }
> fac
> matrix(fac, ncol=1)
>
> # infinite loop
> n <- 10
> while (n > 0) {
    n <- n + 1
  }
```

Loops: `repeat`

- `repeat` loops repeat command(s) until `break`

```
> repeat {
+ expressions
+ if (condition) break
+ }
```

- The `repeat` loop does not contain a limit. Therefore it is necessary to include an `if` statement with the `break` command to make sure you do not have an infinite loop.

```
> i <- f <- 1
> fac <- NULL
> repeat {
    f <- f * i
    fac[i] <- f
    i <- i + 1
    if (i > 10) break
  }
```

Exercise

- Suppose that we want to compute cumulative sum of the following two sequences

$$x = (1, 2, 3, 4, \ldots, 100)$$

and

$$y = (1, 3, 5, \ldots, 99)$$

1. Use `for` loop function
2. Use `while` loop function

Avoiding Loops

- It is a good idea to try and avoid including loops in programs.
- Code that takes a "whole object" approach is usually faster than an iterative approach.
- The `apply()` family of functions does not reduce the number of function calls. So, using `apply()` may not necessarily improve efficiency.
- However, `apply()` is still a great function for making code more transparent and compact.
- When developing code it may be easier to first write an algorithm using a loop.
- But, try and replace the loop with a more efficient expression after you have a draft program.
- Loops are still useful tools, such as simulation studies.

Timing Comparison

- Consider an extreme example, suppose we want the sum of 10 million random standard normal numbers. The `sum()` function, which uses the whole vector, is noticeably faster then the `for` loop, which uses each element.

```
> z <- rnorm(1E7)
> system.time(sum(z))
>
> t <- 0
> system.time(for(i in 1:length(z)) t <- t + z[i])
```

- The `system.time()` prints user time, system time, and elapsed time. Usually elapsed time is the most useful. This is the real elapsed time since the process was started.

Timing Comparison

- Compare a row sum of a large matrix in 3 different ways.

```
M <- matrix(runif(1E7), 100000, 100)

R1 <- NULL
start <- proc.time()
for (i in 1:nrow(M)) {
    ss <- 0
    for (j in 1:ncol(M)) {
        ss <- ss + M[i,j]
    }
    R1[i] <- ss
}
time1 <- proc.time() - start
time1
```

Timing Comparison

```
R2 <- NULL
start <- proc.time()
for (i in 1:nrow(M)) {
    R2[i] <- sum(M[i,])
}
time2 <- proc.time() - start
time2
```

```
start <- proc.time()
R3 <- apply(M, 1, sum)
time3 <- proc.time() - start
time3
```

Exercise

- Suppose that we sequentially roll a fair die and calculate a score in the following rule.
  1. The initial score $S_0 = 100$.
  2. The k-th outcome is $R_k \in \{1, 2, 3, 4, 5, 6\}$
  3. If the k-th outcome is an odd number, $S_k = S_{k-1} - R_k$
  4. If the k-th outcome is an even number, $S_k = S_{k-1} + R_k$

  Compute S_k if $k = 1,000$.

Functions

- R functions are objects that evaluate multiple expressions using arguments that are passed to them. Typically an object is returned.
- To declare a function,

```
> function name <- function(argument list) {
+                  body
+                  }
```

- The argument list is a comma-separated list of formal argument names,
  - `name`
  - `name = default value`
  - `...`
- The `...` is a list of the remaining arguments that do not match any of the formal arguments.

Functions

- Generally, the body is a group of `R` expressions contained in curly brackets `{ }`. If the body is only one expression the curly brackets are optional.
- Functions are usually assigned names, but the names are optional (i.e. the `FUN` argument of `apply()`).
- Be careful with naming functions, could overwrite existing `R` functions!
- Since a function is an object we can pass it as an argument to other functions just like any other object.

Functions

- Often we will want a function to return an object that can be assigned. a function for returning objects is `return()`.
- The `return()` function prints and returns its arguments.
- If the end of a function is reached without calling `return()`, the value of the last evaluated expression is returned.
- A `list` is often a good tool for returning multiple objects.
- Use `cat()` or `print()` to output text.

Example

- A function computing the L_p-norm

$$\left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}} \qquad \text{for} \qquad x \in \mathbb{R}^n$$

```
> pnorm <- function(x, p=2) {
            y <- abs(x)
            z <- sum(y^p)
            return(z^(1/p))
         }
> vec <- rnorm(10)
> pnorm(vec) # p is not defined, so p = 2 by default
> pnorm(vec, 1)
> pnorm(vec, 1/2)
```

Example

- A Fibonacci sequence is defined by

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

It is $F_n = F_{n-1} + F_{n-2}$, where $F_1 = F_2 = 1$. List the first n terms in Fibonacci sequence.

