Math functions in R Programming

The math function denotes the mathematical or numeric calculations with some built-in functions. The set of instructions for particular functions is predefined in the R program, the user just needs to call the function for accomplishing their task of execution. The following table highlights some of the numeric or mathematical built-in functions in R.

Serial No	Built-in function	Description	Example
1	**abs(x)**	It returns the absolute value of input x	x<- -2 print(abs(x)) Output [1] 2
2	**sqrt(x)**	It returns the square root of input x	x<- 2 print(sqrt(x)) Output [1] 1.414214
3	**ceiling(x)**	It returns the smallest integer which is larger than or equal to x.	x<- 2.8 print(ceiling(x)) Output [1] 3
4	**floor(x)**	It returns the largest integer, which is smaller than or equal to x.	x<- 2.8 print(floor(x)) Output [1] 2
5	**trunc(x)**	It returns the truncate value of input x.	x<- c(2.2,6.56,10.11) print(trunc(x)) Output [1] 2 6 10

6	**round(x, digits=n)**	It returns round value of input x.	x=2.456 print(round(x,digits=2)) x=2.4568 print(round(x,digits=3)) Output [1] 2.46 [1] 2.457
7	**cos(x), sin(x), tan(x)**	It returns cos(x), sin(x) , tan(x) value of input x	x<- 2 print(cos(x)) print(sin(x)) print(tan(x)) Output [1] -0.4161468 [1] 0.9092974 [1] -2.18504
8	**log(x)**	It returns natural logarithm of input x	x<- 2 print(log(x)) Output [1] 0.6931472
9	**log10(x)**	It returns common logarithm of input x	x<- 2 print(log10(x))

			Output [1] 0.30103
10	**exp(x)**	It returns exponent	x<- 2 print(exp(x)) Output [1] 7.389056

Simulation Using R Programming

Simulation is a powerful technique in statistics and data analysis, used to model complex systems, understand random processes, and predict outcomes. In R, various packages and functions facilitate simulation studies.

Introduction to Simulation in R

Simulating scenarios is a powerful tool for making informed decisions and exploring potential outcomes without the need for real-world experimentation. This article delves into the world of simulation using R Programming Language versatile programming language widely used for statistical computing and graphics. We'll equip you with the knowledge and code examples to craft effective simulations in R, empowering you to:

- **Predict the Unpredictable:** Explore "what-if" scenarios by simulating various conditions within your system or process.
- **Test Hypotheses with Confidence:** Analyze the behavior of your system under different circumstances to validate or challenge your assumptions.
- **Estimate Parameters with Precision:** Evaluate the impact of changing variables on outcomes, allowing for more accurate parameter estimation.
- **Forecast Trends for Informed Decisions:** Leverage simulated data to predict future behavior and make data-driven choices for your system or process.
-

Types of Simulations

This article will walk you through the basics of simulation in R, covering different types of simulations and practical examples.

1. **Monte Carlo Simulation**: Uses random sampling to compute results and is commonly used for numerical integration and risk analysis.
2. **Discrete Event Simulation**: Models the operation of systems as a sequence of events in time.
3. **Agent-Based Simulation**: Simulates the actions and interactions of autonomous agents to assess their effects on the system.

