UNIT-III

Doing Math and Simulation in R, Math Function, Extended Example Calculating Probability-Cumulative Sums and Products-Minima and Maxima- Calculus, Functions For Statistical Distribution, Sorting, Linear Algebra Operation on Vectors and Matrices, Extended Example: Vector cross Product- Extended Example: Finding Stationary Distribution of Markov Chains, Set Operation, Input /out put, Accessing the Keyboard and Monitor, Reading and writer Files.

DOING MATH AND SIMULATIONS IN R

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

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 ith 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:

_n

i=1

pi(1 − p1)...(1 − pi−1)(1 − pi+1)...(1 − pn)

(The ith 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)
}
```

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.

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()

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.

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

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.

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
```

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.

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 For matrix multiplication in the mathematical sense, the operator to use

is %*%, not *. For instance, here we compute the matrix product:

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

```
> 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:

$$x_1 + x_2 = 2$$

$$-x_1 + x_2 = 4$$

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
```

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
[,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
```

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
[,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 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).

Extended Example: Vector Cross Product

Let's consider the issue of vector cross products. The definition is very simple: The cross product of vectors (*x*1, *x*2, *x*3) and (*y*1, *y*2, *y*3) in three-dimensional space is a new three-dimensional vector,

(*x*2*y*3 − *x*3*y*2,−*x*1*y*3 + *x*3*y*1, *x*1*y*2 − *x*2*y*1)

This can be expressed compactly as the expansion along the top row of the determinant pseudomath. The point is that the cross product vector can be computed as a sum of subdeterminants.

$$\begin{pmatrix} - & - & - \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}$$

*x*2*y*3 − *x*3*y*2, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column

$$\begin{pmatrix} x_2 & x_3 \\ y_2 & y_3 \end{pmatrix}$$

```
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)
}
```

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.)

```
1 findpi2 <- function(p) {
2 n <- nrow(p)
3 # find first eigenvector of P transpose
4 pivec <- eigen(t(p))$vectors[,1]
5 # guaranteed to be real, but could be negative
6 if (pivec[1] < 0) pivec <- -pivec
7 # normalize to sum to 1
8 pivec <- pivec / sum(pivec)
9 return(pivec)
10 }
```

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
> 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
```

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"

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).

> combn(1:3,2,sum)

[1] 3 4 5

INPUT/OUTPUT

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"
> 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"

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
```

Note that we are prompted with the index of the next item to be input, and we signal the end of input with an empty line. If you do not wish scan() to announce the number of items it has read, include the argument quiet=TRUE.

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

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:

1 0 1
1 1 1
1 1 0
1 1 0
0 0 1

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.

Introduction to Connections

Connection is R's term for a fundamental mechanism used in various kinds of I/O operations. Here, it will be used for file access.

The connection is created by calling file(), url(), or one of several other
R functions. To see a list of those functions, type this:

```
> ?connection
```
So, we can now read in the *z1* file (introduced in the previous section)
line by line, as follows:
```
> c <- file("z1","r")
> readLines(c,n=1)
[1] "John 25"
> readLines(c,n=1)
[1] "Mary 28"
> readLines(c,n=1)
[1] "Jim 19"
> readLines(c,n=1)
character(0)
```
We opened the connection, assigned the result to c, and then read the file one line at a time, as specified by the argument n=1. When R encountered the end of file (EOF), it returned an empty result. We needed to set up a connection so that R could keep track of our position in the file as we read through it.
We can detect EOF in our code:
```
> c <- file("z","r")
> while(TRUE) {
+ rl <- readLines(c,n=1)
+ if (length(rl) == 0) {
+ print("reached the end")
+ break
+ } else print(rl)
+ }
[1] "John 25"
[1] "Mary 28"
[1] "Jim 19"
[1] "reached the end"
```
If we wish to “rewind”—to start again at the beginning of the file—we
can use seek():
```
> c <- file("z1","r")
```

```
> readLines(c,n=2)
[1] "John 25" "Mary 28"
> seek(con=c,where=0)
[1] 16
> readLines(c,n=1)
[1] "John 25"
```

The argument where=0 in our call to seek() means that we wish to position the file pointer zero characters from the start of the file—in other words, directly at the beginning.

The call returns 16, meaning that the file pointer was at position 16 before we made the call. That makes sense. The first line consists of "John 25" *plus* the end-of-line character, for a total of eight characters, and the same is true for the second line. So, after reading the first two lines, we were at position 16.

You can close a connection by calling—what else?—close(). You would use this to let the system know that the file you have been writing is complete and should now be officially written to disk. As another example, in a client/server relationship over the Internet (see Section 10.3.1), a client would use close() to indicate to the server that the client is signing off.

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. For instance, let's take the little Jack and Jill example

```
> 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.