```
> fibonacci <- function(n) {
                 x <- numeric(n)
                 x[1:2] <- 1
                 for(i in 3:n) {
                    x[i] = x[i-2] + x[i-1]
                 }
                 return(x)
               }
> fibonacci(40)
```

Exercise

- Build a function the previous exercise problem. In the new function, the input variables are a grid of k values and the output variables are the corresponding values of R_k and S_k.

Suppose that we sequentially roll a fair die and calculate a score in the following rule. 1. The initial score $S_0 = 100$. 2. The k-th outcome is $R_k \in \{1, 2, 3, 4, 5, 6\}$ 3. If the k-th outcome is an odd number, $S_k = S_{k-1} - R_k$ 4. If the k-th outcome is an even number, $S_k = S_{k-1} + R_k$ Compute S_k if $k = 1,000$.

The Newton Method for Root Finding

- Suppose we wish to find a root of an algebraic equation

$$f(x) = 0$$

- If $f(x)$ has a derivative $f'(x)$, then the following iteration will in general converge to a root if a starting value is close enough to the root.

$$\begin{aligned} x_0 &= \text{initial guess} \\ x_n &= x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \end{aligned}$$

- The algorithm runs until $|f(x_n)| < \epsilon$.
- The idea is based on the Taylor approximation.

$$f(x_n) \approx f(x_{n-1}) + (x_n - x_{n-1}) f'(x_{n-1})$$

- Note that this method may fail to converge

Example of Newton's method

- Compute a square-root of A, i.e.,

$$f(x) = x^2 - A = 0$$

```
> root1 <- function(A, n=10) {
            x <- 1
            for (i in 1:n) x <- x - (x^2-A)/(2*x)
            x
            }
> root2 <- function(A, tol=1e-10) {
            x <- 1
            while(abs(x^2-A) > tol)
            x <- x - (x^2-A)/(2*x)
            x
            }
```

Example of Newton's method

- Suppose $f(x) = x^3 + 2x^2 - 7$. Find a real root of $f(x)$.

```
> f <- function(x) x^3 + 2*x^2 - 7
> f.prime <- function(x) 3*x^2 + 4*x
> Newton <- function(x, tol=1e-10) {
            while(abs(f(x)) > tol) {
                x <- x - (f(x) / f.prime(x))
            }
            return(x)
            }
> Newton(4)
[1] 1.428818
> x0 <- Newton(4)
> f(x0)
[1] 2.770761e-11
```

Variable Scope

- When an object is evaluated within a function, R first looks locally within the function and its arguments for the object. If R cannot find the name of the object locally it looks in the parent(global) environment.
- Therefore, an object within a function can have the same name as an object in the global environment; because R will choose to use the object defined in the local environment.
- The bottom line is that objects defined in the global environment are also accessible by a function. However, objects defined within a function are not accessible from the global environment.

Example

- When R evaluates scope.ex() it uses the x and y from the argument list and not x and y from the global environment. Since there is no z in scope.ex(), R gets z from the global environment.

```
> rm(list = ls())   # Remove all objects
> scope.ex <- function(x, y) {
                  w <- x^2 + y^2 + z^2
                  return(w)
              }
> x <- 2
> y <- 7
> z <- 10
> scope.ex(x=5, y=2)
> w     # w is only accessible within scope.ex()
```

Function apply()

- Recall the function `apply(X, MARGIN, FUN, ...)`
- The `FUN` argument can be any function of the vector specified by `MARGIN`, either an R function or one we write.

```
> A <- matrix(sample(1:100, 12), nrow=3, ncol=4)
> apply(A, 2, var)
> apply(A, 2, function(x)
                    sum((x-mean(x))^2)/(length(x)-1))
> fun1 <- function(x, a) sum(x-a)^2/(length(x)-1)
> apply(A, 2, fun1, a=40)
> fun2 <- function(x, a, b) sum(x-a)^2/b
> apply(A, 2, fun2, a=40, b=10)
```

Function `lapply()`

- The function `lapply` applies a given function to each element of a list and returns the computed values in a list
- Essentially, `lapply` runs a loop to carry out its computation.

