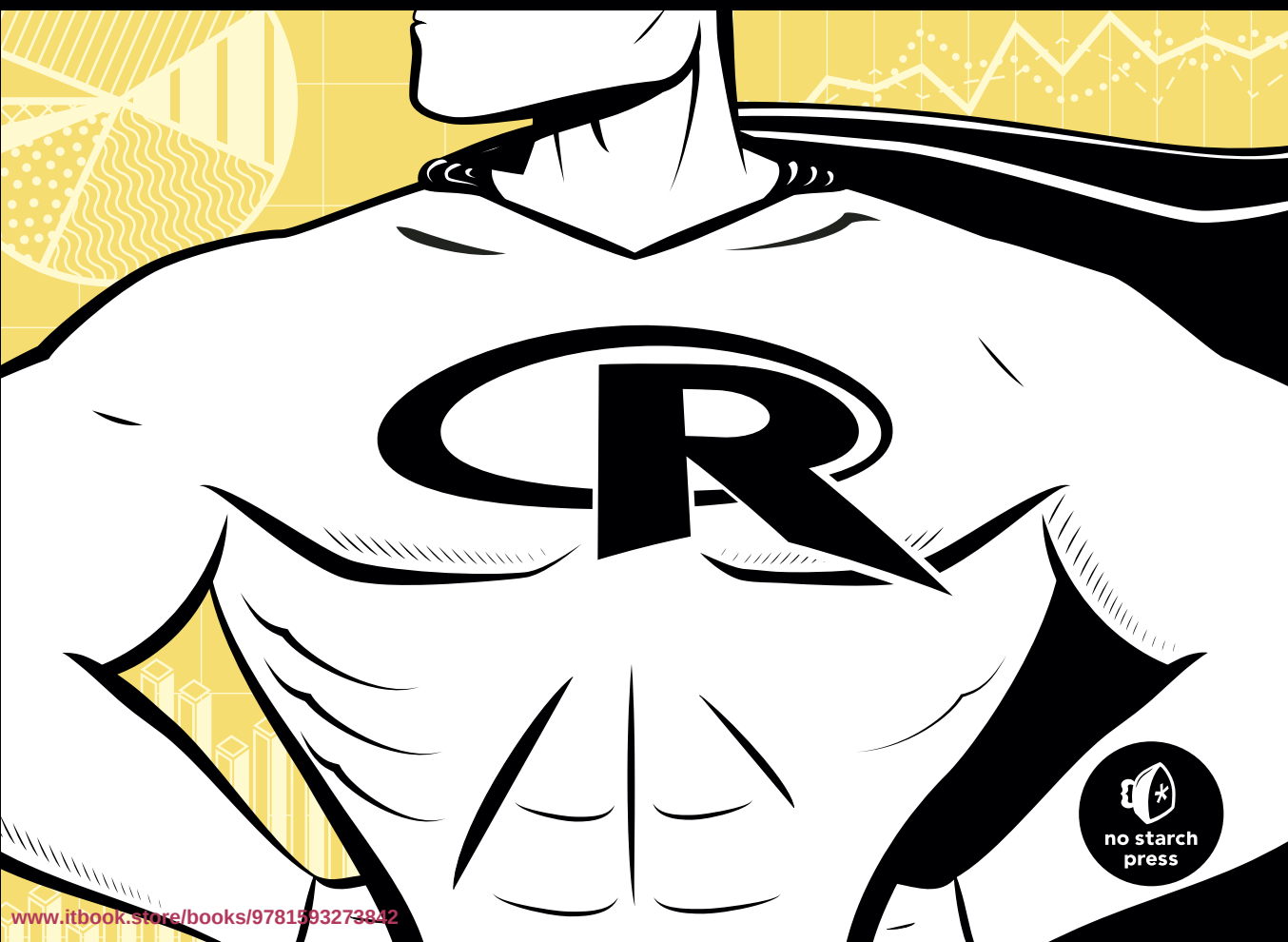
THE ART OF R PROGRAMMING

A *TOUR* OF STATISTICAL SOFTWARE DESIGN

NORMAN MATLOFF



no starch press

3

MATRICES AND ARRAYS

A *matrix* is a vector with two additional attributes: the number of rows and the number of columns. Since matrices are vectors, they also have modes, such as numeric and character. (On the other hand, vectors are *not* one-column or one-row matrices.)

