mathematics and simulations » in R

The following content applies the constructs and theory of linear algebra and multivariate calculus within R.

Basic mathematical functions in R

function	description
exp()	Exponential function, base e
log()	Natural logarithm
log10()	Logarithm base 10
sqrt()	Square root
abs()	Absolute value
sin(), cos(), etc.	Trigonometric functions
min(), max()	Minimum value and maximum value within a vector
which.min(), which.max()	Index of the minimal element and maximal element of a vector
pmin(), pmax()	Element-wise minima and maxima of several vectors
sum(), prod()	Sum and product of the elements of a vector
cumsum(), cumprod()	Cumulative sum and product of the elements of a vector
round(), floor(), ceiling()	Round to the closest integer, to the closest integer below, and to the closest integer above
factorial()	Factorial function

8.1.1 calculating a probability ◎ in R

Using the **prod( )** function, probabilities can be computed in R. Given the following example:

```
> # there are n independent events; the ith event has the probability of
> # pi of occurring. What is the probability of exactly one event occurring?
```

Assuming the first $\boldsymbol{n} = \mathbf{3}$ and the events are named **A**, **B**, and **C**. The computation as follows:

$$
\begin{array}{lcl}
P(exactly\ one\ event\ occurs) = & \rightarrow & \boldsymbol{P(i)} \quad = \\
P(A\ and\ not\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{P(A \cap B' \cap C') +} \\
P(not\ A\ and\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{P((A \cap B)'and\ C') +} \\
P(not\ A\ and\ not\ B\ and\ C) & \rightarrow & \boldsymbol{P(A' \cap (B \cap C)')}
\end{array}
$$

The computation can also be represented in pseudocode as follows:

$$
\begin{array}{lcl}
P(exactly\ one\ event\ occurs) = & \rightarrow & \boldsymbol{p(i)} \quad = \\
P(A\ and\ not\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{p_A(1 - p_B)(1 - p_C) +} \\
P(not\ A\ and\ B\ and\ not\ C) + & \rightarrow & \boldsymbol{(1 - p_A p_B)(1 - p_C) +} \\
P(not\ A\ and\ not\ B\ and\ C) & \rightarrow & \boldsymbol{(1 - p_A)(1 - p_B p_C)}
\end{array}
$$

For the general observation n, the probability of occurrence is calculated as follows:

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

*The $\boldsymbol{i^{th}}$ term inside the summation is the probability that event i occurs and all others ***do not***.

The R code to compute the preceding mathematics (with probabilities $\boldsymbol{p_i}$ stored in vector **p**) is as follows:

```
> exactlyone  #the notp <- creates a vector of “not occur” probabilities 1-pj by recycling
function(p) {
    notp <- 1 - p
    tot <- 0.0
    for(i in 1:length(p))
        tot <- tot + p[i] + prod(notp[-i])
    return(tot)
}
```

The expression **prod(notp[-i])** computes the produce of all elements of **notp**—sans the $\boldsymbol{i^{th}}$, as needed.

cumulative sums · products · minima · maxima in R

the **cumsum( )** and **cumprod( )** return cumulative sums and products of their applied arguments.

```
> x <- c(12,5,13)
> cumsum(x)           #computes the cumulative sum of ordered values in vector x
[1] 12 17 30

> cumprod(x)          #computes the cumulative product of ordered values in vector x
[1]  12  60 780
```

The is a notable difference between the **min( )** and **pmin( )** functions. Function **min( )** combines all arguments into a single vector, returning the ***minimum*** value. Function **pmin( )**, if applied to two or more vectors, returns a vector of the ***pair-wise minima***.

```
> z <- matrix(c(1,5,6,2,3,2), ncol = 2)
> z
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2

> min(z[,1],z[,2])              #returns the smallest value of (1,5,6,2,3,2)
[1] 1

> pmin(z[,1],z[,2])             #returns the smaller of (1,2); of (5,3); and of (6,2)
[1] 1 3 2
```

Additionally, more than two arguments can be used in the **pmin( )** function:

```
> pmin(z[1,],z[2,],z[3,])              #returns the minima of (1,5,6); and of (2,3,2)
[1] 1 2
```

