8

DOING MATH AND SIMULATIONS IN R

R contains built-in functions for your favorite math operations and, of course, for statistical distributions. This chapter provides an overview of using these functions. Given the mathematical nature of this chapter, the examples assume a slightly higher-level knowledge than those in other chapters. You should be familiar with calculus and linear algebra to get the most out of these examples.

8.1 Math Functions

R includes an extensive set of built-in math functions. Here is a partial list:

- `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()`: Cumulative sum and product of the elements of a vector
- `round()`, `floor()`, and `ceiling()`: Round to the closest integer, to the closest integer below, and to the closest integer above
- `factorial()`: Factorial function

8.1.1 Extended Example: Calculating a Probability

As our first example, we'll work through calculating a probability using the `prod()` function. 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?

Suppose first that $n = 3$ and our events are named A, B, and C. Then we break down the computation as follows:

$$\begin{aligned} \text{P(exactly one event occurs)} &= \\ \text{P(A and not B and not C)} &+ \\ \text{P(not A and B and not C)} &+ \\ \text{P(not A and not B and C)} & \end{aligned}$$

P(A and not B and not C) would be $p_A(1-p_B)(1-p_C)$, and so on.

For general n, that is calculated as follows:

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

(The i^{th} term inside the sum is the probability that event i occurs and all the others do *not* occur.)

Here's code to compute this, with our probabilities p_i contained in the vector `p`:

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

How does it work? Well, the assignment

```
notp <- 1 - p
```

creates a vector of all the "not occur" probabilities $1 - p_j$, using recycling. The expression `notp[-i]` computes the product of all the elements of `notp`, except the i^{th}—exactly what we need.

8.1.2 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.

8.1.3 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
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    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().

Function minimization/maximization can be done via nlm() and optim(). For example, let's find 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
```

Here, the minimum value was found to be approximately -0.23, occurring at $x = 0.45$. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case. The second argument specifies the initial guess, which we set to be 8. (This second argument was picked pretty arbitrarily here, but in some problems, you may need to experiment to find a value that will lead to convergence.)

8.1.4 Calculus

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

```
> 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
```

Here, R reported

$$\frac{d}{dx}e^{x^2} = 2xe^{x^2}$$

and

$$\int_0^1 x^2 \, dx \approx 0.3333333$$

You can find R packages for differential equations (`odesolve`), for interfacing R with the Yacas symbolic math system (`ryacas`), and for other calculus operations. These packages, and thousands of others, are available from the Comprehensive R Archive Network (CRAN); see Appendix B.

8.2 Functions for Statistical Distributions

R has functions available for most of the famous 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

The rest of the name indicates the distribution. Table 8-1 lists some common statistical distribution functions.

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()`

As an example, let's simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

```
> mean(rchisq(1000,df=2))
[1] 1.938179
```

The `r` in `rchisq` specifies that we wish to generate random numbers—in this case, from the chi-square distribution. As seen in this example, the first argument in the `r`-series functions is the number of random variates to generate.

These functions also have arguments specific to the given distribution families. In our example, we use the `df` argument for the chi-square family, indicating the number of degrees of freedom.

NOTE *Consult R's online help for details on the arguments for the statistical distribution functions. For instance, to find our more about the chi-square function for quantiles, type* `?qchisq` *at the command prompt.*

Let's also compute the 95th percentile of the chi-square distribution with two degrees of freedom:

```
> qchisq(0.95,2)
[1] 5.991465
```

Here, we used `q` to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile.

The first argument in the `d`, `p`, and `q` series is actually a vector so that we can evaluate the density/pmf, cdf, or quantile function at multiple points. Let's find both the 50th and 95th percentiles of the chi-square distribution with 2 degrees of freedom.

```
qchisq(c(0.5,0.95),df=2)
[1] 1.386294 5.991465
```

8.3 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
    V1 V2
1  def  2
2   ab  5
3 zzzz  1
> r <- order(y$V2)
> r
[1] 3 1 2
> z <- y[r,]
> z
    V1 V2
3 zzzz  1
1  def  2
2   ab  5
```

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

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