Matrices are special cases of a more general R type of object: *arrays*. Arrays can be multidimensional. For example, a three-dimensional array would consist of rows, columns, and layers, not just rows and columns as in the matrix case. Most of this chapter will concern matrices, but we will briefly discuss higher-dimensional arrays in the final section.

Much of R's power comes from the various operations you can perform on matrices. We'll cover these operations in this chapter, especially those analogous to vector subsetting and vectorization.

3.1 Creating Matrices

Matrix row and column subscripts begin with 1. For example, the upper-left corner of the matrix `a` is denoted `a[1,1]`. The internal storage of a matrix is in *column-major order*, meaning that first all of column 1 is stored, then all of column 2, and so on, as you saw in Section 2.1.3.

One way to create a matrix is by using the `matrix()` function:

```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Here, we concatenate what we intend as the first column, the numbers 1 and 2, with what we intend as the second column, 3 and 4. So, our data is `(1,2,3,4)`. Next, we specify the number of rows and columns. The fact that R uses column-major order then determines where these four numbers are put within the matrix.

Since we specified the matrix entries in the preceding example, and there were four of them, we did not need to specify both `ncol` and `nrow`; just `nrow` or `ncol` would have been enough. Having four elements in all, in two rows, implies two columns:

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

Note that when we then print out `y`, R shows us its notation for rows and columns. For instance, `[,2]` means the entirety of column 2, as can be seen in this check:

```
> y[,2]
[1] 3 4
```

Another way to build `y` is to specify elements individually:

```
> y <- matrix(nrow=2,ncol=2)
> y[1,1] <- 1
> y[2,1] <- 2
> y[1,2] <- 3
> y[2,2] <- 4
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Note that we do need to warn R ahead of time that `y` will be a matrix and give the number of rows and columns.

Though internal storage of a matrix is in column-major order, you can set the `byrow` argument in `matrix()` to true to indicate that the data is coming in row-major order. Here's an example of using `byrow`:

```
> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Note that the matrix is still stored in column-major order. The `byrow` argument enabled only our *input* to come in row-major form. This may be more convenient if you are reading from a data file organized that way, for example.

3.2 General Matrix Operations

Now that we've covered the basics of creating a matrix, we'll look at some common operations performed with matrices. These include performing linear algebra operations, matrix indexing, and matrix filtering.

3.2.1 Performing Linear Algebra Operations on Matrices

You can perform various linear algebra operations on matrices, such as matrix multiplication, matrix scalar multiplication, and matrix addition. Using `y` from the preceding example, here is how to perform those three operations:

```
> y %*% y  # mathematical matrix multiplication
  [,1] [,2]
[1,] 7   15
[2,]10   22
> 3*y  # mathematical multiplication of matrix by scalar
  [,1] [,2]
[1,] 3    9
[2,] 6   12
> y+y  # mathematical matrix addition
  [,1] [,2]
[1,] 2    6
[2,] 4    8
```

For more on linear algebra operations on matrices, see Section 8.4.

3.2.2 Matrix Indexing

The same operations we discussed for vectors in Section 2.4.2 apply to matrices as well. Here's an example:

```
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> z[,2:3]
  [,1] [,2]
[1,] 1    1
[2,] 1    0
[3,] 0    1
[4,] 0    0
```

Here, we requested the submatrix of z consisting of all elements with column numbers 2 and 3 and any row number. This extracts the second and third columns.

Here's an example of extracting rows instead of columns:

```
> y
  [,1] [,2]
[1,]11   12
[2,]21   22
[3,]31   32
> y[2:3,]
  [,1] [,2]
[1,]21   22
[2,]31   32
> y[2:3,2]
[1] 22 32
```

You can also assign values to submatrices:

```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[c(1,3),] <- matrix(c(1,1,8,12),nrow=2)
> y
     [,1] [,2]
[1,]    1    8
[2,]    2    5
[3,]    1   12
```

Here, we assigned new values to the first and third rows of y.

And here's another example of assignment to submatrices:

```
> x <- matrix(nrow=3,ncol=3)
> y <- matrix(c(4,5,2,3),nrow=2)
> y
     [,1] [,2]
[1,]    4    2
[2,]    5    3
> x[2:3,2:3] <- y
> x
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA    4    2
[3,]   NA    5    3
```

Negative subscripts, used with vectors to exclude certain elements, work the same way with matrices:

```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[-2,]
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

In the second command, we requested all rows of y except the second.

3.2.3 Extended Example: Image Manipulation

Image files are inherently matrices, since the pixels are arranged in rows and columns. If we have a grayscale image, for each pixel, we store the *intensity*—the brightness–of the image at that pixel. So, the intensity of a pixel in, say, row 28 and column 88 of the image is stored in row 28, column 88 of the matrix. For a color image, three matrices are stored, with intensities for red, green, and blue components, but we'll stick to grayscale here.

For our example, let's consider an image of the Mount Rushmore National Memorial in the United States. Let's read it in, using the `pixmap` library. (Appendix B describes how to download and install libraries.)

```
> library(pixmap)
> mtrush1 <- read.pnm("mtrush1.pgm")
> mtrush1
Pixmap image
  Type          : pixmapGrey
```

```
  Size          : 194x259
  Resolution    : 1x1
  Bounding box  : 0 0 259 194
> plot(mtrush1)
```

We read in the file named *mtrush1.pgm*, returning an object of class `pixmap`. We then plot it, as seen in Figure 3-1.

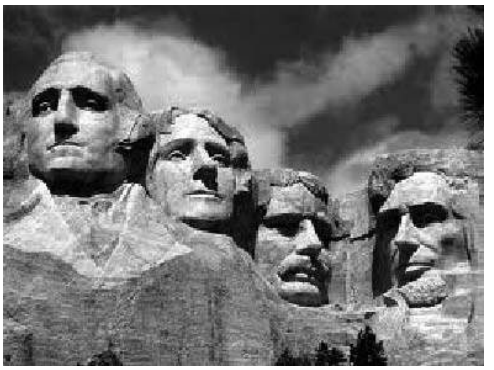


Figure 3-1: Reading in Mount Rushmore

Now, let's see what this class consists of:

```
> str(mtrush1)
Formal class 'pixmapGrey' [package "pixmap"] with 6 slots
  ..@ grey    : num [1:194, 1:259] 0.278 0.263 0.239 0.212 0.192 ...
  ..@ channels: chr "grey"
  ..@ size    : int [1:2] 194 259
...
```

The class here is of the S4 type, whose components are designated by `@`, rather than `$`. S3 and S4 classes will be discussed in Chapter 9, but the key item here is the intensity matrix, `mtrush1@grey`. In the example, this matrix has 194 rows and 259 columns.

The intensities in this class are stored as numbers ranging from 0.0 (black) to 1.0 (white), with intermediate values literally being shades of gray. For instance, the pixel at row 28, column 88 is pretty bright.

```
> mtrush1@grey[28,88]
[1] 0.7960784
```

To demonstrate matrix operations, let's blot out President Roosevelt. (Sorry, Teddy, nothing personal.) To determine the relevant rows and columns, you can use R's `locator()` function. When you call this function, it

waits for the user to click a point within a graph and returns the exact coordinates of that point. In this manner, I found that Roosevelt's portion of the picture is in rows 84 through 163 and columns 135 through 177. (Note that row numbers in `pixmap` objects increase from the top of the picture to the bottom, the opposite of the numbering used by `locator()`.) So, to blot out that part of the image, we set all the pixels in that range to 1.0.

```
> mtrush2 <- mtrush1
> mtrush2@grey[84:163,135:177] <- 1
> plot(mtrush2)
```

The result is shown in Figure 3-2.