The **max( )** and **pmax( )** are exhibit analogous behavior to those of the **min( )** and **pmin( )** functions:

Functions ***minimization/maximization*** can be accomplished through the **nlm( )** and **optim( )** functions. The following example identifies the smallest value of $f(x) = x^2 - \sin(x)$:

```
> nlm(function(x) return(x^2 - sin(x)), 8)
$minimum
[1] -0.2324656

$estimate
[1] 0.4501831

$gradient
[1] 4.024558e-09

$code
[1] 1

$iterations
[1] 5
```

The **minimum value** in the above illustration was identified as approximately -0.23, occurring at $x = 0.45$.

The above technique derives from a Newton-Raphson method of numerical analysis for approximating roots; the functions runs through **5 iterations** in the above example. The second argument in the **nlm( )** function specifies the initial estimation (**8**); Note, that the example above employs **8** arbitrarily. More discipline should be applied in practice to ensure convergence.

calculus $\int_0^1 f(x)$ in R

R has many capable calculus applications, including symbolic differentiation and numerical integration.

$$\frac{d}{dx}e^{x^2} = 2xe^{x^2} \qquad \text{and} \qquad \int_0^1 x^2 dx \approx 0.3333333$$

```
> D(expression(exp(x^2)), "x")
exp(x^2) * (2 * x)

> integrate(function(x) x^2, 0, 1)
0.3333333 with absolute error < 3.7e-15
```

There are many available calculus packages in R to leverage (a small few listed below):

Example calculus packages in R

function	description
odesolve	differential equations
ryacas	interfacing R with the Yacas symbolic mathematics system
Deriv	symbolic differentiation
numDeriv	the standard for numerical differentiation in R
pracma	functions for computing numerical derivatives
gaussquad	a collection of functions to perform Gaussian quadrature

statistical distribution functions in R

To no surprise, R has a core magnitude of statistical distributions covered in the CRAN.

The distribution is typically prefixed with the data scope:

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

with what follows, after the prefix, indicating the distribution applied;

Common R statistical distribution function examples

distribution	density/pmf	cdf	quantiles	random numbers
normal	dnorm()	pnorm	qnorm()	rnorm()
chi square	dchisq()	pchisq()	qchisq()	rchisq()
binomial	dbinom()	pbinom()	qbinom()	rbinom()

The following simulates 1,000 chi-square variates with 2 degrees of freedom; finding their mean:

```
> mean(rchisq(1000, df = 2))       #"r" specifies the generation of random numbers
[1] 1.994469
```

The above initial argument specifies the 1000 random numbers to be generated in the simulation. Additionally, distribution functions in R also have arguments specific to the distribution families. In the above example, the **df =** argument refers to the degrees of freedom belonging to the **chi-square** family. The following example computes the 95th percentile of the chi-square distribution with 2 degrees of freedom:

```
> qchisq(0.95,2)                         #returns 95% quantile of chi-square distribut
[1] 5.991465

> qchisq(c(0.5,0.95), df = 2)     #returns 50% and 95% quantile of chi-square distribut
[1] 1.386294 5.991465
```

The 1st argument of distribution functions is a vector to evaluate the **d**, **p**, **q**, at multiple points (seen above).

sorting in R

Ordinary numerical sorting of a vector is available through the **sort( )** function:

```
> x <- c(13,5,12,5)
> sort(x)              #returns a numerically sorted vector x
[1]  5  5 12 13

> x                    #vector x remains in the original assigned order
[1] 13  5 12  5
```

The **order( )** function will provide the indices of the sorted values from the original vector:

```
> order(x)
[1] 2 4 3 1
```

The **order( )** function indicates that **x[2]** is the smallest value in vector **x**; **x[1]** being the largest value in **x**.

The **order( )** function can be applied along with **indexing** to sort ***dataframes***:

```
> y
    V1 V2
1  def  2
2   ab  5
3 zzzz  1

> r <- order(y$V2)                #return the indices of column V2 in dataframe y
> r
[1] 3 1 2

> z <- y[r,]              #assign the sorted index of column V2 from dataframe y to z
> z
    V1 V2
3 zzzz  1
1  def  2
2   ab  5
```

Looking at the **order(y$V2)** call, the resulting **3** identifies **x[3,2]** as the smallest number in **x[,2]**; the **1** identifies **x[1,2]** as the middle number in **x[,2]**; the **2** identifies **x[2,2]** as the largest number in **x[,2]**. The latter call assigned an **index** to be used as an argument in the assignment of **z** for a sorted dataframe.