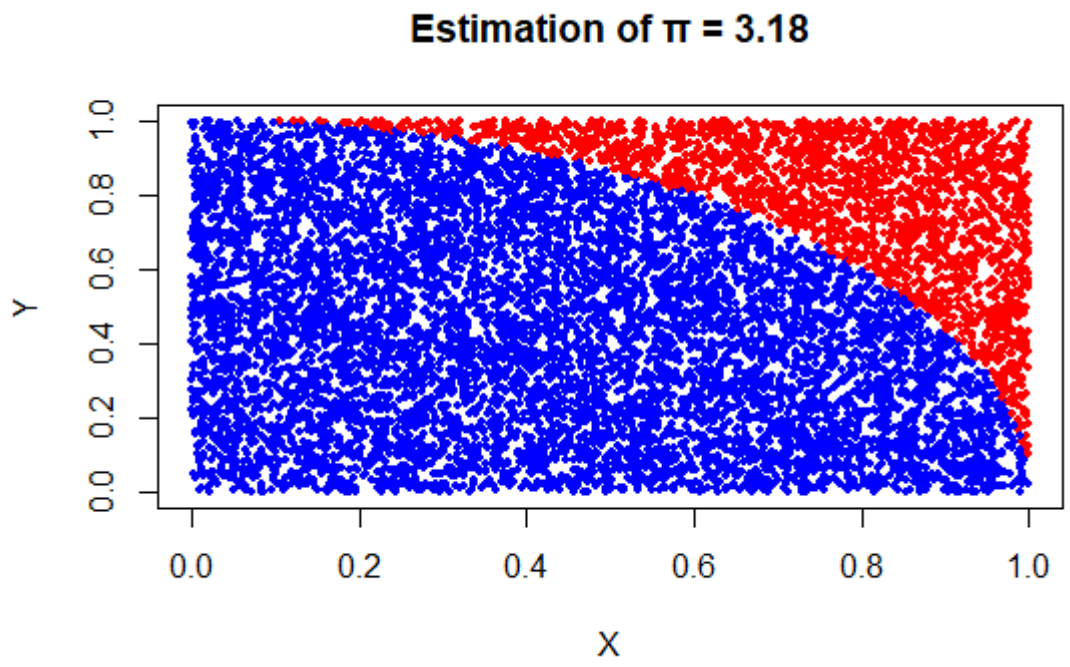
Let's start with a simple Monte Carlo simulation to estimate the value of π.

Estimating π Using Monte Carlo Simulation

The idea is to randomly generate points in a unit square and count how many fall inside the unit circle. The ratio of points inside the circle to the total number of points approximates π/4.

```r
# Number of points to generate
n <- 10000

# Generate random points
set.seed(123)
x <- runif(n)
y <- runif(n)

# Calculate distance from (0,0) and check if inside the unit circle
inside <- x^2 + y^2 <= 1

# Estimate π
pi_estimate <- (sum(inside) / n) * 4
pi_estimate

# Plot the points
plot(x, y, col = ifelse(inside, 'blue', 'red'), pch = 19, cex = 0.5,
     main = paste("Estimation of π =", round(pi_estimate, 4)),
     xlab = "X", ylab = "Y")
```

Output:

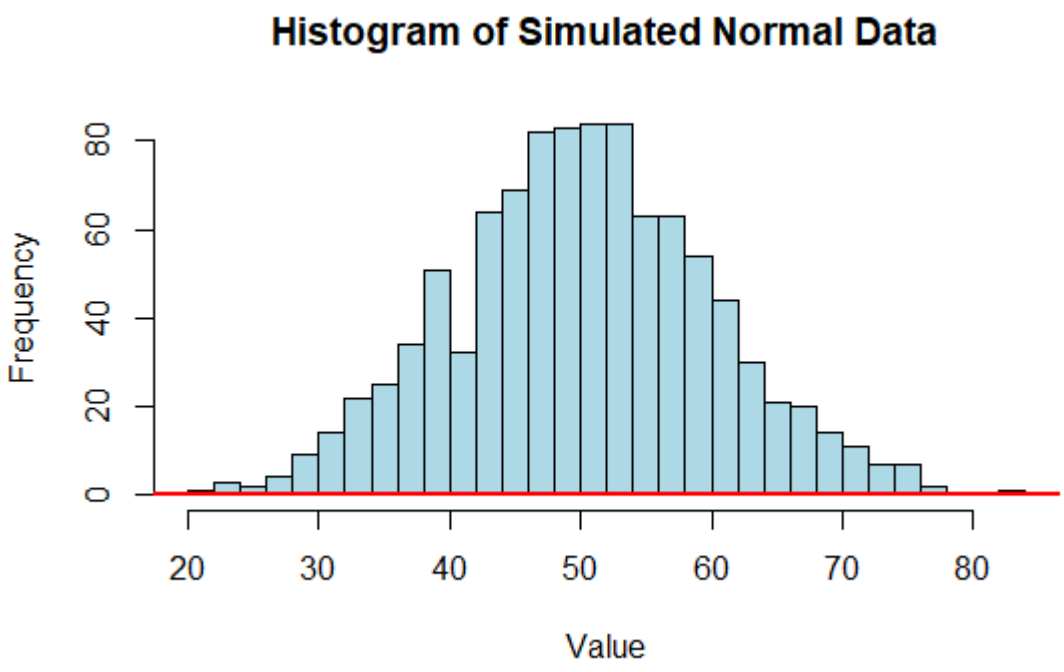


- **runif(n):** Generates n random numbers uniformly distributed between 0 and 1.
- **inside:** Logical vector indicating whether each point lies inside the unit circle.
- **sum(inside) / n * 4:** Estimates π using the ratio of points inside the circle to total points.

Simulating a Normal Distribution

Simulating data from a normal distribution is straightforward with R's rnorm() function. Let's simulate 1000 data points from a normal distribution with a mean of 50 and a standard deviation of 10.

```
# Parameters
mean <- 50
sd <- 10
n <- 1000

# Simulate data
set.seed(123)
data <- rnorm(n, mean = mean, sd = sd)

# Plot the histogram
hist(data, breaks = 30, col = "lightblue", main = "Histogram of Simulated Normal Data",
     xlab = "Value", ylab = "Frequency")

# Add a density curve
lines(density(data), col = "red", lwd = 2)
```



- **rnorm(n, mean, sd):** Generates n random numbers from a normal distribution with specified mean and sd.
- **hist():** Plots a histogram of the simulated data.
- **lines(density(data)):** Adds a kernel density estimate to the histogram.

Discrete Event Simulation

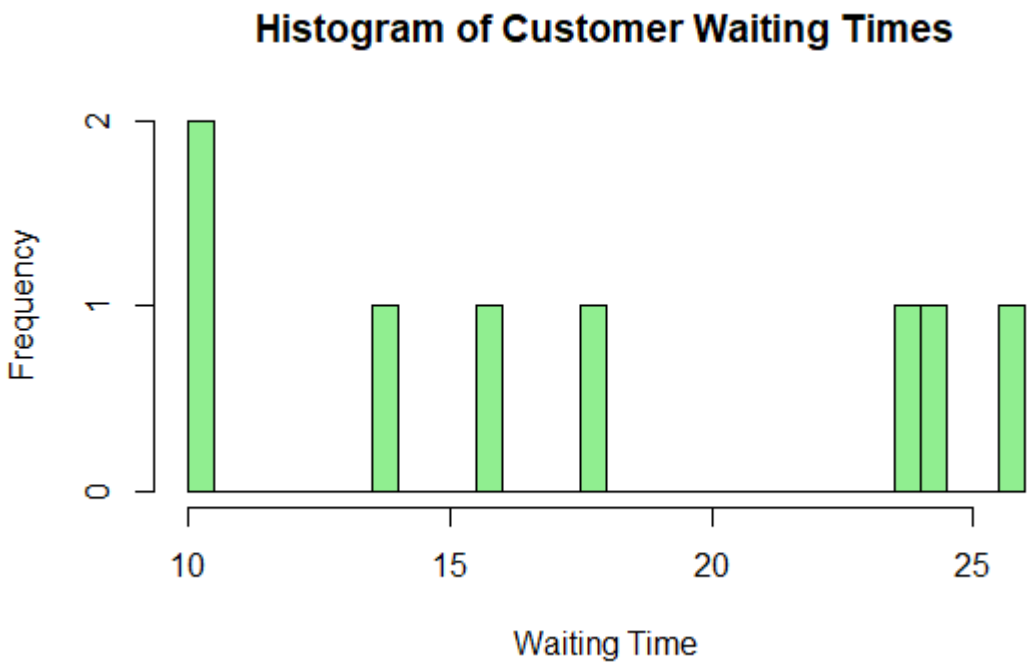
Let's simulate the operation of a simple queue system using the simmer package.

```
# Install and load simmer package
```

```
install.packages("simmer")
library(simmer)

# Define a simple queueing system
env <- simmer("queueing_system")

# Define arrival and service processes
arrival <- trajectory("arrival") %>%
  seize("server", 1) %>%
  timeout(function() rexp(1, 1/10)) %>%
  release("server", 1)

# Add resources and arrivals to the environment
env %>%
  add_resource("server", 1) %>%
  add_generator("customer", arrival, function() rexp(1, 1/5))

# Run the simulation for a specified period
env %>%
  run(until = 100)

# Extract and plot results
arrivals <- get_mon_arrivals(env)
hist(arrivals$end_time - arrivals$start_time, breaks = 30, col = "lightgreen",
     main = "Histogram of Customer Waiting Times",
     xlab = "Waiting Time", ylab = "Frequency")
```