```
> lapply(list(a = 2, b = 3), sqrt)
$a
[1] 1.414214

$b
[1] 1.732051
> lapply(matrix(1:6, 2, 3), function(x) x^2)
```

```r
> mylapply <- function(lst, fun) {
                lst <- as.list(lst)
                ans <- vector("list", length(lst))
                names(ans) <- names(lst)
                for(i in 1:length(lst))
                    ans[i] <- list(fun(lst[[i]]))
                ans
              }
> mylapply(list(a=2, b=3), sqrt)

$a
[1] 1.414214

$b
[1] 1.732051
```

Function sapply()

- The `sapply` function behaves just like the `lapply` function, except that it tries to simplify its result into a vector or array
- The result is a numeric vector with named elements.
- The `sapply` function can be used to produce a vectorized version of functions of a single variable.

```
> sapply(list(a=2, b=3), sqrt)

       a        b
1.414214 1.732051
```

```
> sapply(matrix(1:6, 2, 3), function(x) x^2)
[1] 1 4 9 16 25 36
```

Invisible Return Values

- All R functions return a value. It is possible to make the value returned by a function be non-printing by returning it as the value of the invisible function.

```
> no.print <- function(x) invisible(x^2)
> no.print(1:10)
> x <- no.print(1:10)
> x
[1] 1 4 9 16 25 36 49 64 81 100
```

Variable Numbers of Arguments

- R functions can be defined to take a variable number of arguments. The special argument name `...` will match any number of arguments in the call (which are not matched by any other arguments).
- The mean function computes the mean of the values in a single vector. We can easily create an equivalent function which will compute the mean of all the values in all its arguments.

```
> mean.of.all <- function(...) {
                   mean(c(...))
                 }
> mean.of.all(1:3, 1:5)
[1] 2.625
```

More on ...

- Ony one `...` can be used in the argument list for a function.
- The only thing which can be done with `...` inside a function is to pass it as an argument to a function call.
- The following function assembles its arguments into a vector

```
> c2 <- function(...) c(..., ...)
> c2(1, 2, 3)
[1] 1 2 3 1 2 3
> c2(a=1, b=2)
a b a b
1 2 1 2
```

Example

- The following function computes the trimmed mean of the means of several vectors.

```
> trim.means <- function(..., trim=0) {
        x <- list(...)
        n <- length(x)
        r <- numeric(n)
        for (i in 1:n)
            r[i] <- mean(x[[i]], trim=trim)
        mean(r)
    }
> trim.means(c(1,2,3,4,100), c(-10,2,5))
[1] 10.5
> trim.means(c(1,2,3,4,100), c(-10,2,5), trim=1)
[1] 2.5
```

Stopping Computations

- Sometimes a situation arises during a computation where it is necessary to simply give up and abandon the computation.
- There is an R function called stop which makes this easy
- The argument to stop is a character string which explains why the computation is being stopped.

```
> stop.fun <- function(x) {
                 if (x > 10) stop("bad x value")
                 else x
               }
> stop.fun(5)
[1] 5
> stop.fun(12)
Error in stop.fun(12) :  bad x value
```

Advantages of Functions

- Leads to more compact code when a task needs to be performed multiple times.
- Easier to make changes to a function then to make changes to several locations throughout a program.
- A well written function is more trustworthy and transparent, all of the expressions are contained within the function.
- Easier to interact through a function interface than to type multiple lines of code.
- Function computations are performed locally; objects created within a function will not accidentally overwrite global objects.
- Most convenient way to share programs with other users.
- A well written function can be adapted to different situations; solve more then one problem with a single function.

source()

- Once we have finished writing a function, we can save the code to a file and then use the `source()` function to read the contents of the file when the function is needed.
- Using `source()` is similar to loading a package with `library()`. We can use the functions located in the source file without declaring and evaluating the functions during the R session.
- Technically, the code in the file read by `source()` can be any R expressions, not just functions.
- To avoid specifying the path name, set the working directory to the location of the source file.

```
> rm(list = ls())
> source("xxx.R")
```