The **order( )** function can also be applied to ***character variables***:

```
> d                                          #dataframe d
   kids ages
1  Jack   12
2  Jill   10
3 Billy   13

> d[order(d$kids),] d                        #sort dataframe d by kids' names
   kids ages
3 Billy   13
1  Jack   12
2  Jill   10

> d[order(d$ages),]                          #sort dataframe d by kids' ages
   kids ages
2  Jill   10
1  Jack   12
3 Billy   13
```

A related function to sorting in R is the **rank( )** function; reporting the rank of each element in a vector:

```
> x
[1] 13  5 12  5

> rank(x)         #element 13 is ranked 4 (smallest); element 5 appears twice, ranked 1.5
[1] 4.0 1.5 3.0 1.5
```

linear algebra operations on vectors and matrices in R

Multiplying vectors by scalars:

```
> y
[1]  1  3  4 10

> 2*y                              #element wise multiplication of 2 by vector y
[1]  2  6  8 20
```

Computing the **inner-product** (dot product) of two vectors with **crossprod( )**:

```
> crossprod(1:3, c(5,12,13))        #does not calculate actual vector cross product
     [,1]
[1,]   68
```

The compute → $\mathbf{1 * 5 + 2 * 12 + 3 * 13 = 68}$; note **crossprod( )** *does not* calculate the vector cross product.

Mathematical matrix multiplication is applied through the **%*%** operator, opposed to the ***** operator:

Matrix product notation → $\begin{pmatrix} \mathbf{1} & \mathbf{3} \\ \mathbf{2} & \mathbf{4} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{-1} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{3} \end{pmatrix}$ → The R code as follows:

```
> a <- matrix(1:4, ncol = 2 ,byrow = TRUE)
> a
     [,1] [,2]
[1,]    1    2
[2,]    3    4

> b <- matrix(c(1,0,-1,1), ncol = 2)
> b
     [,1] [,2]
[1,]    1   -1
[2,]    0    1

> a %*% b
     [,1] [,2]
[1,]    1    1
[2,]    3    1
```

The **solve( )** function solves systems of ***linear equations*** and also provide ***matrix inverses***.

Linear System → $\begin{matrix} \boldsymbol{x_1 + x_2 = 2} \\ \boldsymbol{-x_1 + x_2 = 4} \end{matrix}$ → Matrix Notation → $\begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{-1} & \mathbf{1} \end{pmatrix} \begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \end{pmatrix} = \begin{pmatrix} \mathbf{2} \\ \mathbf{4} \end{pmatrix}$ → code below:

```
> a <- matrix(c(1,-1,1,1), nrow = 2)
> b <- c(2,4)
> solve(a,b)
[1] -1  3

> solve(a)                     #the missing 2nd argument causes the inverse to be compute
     [,1] [,2]
[1,]  0.5 -0.5
[2,]  0.5  0.5
```

Examples of available Linear Algebra functions in R (a few provided below):

Example R linear algebraic functions			
t()	matrix transpose	diag()	extracts the diagonal vector of a square matrix (useful for obtaining variances from a covariance matrix and for constructing a diagonal matrix)
qr()	QR decomposition		
chol()	Cholesky decomposition		
det()	determinant		
eigen()	eigenvalues/eigenvectors	sweep()	numerical analysis sweep operations

The following example notes the flexibility of the **diag( )** function:

```
> m
     [,1] [,2]
[1,]    1    2
[2,]    7    8

> dm  <- diag(m)          #takes the diagonal axis of matrix m and assigns to vector d
> dm
[1] 1 8

> diag(dm)                #creates a matrix using vector dm as the diagonal axis
     [,1] [,2]
[1,]    1    0
[2,]    0    8

> diag(3)                 #creates an identity matrix of size 3x3
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

If the argument is a matrix, **diag( )** returns a vector; If the argument is a vector, **diag( )** returns a matrix. Additionally, if the argument is a scalar, **diag( )** returns the ***identity matrix*** of the specified size.

The **sweep( )** function makes more complex operations available in R. the following illustration takes a 3x3 matrix $\mathbb{R}^{3\times 3}$ and adds **1** to row 1; **4** to row 2; and **7** to row 3.