Output:

```
simmer environment: queueing_system | now: 100 | next: 100.308581297463
{ Monitor: in memory }
{ Resource: server | monitored: TRUE | server status: 1(1) | queue status: 7(Inf) }
{ Source: customer | monitored: 1 | n_generated: 17 }
```



- **simmer("queueing_system"):** Creates a new simulation environment.
- **trajectory():** Defines the sequence of operations for arriving customers.
- **seize(), timeout(), release():** Define the customer actions (seizing a server, spending time being served, and releasing the server).
- **add_resource(), add_generator():** Add resources (servers) and customer arrival processes to the environment.
- **run(until = 100):** Runs the simulation for 100 time units.
- **get_mon_arrivals():** Extracts arrival data for analysis.

Probabilities using R

Probability theory is a fundamental concept in mathematics and statistics that plays a crucial role in various fields such as finance, engineering, medicine, and more. Understanding probabilities allows us to make informed decisions in uncertain situations. In this comprehensive guide, we'll delve into the basics of probabilities using R Programming Language.

Basic Concepts of Probability in R

Probability in R is the measure of the likelihood that an event will occur. The probability of an event A, denoted as P(A), lies between 0 and 1, where 0 indicates impossibility and 1 indicates certainty. Some key concepts include:

- **Sample Space (S):** The set of all possible outcomes of a random experiment.
- **Event:** Any subset of the sample space.
- **Probability of an Event:** The likelihood of occurrence of an event, calculated as the ratio of favorable outcomes to the total number of outcomes.

Calculating Probabilities in R

R offers various functions and packages for calculating Probability in R and performing statistical analyses. Some commonly used functions include:

- dbinom(): Computes the probability mass function (PMF) for the binomial distribution.
- pnorm(): Calculates the cumulative distribution function (CDF) for the normal distribution.
- dpois(): Computes the PMF for the Poisson distribution.
- punif(): Calculates the CDF for the uniform distribution.

Here is the basic example of calculating Probability in R:

```
# Define the sample space
sample_space <- c(1, 2, 3, 4, 5, 6)

# Define an event, for example, rolling an even number
event <- c(2, 4, 6)

# Calculate the probability of the event
probability <- length(event) / length(sample_space)
print(probability)
```

Output:

```
[1] 0.5
```

Probability Distributions in R

R provides extensive support for probability distributions, which are mathematical functions that describe the likelihood of different outcomes in a random experiment. Common probability distributions include:

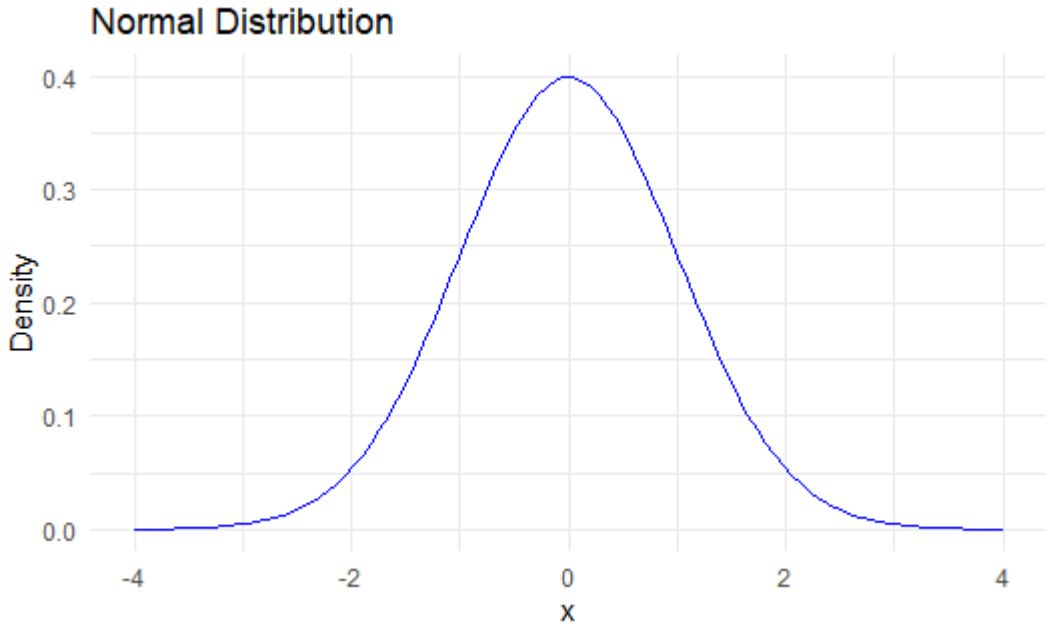
- **Uniform Distribution:** All outcomes are equally likely.
- **Normal Distribution:** Symmetric bell-shaped curve, characterized by mean (μ) and standard deviation (σ).
- **Binomial Distribution:** Describes the number of successes in a fixed number of independent Bernoulli trials.
- **Poisson Distribution:** Models the number of events occurring in a fixed interval of time or space.

Let's visualize the normal distribution with a mean of 0 and standard deviation of 1.

```
library(ggplot2)

# Generate a sequence of x values
x <- seq(-4, 4, length.out = 100)
# Calculate the corresponding densities for normal distribution
y <- dnorm(x, mean = 0, sd = 1)

# Create a data frame
df <- data.frame(x, y)

# Plot the normal distribution
ggplot(df, aes(x = x, y = y)) +
  geom_line(color = "blue") +
  labs(title = "Normal Distribution", x = "x", y = "Density") +
```

```
  theme_minimal()
```

Output:



Normal Distribution

Simulating Probabilistic Experiments in R

Simulation is a powerful tool for understanding probabilities through empirical experiments. R facilitates simulation by allowing the generation of random numbers from different probability distributions. Key functions for simulation include:

- runif(): Generates random numbers from a uniform distribution.
- rnorm(): Generates random numbers from a normal distribution.
- rbinom(): Generates random numbers from a binomial distribution.
- rpois(): Generates random numbers from a Poisson distribution.

```
# Simulating coin flips with a binomial distribution
num_flips <- 1000
num_heads <- sum(rbinom(num_flips, size = 1, prob = 0.5))
probability_heads <- num_heads / num_flips
print(probability_heads)
```

