MODULE 3

DOING MATH AND SIMULATION IN R

Math Function

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

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 pi 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:

P(exactly one event occurs) =

P(A and not B and not C) +

P(not A and B and not C) +

P(not A and not B and C)

P(A and not B and not C) would be pA(1 − pB)(1 − pC), 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 pi 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 $1 - pj$, using recycling. The expression notp[-i] computes the product of all the elements of notp, except the i th—exactly what we need.

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

[,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) = x2 − 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.)

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);

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.

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

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.

> 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

[1] 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 · 5+2 · 12 + 3 · 13 = 68. Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product. We'll develop a function to compute real cross products in Section 8.4.1. 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

[,1] [,2]

[1,] 1 2

[2,] 3 4

> b

[,1] [,2]

[1,] 1 -1

[2,] 0 1

>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 = 2

−x1 + x2 = 4

Its matrix form is as follows:

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

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

> m

[,1] [,2]

[1,] 1 2

[2,] 7 8

> 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

Extended Example: Vector Cross Product

Let's consider the issue of vector cross products. The definition is very simple: The cross product of vectors (x1, x2, x3) and (y1, y2, y3) in threedimensional space is a new three-dimensional vector, as shown in Equation 8.1.

$$(x_2y_3 - x_3y_2, -x_1y_3 + x_3y_1, x_1y_2 - x_2y_1) \qquad (8.1)$$

This can be expressed compactly as the expansion along the top row of the determinant, as shown in Equation 8.2.

$$\begin{pmatrix} - & - & - \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix} \qquad (8.2)$$

Here, the elements in the top row are merely placeholders. Don't worry about this bit of pseudomath. The point is that the cross product vector can be computed as a sum of subdeterminants. For instance, the first component in Equation 8.1, x2y3 − x3y2, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column in Equation 8.2, as shown in Equation 8.3.

$$\begin{pmatrix} x_2 & x_3 \\ y_2 & y_3 \end{pmatrix} \qquad (8.3)$$

Our need to calculate subdeterminants—that is determinants of submatrices—fits perfectly with R, which excels at specifying submatrices. This suggests calling det() on the proper submatrices, as follows:

```
xprod <- function(x,y) {

m <- rbind(rep(NA,3),x,y)

xp <- vector(length=3)

for (i in 1:3)

xp[i] <- -(-1)^i * det(m[2:3,-i])

return(xp)

}
```

Note that even R's ability to specify values as NA came into play here to deal with the "placeholders" mentioned above. All this may seem like overkill. After all, it wouldn't have been hard to code Equation 8.1 directly, without resorting to use of submatrices and determinants. But while that may be true in the three-dimensional case, the approach shown here is quite fruitful in the n-ary case, in n-dimensional space. The cross product there is defined as an n-by-n determinant of the form shown in Equation 8.1, and thus the preceding code generalizes perfectly.

Extended Example: Finding Stationary Distributions of Markov Chains

A Markov chain is a random process in which we move among various states, in a "memoryless" fashion, whose definition need not concern us here. The state could be the number of jobs in a queue, the number of items stored in inventory, and so on. We will assume the number of states to be finite. As a simple example, consider a game in which we toss a coin repeatedly and win a dollar whenever we accumulate three consecutive heads. Our state at any time i will the number of consecutive heads

we have so far, so our state can be 0, 1, or 2. (When we get three heads in a row, our state reverts to 0.)

The central interest in Markov modeling is usually the long-run state distribution, meaning the long-run proportions of the time we are in each state. In our coin-toss game, we can use the code we'll develop here to calculate that distribution, which turns out to have us at states 0, 1, and 2 in proportions 57.1%, 28.6%, and 14.3% of the time. Note that we win our dollar if we are in state 2 and toss a head, so 0.143 × 0.5 = 0.071 of our tosses will result in wins.

sult in wins. Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads. Let pij denote the transition probability of moving from state i to state j during a time step. In the game example, for instance, p23 = 0.5, reflecting the fact that with probability 1/2, we will toss a head and thus move from having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus p21 = 0.5. We are interested in calculating the vector π = (π1, ..., πs), where πi is the long-run proportion of time spent at state i, over all states i. Let P denote the transition probability matrix whose i th row, j th column element is pij . Then it can be shown that π must satisfy Equation 8.4,

$$\pi = \pi P \qquad (8.4)$$

which is equivalent to Equation 8.5:

$$(I - P^T)\pi = 0 \qquad (8.5)$$

Here I is the identity matrix and P T denotes the transpose of P. Any single one of the equations in the system of Equation 8.5 is redundant. We thus eliminate one of them, by removing the last row of I−P in Equation 8.5. That also means removing the last 0 in the 0 vector on the right-hand side of Equation 8.5. But note that there is also the constraint shown in Equation 8.6.

$$\sum_i \pi_i = 1 \qquad (8.6)$$

In matrix terms, this is as follows:

$$1_n^T \pi = 1$$

where 1n is a vector of n 1s. So, in the modified version of Equation 8.5, we replace the removed row with a row of all 1s and, on the right-hand side, replace the removed 0 with a 1. We can then solve the system. All this can be computed with R's solve() function, as follows:

```
findpi1 <- function(p) {
n <- nrow(p)
imp <- diag(n) - t(p)
imp[n,] <- rep(1,n)
rhs <- c(rep(0,n-1),1)
pivec <- solve(imp,rhs)
return(pivec)
```

}

Here are the main steps:

1. Calculate I − P T in line 3. Note again that diag(), when called with a scalar argument, returns the identity matrix of the size given by that argument.
2. Replace the last row of P with 1 values in line 4.
3. Set up the right-hand side vector in line 5.
4. Solve for π in line 6.