```
> m <- matrix(1:9, nrow = 3, byrow = TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> sweep(m, 1, c(1,4,7), "+")
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    8    9   10
[3,]   14   15   16
```

The first two arguments of **sweep( )** are similar to **apply( )**: the ***array***; and the ***margin*** (1 for rows, example above). The 4th argument is the ***function*** to apply, with the 3rd argument being the function ***argument***.

set operations ∩∪ in R

Example set operations in R	
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 not in y
Setequal(x,y)	Test for equality between x and y
X %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

The following illustrates the use of the above ***set operations*** in R:

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

> union(x,y)
[1] 1 2 5 8 9

> intersect(x,y)
[1] 1 5

> setdiff(x,y)
[1] 2

> setdiff(y,x)
[1] 8 9
```

```
> setequal(x,y)
[1] FALSE

> setequal(x, c(1,2,5))
[1] TRUE

> 2 %in% x
[1] TRUE

> 2 %in% y
[1] FALSE

> choose(5,2)
[1] 10
```

Considering the symmetric difference between two sets—all elements belong to exactly one of the two operand sets. Because the symmetric difference between sets **x** and **y** consist exactly of those elements in **x** but not in **y** (and vice versa), the code consists of easy calls to **setdiff( )** and **union( )**:

```
> symdiff
function(a,b) {
    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
```

Below offers additional illustration of a binary operand for determining whether one set **u** is a subset of **v**:

```
> "%subsetof%" <- function(
+     return(setequal(inter
+ }
> c(3,8) %subsetof% 1:10
[1] TRUE

> c(3,8) %subsetof% 5:10
[1] FALSE
```

Below applies the **combn( )** function to generate combinations, resulting as follows:

```
> c32 <- combn(1:3,2)
> c32
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3

> class(c32)
[1] "matrix"
```

A function can also be called within **combn( )**:

```
> combn(1:3,2,sum)
[1] 3 4 5
```

simulation programming in R

A common use for R programming is that of running simulations.

The **rbinom( )** function random binomial (Bernoulli) variates. Assuming the probability of correctly predicting ≥ (at least) 4 heads out of 5 coin tosses:

```
> x <- rbinom(100000,5,0.5)      #100,000 random variates, 5 trials, 50% success rate
> mean(x >= 4)                   #resulting in a Boolean vector of equal length to x
[1] 0.1869
```

Of many available simulation functions, some of R's core functions are listed as follows:

Example simulation functions in R	
rnorm()	Normal distribution simulations
rexp()	Exponential simulations
runif()	Uniform simulations
rgamma()	Gamma simulations
rpois()	Poisson simulations

Below finds $E[\max(X,Y)]$; expected value of the maximum of independent N(0,1) random variables X and T:

```
> sum <- 0
> nreps <- 100000
> for (i in 1:nreps) {              #generates 100,000 pairs
+     xy <- rnorm(2)                #generates 2 N(0,1) values
+     sum <- sum + max(xy)          #adds the maximum of each averaged value
+ }
> print(sum/nreps)
[1] 0.5647364
```

The above code uses an **explicit loop** for the convenience of clarity, the above can be achieved more efficiently with the sacrifice of some computational cost and clarity; overall more compact coding:

```
> emax
function(nreps) {
    x <-  rnorm(2*nreps)                                #2x nreps value
    maxxy <- pmax(x[1:nreps], x[(nreps+1):(2*nreps)])   #fist simulates X, second Y
    return(mean(maxxy))                                 #pmax computes pair-wise maxi
}
> emax(nreps)
[1] 0.566471
```

R-documentation states that all random-number generators use 32-bit integers for seed values. R will generate a different stream of random numbers for each run; the stream can be set with **set.seed( )**.

```
set.seed(8888)        #seed is set to 8888, but can be any number)
```