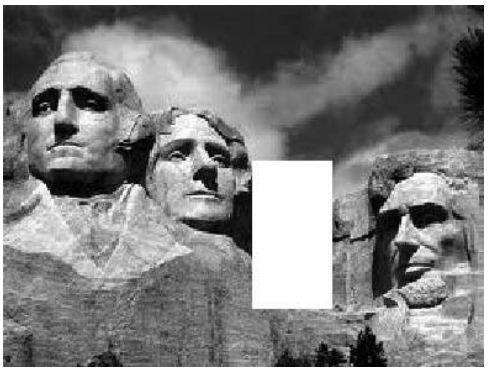


Figure 3-2: Mount Rushmore, with President Roosevelt removed

What if we merely wanted to disguise President Roosevelt's identity? We could do this by adding random noise to the picture. Here's code to do that:

```
# adds random noise to img, at the range rows,cols of img; img and the
# return value are both objects of class pixmap; the parameter q
# controls the weight of the noise, with the result being 1-q times the
# original image plus q times the random noise
blurpart <- function(img,rows,cols,q) {
  lrows <- length(rows)
  lcols <- length(cols)
  newimg <- img
  randomnoise <- matrix(nrow=lrows, ncol=ncols,runif(lrows*lcols))
  newimg@grey <- (1-q) * img@grey + q * randomnoise
  return(newimg)
}
```

As the comments indicate, we generate random noise and then take a weighted average of the target pixels and the noise. The parameter `q` controls the weight of the noise, with larger `q` values producing more blurring. The random noise itself is a sample from U(0,1), the uniform distribution on the interval (0,1). Note that the following is a matrix operation:

```
newimg@grey <- (1-q) * img@grey + q * randomnoise
```

So, let's give it a try:

```
> mtrush3 <- blurpart(mtrush1,84:163,135:177,0.65)
> plot(mtrush3)
```

The result is shown in Figure 3-3.

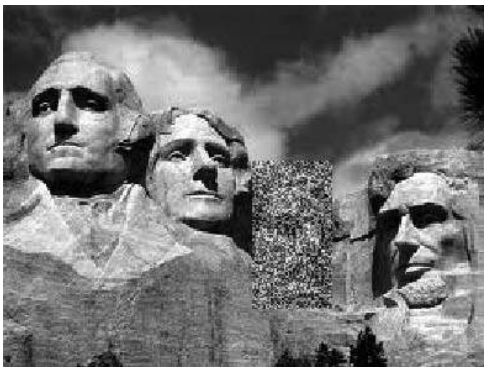


Figure 3-3: Mount Rushmore, with President Roosevelt blurred

3.2.4 Filtering on Matrices

Filtering can be done with matrices, just as with vectors. You must be careful with the syntax, though. Let's start with a simple example:

```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

Again, let's dissect this, just as we did when we first looked at filtering in Chapter 2:

```
> j <- x[,2] >= 3
> j
[1] FALSE  TRUE  TRUE
```

Here, we look at the vector `x[,2]`, which is the second column of `x`, and determine which of its elements are greater than or equal to 3. The result, assigned to `j`, is a Boolean vector.

Now, use `j` in `x`:

```
> x[j,]
     x
[1,] 2 3
[2,] 3 4
```

Here, we compute `x[j,]`—that is, the rows of `x` specified by the true elements of `j`—getting the rows corresponding to the elements in column 2 that were at least equal to 3. Hence, the behavior shown earlier when this example was introduced:

```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

For performance purposes, it's worth noting again that the computation of `j` here is a completely vectorized operation, since all of the following are true:

- The object `x[,2]` is a vector.
- The operator `>=` compares two vectors.
- The number 3 was recycled to a vector of 3s.