Output:

```
[1] 0.494
```

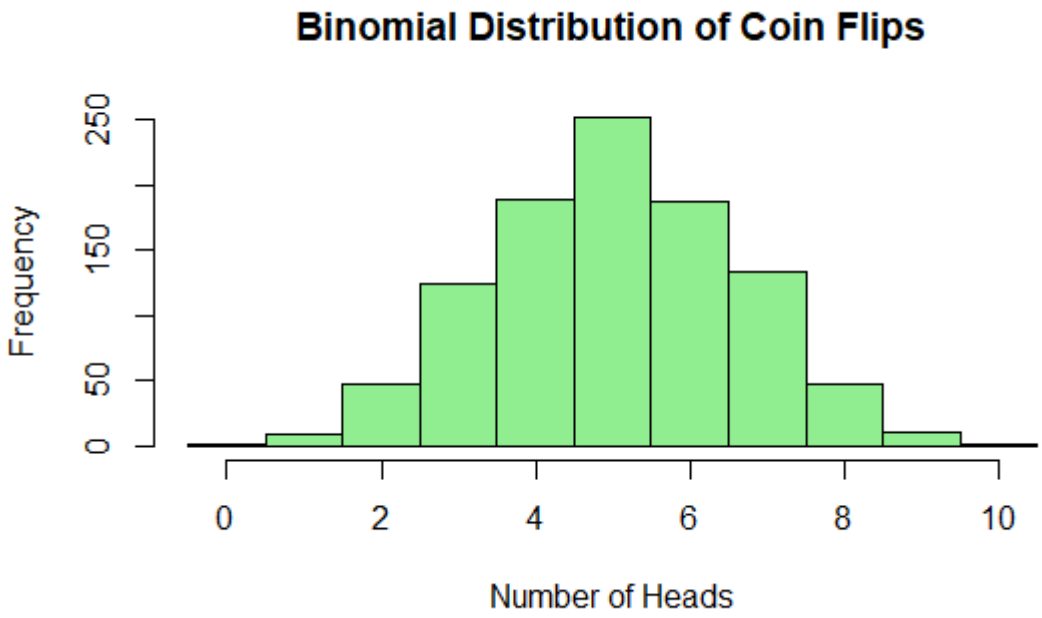
Visualizing Probabilities in R

Visualization is essential for gaining insights from Probability in R and it offers numerous packages such as ggplot2, lattice, and base graphics for creating visualizations. Common plots include histograms, density plots, boxplots, and scatter plots, which help in understanding the shape and characteristics of probability distributions.

```
# Visualizing the binomial distribution of coin flips
flips <- rbinom(1000, size = 10, prob = 0.5)
hist(flips, breaks = seq(-0.5, 10.5, by = 1), col = "lightgreen",
```

```
    main = "Binomial Distribution of Coin Flips", xlab = "Number of Heads",
    ylab = "Frequency")
```

Output:



Cumulative Sums and Products

As mentioned, the functions cumsum() and cumprod() return cumulative sums and products

x <- c(12,5,13)

cumsum(x)

[1] 12 17 30

cumprod(x)

[1] 12 60 780

In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30. The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

Minima and Maxima

There is quite a difference between min() and pmin(). The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if

pmin() is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.

Here's an example:

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

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

```
pmin(z[,1],z[,2])
```

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

In the first case, min() computed the smallest value in (1,5,6,2,3,2). But the call to pmin() computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

You can use more than two arguments in pmin(), like this:

```
pmin(z[1,],z[2,],z[3,])
```

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

The 1 in the output is the minimum of 1, 5, and 6, with a similar computation leading to the 2. The max() and pmax() functions act analogously to min() and pmin().

Sorting

Ordinary numerical sorting of a vector can be done with the sort() function, as in this example:

```
x <- c(13,5,12,5)
sort(x)
```

```
## [1]  5  5 12 13
```

```
x
```

```
## [1] 13  5 12  5
```

Note that x itself did not change, in keeping with R's functional language philosophy. If you want the indices of the sorted values in the original vector, use the order() function. Here's an example:

```
order(x)
```

```
## [1] 2 4 3 1
```

This means that x[2] is the smallest value in x, x[4] is the second smallest, x[3] is the third smallest, and so on.

You can use order(), together with indexing, to sort data frames, like this:

```
y<-data.frame(V1=c("def","ab","zzzz"),
```

```
            V2=c(2,5,1))
r <- order(y$V2)
r
```

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

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

	V1
	<chr>
3	zzzz
1	def
2	ab

3 rows

What happened here? We called order() on the second column of y, yielding a vector r, telling us where numbers should go if we want to sort them. The 3 in this vector tells us that x[3,2] is the smallest number in x[,2]; the 1 tells us that x[1,2] is the second smallest; and the 2 tells us that x[2,2] is the third smallest. We then use indexing to produce the frame sorted by column 2, storing it in z. You can use order() to sort according to character variables as well as numeric ones, as follows:

```
d<-data.frame(kids=c("Jack", "Jill","Billy"),
            ages=c(12,10,13))
d[order(d$kids),]
```

	kids
	<chr>
3	Billy
1	Jack
2	Jill

3 rows

```
d[order(d$ages),]
```

	kids
	<chr>
2	Jill
1	Jack
3	Billy

3 rows

A related function is rank(), which reports the rank of each element of a vector.

```
x <- c(13,5,12,5)
rank(x)
```

```
## [1] 4.0 1.5 3.0 1.5
```