Another approach, using more advanced knowledge, is based on eigenvalues. Note from Equation 8.4 that π is a left eigenvector of P with eigenvalue 1. This suggests using R's eigen() function, selecting the eigenvector corresponding to that eigenvalue. (A result from mathematics, the PerronFrobenius theorem, can be used to carefully justify this.) Since π is a left eigenvector, the argument in the call to eigen() must be P transpose rather than P. In addition, since an eigenvector is unique only up to scalar multiplication, we must deal with two issues regarding the eigenvector returned to us by eigen():

- It may have negative components. If so, we multiply by −1.
- It may not satisfy Equation 8.6. We remedy this by dividing by the length of the returned vector.

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 from Section 7.12 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))
}
```

Input /out put

R is not the tool you would choose for running an ATM, but it features a highly versatile array of I/O capabilities, as you will learn in this chapter. We'll start with the basics of access to the keyboard and monitor, and then go into considerable detail on reading and writing files, including the navigation of file directories. Finally, we discuss R's facilities for accessing the Internet.

Accessing the Keyboard and Monitor

R provides several functions for accesssing the keyboard and monitor. Here, we'll look at the scan(), readline(), print(), and cat() functions.

Using the scan() Function

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list. Suppose we have files named z1.txt, z2.txt, z3.txt, and z4.txt. The z1.txt file contains the following:

```
123
4 5
6
```

The z2.txt file contents are as follows:

123

4.2 5

6

The z3.txt file contains this:

abc

de f

g

And finally, the z4.txt file has these contents:

abc

123 6

Y

Let’s see what we can do with these files using the scan() function.

```
> scan("z1.txt")
Read 4 items
[1] 123 4 5 6
> scan("z2.txt")
Read 4 items
[1] 123.0 4.2 5.0 6.0
> scan("z3.txt")
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
scan() expected 'a real', got 'abc'
> scan("z3.txt",what="")
Read 4 items
[1] "abc" "de" "f" "g"
> scan("z4.txt",what="")
Read 4 items
[1] "abc" "123" "6" "y"
```

In the first call, we got a vector of four integers (though the mode is numeric). The second time, since one number was nonintegral, the others were shown as floating-point numbers, too. In the third case, we got an error. The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So, the nonnumeric contents of the file z3 produced an error. But we then tried again, with what="". This assigns a character string to what, indicating that we want character mode. (We could have set what to any character string.) The last call worked the same way. The first

item was a character string, so it treated all the items that followed as strings too. Of course, in typical usage, we would assign the return value of scan() to a variable. Here's an example:

```
> v <- scan("z1.txt")
```

By default, scan() assumes that the items of the vector are separated by whitespace, which includes blanks, carriage return/line feeds, and horizontal tabs. You can use the optional sep argument for other situations. As example, we can set sep to the newline character to read in each line as a string, as follows:

```
> x1 <- scan("z3.txt",what="")
Read 4 items
> x2 <- scan("z3.txt",what="",sep="\n")
Read 3 items
> x1
[1] "abc" "de" "f" "g"
> x2
[1] "abc" "de f" "g"
> x1[2]
[1] "de"
> x2[2]
[1] "de f"
```

In the first case, the strings "de" and "f" were assigned to separate elements of x1. But in the second case, we specified that elements of x2 were to be delineated by end-of-line characters, not spaces. Since "de" and "f" are on the same line, they are assigned together to x[2]. More sophisticated methods for reading files will be presented later in this chapter, such as methods to read in a file one line at a time. But if you want to read the entire file at once, scan() provides a quick solution.

You can use scan() to read from the keyboard by specifying an empty string for the filename:

```
> v <- scan("")
1: 12 5 13
4: 3 4 5
7: 8
8:
Read 7 items
> v
[1] 12 5 13 3 4 5 8
```

Using the readline() Function

If you want to read in a single line from the keyboard, readline() is very handy

```
> w <- readline()
abc de f
> w
[1] "abc de f"
```

Typically, readline() is called with its optional prompt, as follows:

```
> inits <- readline("type your initials: ")
type your initials: NM
> inits
[1] "NM"
```

Printing to the Screen

At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression. This won't work if you need to print from within the body of a function. In that case, you can use the print() function, like this:

```
> x <- 1:3
> print(x^2)
[1] 1 4 9
```

Recall that print() is a generic function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class "table", then the print.table() function will be called.

It's a little better to use cat() instead of print(), as the latter can print only one expression and its output is numbered, which may be a nuisance. Compare the results of the functions:

```
> print("abc")
[1] "abc"
> cat("abc\n")
Abc
```

Note that we needed to supply our own end-of-line character, "\n", in the call to cat(). Without it, our next call would continue to write to the same line. The arguments to cat() will be printed out with intervening spaces:

```
> x
[1] 1 2 3
> cat(x,"abc","de\n")
1 2 3 abc de
```

If you don’t want the spaces, set sep to the empty string "", as follows:

```
> cat(x,"abc","de\n",sep="")
123abcde
```

Any string can be used for sep. Here, we use the newline character:

```
> cat(x,"abc","de\n",sep="\n")
1
2
3
abc
de
```

You can even set sep to be a vector of strings, like this:

```
> x <- c(5,12,13,8,88)
> cat(x,sep=c(".",".",".","\n","\n"))
5.12.13.8
88
```

Reading and Writing Files

Now that we’ve covered the basics of I/O, let’s get to some more practical applications of reading and writing files. The following sections discuss reading data frames or matrices from files, working with text files, accessing files on remote machines, and getting file and directory information.

Reading a Data Frame or Matrix from a File

In Section 5.1.2, we discussed the use of the function read.table() to read in a data frame. As a quick review, suppose the file z looks like this:

```
name age
John 25
Mary 28
Jim 19
```

The first line contains an optional header, specifying column names. We could read the file this way:

```
> z <- read.table("z",header=TRUE)
> z
  name age
1 John 25
2 Mary 28
```

3 Jim 19

Note that scan() would not work here, because our file has a mixture of numeric and character data (and a header). There appears to be no direct way of reading in a matrix from a file, but it can be done easily with other tools. A simple, quick way is to use scan() to read in the matrix row by row. You use the byrow option in the function matrix() to indicate that you are defining the elements of the matrix in a row-wise, rather than column-wise, manner. For instance, say the file x contains a 5-by-3 matrix, stored row-wise:

101

111

110

110

001

We can read it into a matrix this way:

> x <- matrix(scan("x"),nrow=5,byrow=TRUE)

This is fine for quick, one-time operations, but for generality, you can use read.table(), which returns a data frame, and then convert via as.matrix(). Here is a general method:

read.matrix <- function(filename) {

as.matrix(read.table(filename))

}

Reading Text Files

In computer literature, there is often a distinction made between text files and binary files. That distinction is somewhat misleading—every file is binary in the sense that it consists of 0s and 1s. Let's take the term text file to mean a file that consists mainly of ASCII characters or coding for some other human language (such as GB for Chinese) and that uses newline characters to give humans the perception of lines. The latter aspect will turn out to be central here. Nontext files, such as JPEG images or executable program files, are generally called binary files. You can use readLines() to read in a text file, either one line at a time or in a single operation. For example, suppose we have a file z1 with the following contents:

John 25

Mary 28

Jim 19

We can read the file all at once, like this:

> z1 <- readLines("z1")

> z1

[1] "John 25" "Mary 28" "Jim 19"

Since each line is treated as a string, the return value here is a vector of strings—that is, a vector of character mode. There is one vector element for each line read, thus three elements here. Alternatively, we can read it in one line at a time. For this, we first need to create a connection, as described next.

Writing to a File

Given the statistical basis of R, file reads are probably much more common than writes. But writes are sometimes necessary, and this section will present methods for writing to files.

The function write.table() works very much like read.table(), except that it writes a data frame instead of reading one.

```
> kids <- c("Jack","Jill")

> ages <- c(12,10)

> d <- data.frame(kids,ages,stringsAsFactors=FALSE)

> d

kids ages

1 Jack 12

2 Jill 10

> write.table(d,"kds")
```

The file kds will now have these contents:

```
"kids" "ages"

"1" "Jack" 12

"2" "Jill" 10
```

In the case of writing a matrix to a file, just state that you do not want row or column names, as follows:

```
> write.table(xc,"xcnew",row.names=FALSE,col.names=FALSE)
```

The function cat() can also be used to write to a file, one part at a time. Here's an example:

```
> cat("abc\n",file="u")

> cat("de\n",file="u",append=TRUE)
```

The first call to cat() creates the file u, consisting of one line with contents "abc". The second call appends a second line. Unlike the case of using the writeLines() function (which we'll discuss next), the file is automatically saved after each operation. For instance, after the previous calls, the file will look like this:

```
abc

de
```

You can write multiple fields as well. So:

```
> cat(file="v",1,2,"xyz\n")
```

would produce a file v consisting of a single line:

1 2 xyz

You can also use writeLines(), the counterpart of readLines(). If you use a connection, you must specify "w" to indicate you are writing to the file, not reading from it:

```
> c <- file("www","w")
> writeLines(c("abc","de","f"),c)
> close(c)
```

The file www will be created with these contents:

abc

de

f

Note the need to proactively close the file.