Lecture 9. Doing Math and Simulations in R

R and Data Visualization

BIG2006, Hanyang University, Fall 2022

Math Functions

- `exp()`: Exponential function, base e
- `log()`: Natural logarithm
- `log10()`: Logarithm base 10
- `sqrt()`: Square root
- `abs()`: Absolute value
- `sin()`, `cos()`, and so on: Trig functions
- `min()` and `max()`: Minimum value and maximum value within a vector
- `which.min()` and `which.max()`: Index of the minimal element and maximal element of a vector

- `pmin()` and `pmax()`: Element-wise minima and maxima of several vectors
- `sum()` and `prod()`: Sum and product of the elements of a vector
- `cumsum()` and `cumprod()`: Cummulative sum and product of the elements of a vector
- `round()`, `floor()`, and `ceiling()`: Round to the closest integer, to the cloest integer below, and to the closest integer above
- `factorial()`: Factorial function

Calculating a Probability

Q. Suppose we have n independent events, and the i^{th} event has the probability p_i of occurring. What is the probability of exactly one of these events occurring?

A. For general n, the probability is calculated as follows:

$$\sum_{i=1}^{n} p_i(1-p_1)\cdots(1-p_{i-1})(1-p_{i+1})\cdots(1-p_n)$$

Here, the i^{th} term inside the sum is the probability that event i occurs and all the others do not occurs.

- Suppose $n = 4$ and four events have the probabilities as follows: (0.3,0.4,0.2,0.1)

```
exactlyone <- function(p){
  notp <- 1 - p
  tot <- 0.0
  for (i in 1:length(p))
    tot <- tot + p[i] * prod(notp[-i])
  return(tot)
}
exactlyone(c(0.3,0.4,0.2,0.1))
```

```
## [1] 0.4404
```

Minima and Maxima

- min() vs pmin() and max() vs pmax()

```
z <- matrix(c(1,5,6,2,3,2),nrow=3)
c(min(z[,1],z[,2]), "||", pmin(z[,1],z[,2]))
```

```
## [1] "1"  "||" "1"  "3"  "2"
```

```
c(max(z[,1],z[,2]), "||", pmax(z[,1],z[,2]))
```

```
## [1] "6"  "||" "2"  "5"  "6"
```

- Function minimization/maximization can be done via `nlm()` and `optim()`.
- Find the smallest value of $f(x) = x^2 - \sin(x)$:

```
nlm(function(x) return(x^2-sin(x)),8) # Newton-Raphson method
```

```
## $minimum
## [1] -0.2324656
##
## $estimate
## [1] 0.4501831
##
## $gradient
## [1] 4.024558e-09
##
## $code
## [1] 1
##
## $iterations
## [1] 5
```

Calculus

- Calculus capabilities, including symbolic differentiation and numerical integration.

```
D(expression(exp(x^2)),"x") # derivative
```

```
## exp(x^2) * (2 * x)
```

```
integrate(function(x) x^2, 0, 1)
```

```
## 0.3333333 with absolute error < 3.7e-15
```

Note: You can find R packages for differential equations (`odesolve`), for interfacing R with Yacas symbolic math system (`ryacas`), and for other calculus operations.

Functions for Statistical Distributions

Prefix the name as follows:

- With `d` for the density or probability mass function (pmf)
- With `p` for the cumulative distribution function (cdf)
- With `q` for quantiles
- With `r` for random number generation

Table 8-1: Common R Statistical Distribution Functions

Distribution	Density/pmf	cdf	Quantiles	Random Numbers
Normal	dnorm()	pnorm()	qnorm()	rnorm()
Chi square	dchisq()	pchisq()	qchisq()	rchisq()
Binomial	dbinom()	pbinom()	qbinom()	rbinom()

- Simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean:

```
mean(rchisq(1000,df=2))
```

```
## [1] 1.884069
```

- Compute the 5^{th} and 95^{th} percentiles of the chi-square distribution with two degrees of freedom:

```
qchisq(c(0.05,0.95),2)
```

```
## [1] 0.1025866 5.9914645
```

Sorting

- `sort()`: ordinary numerical sorting of a vector
- `order()`: indices of the sorted values in the original vector
- `rank()`: rank of each element of a vector

```
x <- c(9,5,4,4)
sort(x)
```

```
## [1] 4 4 5 9
```

```
c(order(x), "||", rank(x))
```

```
## [1] "3"   "4"   "2"   "1"   "||"  "4"   "3"   "1.5" "1.5"
```

```r
y <- data.frame(list(V1=c("math","stat","data"),
                     V2=c(2,5,1)))
r <- order(y$V2)
r
```

```
## [1] 3 1 2
```

```r
z <- y[r,]
z
```

```
##     V1 V2
## 3 data  1
## 1 math  2
## 2 stat  5
```

```r
d <- data.frame(list(city=c("Seoul","Tokyo","New York"),
                     temp=c(28,22,25)))
d[order(d$city),]
```

```
##       city temp
## 3 New York   25
## 1    Seoul   28
## 2    Tokyo   22
```

```r
d[order(d$temp),]
```

```
##       city temp
## 2    Tokyo   22
## 3 New York   25
## 1    Seoul   28
```

Linear Algebra Operations on Vectors and Matrices

- `crossprod()`: inner product (or dot product)

```
crossprod(1:3,c(5,12,13))
```

```
##      [,1]
## [1,]   68
```

Note: The name `crossprod()` is a misnomer, as the function does not compute the vector cross product.