Also note that even though `j` was defined in terms of `x` and then was used to extract from `x`, it did not need to be that way. The filtering criterion can be based on a variable separate from the one to which the filtering will be applied. Here's an example with the same `x` as above:

```
> z <- c(5,12,13)
> x[z %% 2 == 1,]
     [,1] [,2]
```

```
[1,]    1    4
[2,]    3    6
```

Here, the expression `z %% 2 == 1` tests each element of `z` for being an odd number, thus yielding `(TRUE,FALSE,TRUE)`. As a result, we extracted the first and third rows of `x`.

Here is another example:

```
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m[m[,1] > 1 & m[,2] > 5,]
[1] 3 6
```

We're using the same principle here, but with a slightly more complex set of conditions for row extraction. (Column extraction, or more generally, extraction of any submatrix, is similar.) First, the expression `m[,1] > 1` compares each element of the first column of `m` to 1 and returns `(FALSE,TRUE,TRUE)`. The second expression, `m[,2] > 5`, similarly returns `(FALSE,FALSE,TRUE)`. We then take the logical AND of `(FALSE,TRUE,TRUE)` and `(FALSE,FALSE,TRUE)`, yielding `(FALSE,FALSE,TRUE)`. Using the latter in the row indices of `m`, we get the third row of `m`.

Note that we needed to use `&`, the vector Boolean AND operator, rather than the scalar one that we would use in an `if` statement, `&&`. A complete list of such operators is given in Section 7.2.

The alert reader may have noticed an anomaly in the preceding example. Our filtering should have given us a submatrix of size 1 by 2, but instead it gave us a two-element vector. The elements were correct, but the data type was not. This would cause trouble if we were to then input it to some other matrix function. The solution is to use the `drop` argument, which tells R to retain the two-dimensional nature of our data. We'll discuss `drop` in detail in Section 3.6 when we examine unintended dimension reduction.

Since matrices are vectors, you can also apply vector operations to them. Here's an example:

```
> m
     [,1] [,2]
[1,]    5   -1
[2,]    2   10
[3,]    9   11
> which(m > 2)
[1] 1 3 5 6
```

R informed us here that, from a vector-indexing point of view, elements 1, 3, 5, and 6 of `m` are larger than 2. For example, element 5 is the element in row 2, column 2 of `m`, which we see has the value 10, which is indeed greater than 2.

3.2.5 Extended Example: Generating a Covariance Matrix

This example demonstrates R's `row()` and `col()` functions, whose arguments are matrices. For example, for a matrix `a`, `row(a[2,8])` will return the row number of that element of `a`, which is 2. Well, we knew `row(a[2,8])` is in row 2, didn't we? So why would this function be useful?

Let's consider an example. When writing simulation code for multivariate normal distributions—for instance, using `mvrnorm()` from the `MASS` library—we need to specify a covariance matrix. The key point for our purposes here is that the matrix is symmetric; for example, the element in row 1, column 2 is equal to the element in row 2, column 1.

Suppose that we are working with an `n`-variate normal distribution. Our matrix will have `n` rows and `n` columns, and we wish each of the `n` variables to have variance 1, with correlation `rho` between pairs of variables. For `n = 3` and `rho = 0.2`, for example, the desired matrix is as follows:

$$\begin{pmatrix} 1 & 0.2 & 0.2 \\ 0.2 & 1 & 0.2 \\ 0.2 & 0.2 & 1 \end{pmatrix}$$

Here is code to generate this kind of matrix:

```
1 makecov <- function(rho,n) {
2    m <- matrix(nrow=n,ncol=n)
3    m <- ifelse(row(m) == col(m),1,rho)
4    return(m)
5 }
```