This says that 13 had rank 4 in x; that is, it is the fourth smallest. The value 5 appears twice in x, with those two being the first and second smallest, so the rank 1.5 is assigned to both. Optionally, other methods of handling ties can be specified.

Linear Algebra Operations on Vectors and Matrices

Multiplying a vector by a scalar works directly, as you saw earlier. Here's another example:

```
y<-c(1,3,4,10)
2*y
```

```
## [1]  2  6  8 20
```

If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:

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

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

The function computed $1 \cdot 5 + 2 \cdot 12 + 3 \cdot 13 = 68$. Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product.

For matrix multiplication in the mathematical sense, the operator to use is %%, *not* . For instance, here we compute the matrix product:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 1 \end{pmatrix}$$

Here's the code:

```
a <-matrix(c(1,3,2,4), nrow=2)
b<-matrix(c(1,0,-1,1),nrow=2)
c<-matrix(c(1,3,1,1), nrow=2)
a %*% b
```

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

The function solve() will solve systems of linear equations and even find matrix inverses. For example, let's solve this system:

$$x1+x2=2x1+x2=2$$

$$-x1+x2=4-x1+x2=4$$

Its matrix form is as follows:

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Here's the code:

```
a <- matrix(c(1,1,-1,1),nrow=2,ncol=2)
b <- c(2,4)
solve(a,b)
```

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

```
solve(a)
```

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

In that second call to solve(), the lack of a second argument signifies that we simply wish to compute the inverse of the matrix. Here are a few other 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 (useful for obtaining variances from a covariance matrix and for constructing a diagonal matrix). - sweep(): Numerical analysis sweep operations

Note the versatile nature of diag(): If its argument is a matrix, it returns a vector, and vice versa. Also, if the argument is a scalar, the function returns the identity matrix of the specified size.

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

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

```
diag(dm)
```

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

```
diag(3)
```

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

The sweep() function is capable of fairly complex operations. As a simple example, let's take a 3-by-3 matrix and add 1 to row 1, 4 to row 2, and 7 to row 3.

```
m<-matrix(c(1,4,7,2,5,8,3,6,9),nrow=3)
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 to sweep() are like those of apply(): the array and the margin, which is 1 for rows in this case. The fourth argument is a function to be applied, and the third is an argument to that function (to the "+" function).

Set Operations

R includes some handy set operations, including these: - 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

Here are some simple examples of using these functions:

```
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
```

Recall that you can write your own binary operations. For instance, consider coding the symmetric difference between two sets— that is, all the elements belonging to exactly one of the two operand sets. Because the symmetric difference between sets x and y consists exactly of those elements in x but not y and vice versa, the code consists of easy calls to setdiff() and union(), as follows:

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

Let's try it.

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

y
## [1] 5 1 8 9

symdiff(x,y)
## [1] 2 8 9
```

Here's another example: a binary operand for determining whether one set u is a subset of another set v. A bit of thought shows that this property is equivalent to the intersection of u and v being equal to u. Hence we have another easily coded function:

```
"%subsetof%" <- function(u,v) {
```

```
 return(setequal(intersect(u,v),u))
}

c(3,8) %subsetof% 1:10
## [1] TRUE

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

The function combn() generates combinations. Let's find the subsets of {1,2,3} of size 2.

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

class(c32)
## [1] "matrix" "array"
```

The results are in the columns of the output. We see that the subsets of {1,2,3} of size 2 are (1,2), (1,3), and (2,3). The function also allows you to specify a function to be called by combn() on each combination. For example, we can find the sum of the numbers in each subset, like this:

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

The first subset, {1,2}, has a sum of 2, and so on.