- Matrix multiplication

```
a <- matrix(1:4,nrow=2,ncol=2,byrow=T)
b <- matrix(c(1,0,-1,1),2,2)
a %*% b
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    3    1
```

- `solve()`: solve systems of linear equations

```
a <- matrix(c(1,1,-1,1),2,2)
b <- c(2,4)
solve(a,b) # x_1 - x_2 = 2 and x_1 + x_2 = 4
```

```
## [1] 3 1
```

```
solve(a) # find matrix inverse
```

```
##      [,1] [,2]
## [1,]  0.5  0.5
## [2,] -0.5  0.5
```

Linear algebra functions:

- `t()`: matrix transpose
- `qr()`: QR decomposition
- `chol()`: Cholesky decomposition
- `det()`: Determinant
- `eigen()`: Eigenvalues/eigenvectors
- `diag()`: extracts the diagonal of a square matrix
- `sweep()`: numerical analysis sweep operations

```r
m <- matrix(c(1,7,2,8),2,2)
dm <- diag(m)
dm
```

```
## [1] 1 8
```

```r
diag(dm)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    8
```

```r
diag(3)
```

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

```r
m <- matrix(1:9,3,3,byrow=T)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```r
sweep(m,1,c(1,4,7),"+")
```

```
##      [,1] [,2] [,3]
## [1,]    2    3    4
## [2,]    8    9   10
## [3,]   14   15   16
```

Note: The first two arguments to `sweep()` are like those of `apply()`: array and the margin (1=row). The fourth is a function to be applied, and the third is an argument to that function.

Set Operations

- `union(x, y)`: union of the sets `x` and `y`
- `intersect(x, y)`: intersection of the sets `x` and `y`
- `setdiff(x, y)`: set difference between `x` and `y`, consisting of all elements of `x` that are not in `y`
- `setequal(x, y)`: test for equality between `x` and `y`
- `c %in% y`: membership, testing whether `c` is an element of the set `y`
- `choose(n, k)`: number of possible subsets of size `k` chosen from a set of size `n`

```r
x <- c(1,2,5)
y <- c(5,1,8,9)
intersect(x,y)
```

```
## [1] 1 5
```

```r
setequal(x,y)
```

```
## [1] FALSE
```

```r
2 %in% x
```

```
## [1] TRUE
```

- Coding the symmetric difference between two sets, i.e., all the elements belonging to exactly one of the two operand sets:

```
symdiff <- function(x,y){
  sdfxy <- setdiff(x,y)
  sdfyx <- setdiff(y,x)
  return(union(sdfxy,sdfyx))
}
x
```

```
## [1] 1 2 5
```

```
y
```

```
## [1] 5 1 8 9
```

```
symdiff(x,y)
```

```
## [1] 2 8 9
```

- combn(): generate combinations

```
c32 <- combn(1:3,2) # find the subsets of {1,2,3} of size 2
c32
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    2
## [2,]    2    3    3
```

```
class(c32)
```

```
## [1] "matrix" "array"
```

```
combn(1:3,2,sum) # find the sum of the members in each subset
```

```
## [1] 3 4 5
```

Simulation Programming in R

Built-In R Random Variate Generators

- Find the probability of getting at least four heads out of five tosses of a coin.

```
x <- rbinom(100000,5,0.5)
mean(x >=4)
```

```
## [1] 0.18532
```

Note: The `TRUE` and `FALSE` values in `x` above are treated as 1s and 0s by `mean()`, giving us our estimated probability.

Note: Other functions include `rnorm`() for the normal distribution, `rexp`() for the exponential, `runif`() for the uniform, `rgamma`() for the gamma, `rpois`() for the Poisson, and so on.

- Find $E[\max(X,Y)]$ of the maximum of independent $N(0,1)$ random variables X and Y.

```
sum <- 0
nreps <- 100000
for (i in 1:nreps){
  xy <- rnorm(2) # generate 2 N(0,1)s
  sum <- sum + max(xy)
}
print(sum/nreps)
```

```
## [1] 0.5653839
```

Note: We generated 100,000 pairs, found the maximum for each, and averaged those maxima to obtain our estimated expected value.

Obtaining the Same Random Stream in Repeated Runs

- R random-number generators use 32-bit integers for seed values (maybe 64-bit).
- Other than round-off error, the same initial seed should generate the same stream of numbers.
- Use `set.seed()` if you want the same stream each time.

```
set.seed(0922) # or your favorite number as an argument
```

Reference

- Matloff, N. The Art of R Programming: A Tour of Statistical Software Design. No Starch Press. Chapter 8.