Let's see how this works. First, as you probably guessed, `col()` returns the column number of its argument, just as `row()` does for the row number. Then the expression `row(m)` in line 3 returns a matrix of integer values, each one showing the row number of the corresponding element of `m`. For instance,

```
> z
     [,1] [,2]
[1,]    3    6
[2,]    4    7
[3,]    5    8
> row(z)
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
```

Thus the expression `row(m) == col(m)` in the same line returns a matrix of `TRUE` and `FALSE` values, `TRUE` values on the diagonal of the matrix and `FALSE` values elsewhere. Once again, keep in mind that binary operators—in this case, `==`—are functions. Of course, `row()` and `col()` are functions too, so this expression:

```
row(m) == col(m)
```

applies that function to each element of the matrix `m`, and it returns a `TRUE/FALSE` matrix of the same size as `m`. The `ifelse()` expression is another function call.

```
ifelse(row(m) == col(m),1,rho)
```

In this case, with the argument being the `TRUE/FALSE` matrix just discussed, the result is to place the values 1 and `rho` in the proper places in our output matrix.

3.3 Applying Functions to Matrix Rows and Columns

One of the most famous and most used features of R is the `*apply()` family of functions, such as `apply()`, `tapply()`, and `lapply()`. Here, we'll look at `apply()`, which instructs R to call a user-specified function on each of the rows or each of the columns of a matrix.

3.3.1 Using the apply() Function

This is the general form of `apply` for matrices:

```
apply(m,dimcode,f,fargs)
```

where the arguments are as follows:

- `m` is the matrix.
- `dimcode` is the dimension, equal to 1 if the function applies to rows or 2 for columns.
- `f` is the function to be applied.
- `fargs` is an optional set of arguments to be supplied to `f`.

For example, here we apply the R function `mean()` to each column of a matrix `z`:

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
> apply(z,2,mean)
[1] 2 5
```

In this case, we could have used the `colMeans()` function, but this provides a simple example of using `apply()`.

A function you write yourself is just as legitimate for use in `apply()` as any R built-in function such as `mean()`. Here's an example using our own function `f`:

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
  [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

Our `f()` function divides a two-element vector by the vector (2,8). (Recycling would be used if `x` had a length longer than 2.) The call to `apply()` asks R to call `f()` on each of the rows of `z`. The first such row is (1,4), so in the call to `f()`, the actual argument corresponding to the formal argument `x` is (1,4). Thus, R computes the value of (1,4)/(2,8), which in R's element-wise vector arithmetic is (0.5,0.5). The computations for the other two rows are similar.

You may have been surprised that the size of the result here is 2 by 3 rather than 3 by 2. That first computation, (0.5,0.5), ends up at the first column in the output of `apply()`, not the first row. But this is the behavior of `apply()`. If the function to be applied returns a vector of *k* components, then the result of `apply()` will have *k* rows. You can use the matrix transpose function `t()` to change it if necessary, as follows:

```
> t(apply(z,1,f))
     [,1]  [,2]
[1,]  0.5 0.500
[2,]  1.0 0.625
[3,]  1.5 0.750
```

If the function returns a scalar (which we know is just a one-element vector), the final result will be a vector, not a matrix.

As you can see, the function to be applied needs to take at least one argument. The formal argument here will correspond to an actual argument of one row or column in the matrix, as described previously. In some cases, you will need additional arguments for this function, which you can place following the function name in your call to `apply()`.

For instance, suppose we have a matrix of 1s and 0s and want to create a vector as follows: For each row of the matrix, the corresponding element of the vector will be either 1 or 0, depending on whether the majority of the first `d` elements in that row is 1 or 0. Here, `d` will be a parameter that we may wish to vary. We could do this:

```
> copymaj
function(rw,d) {
   maj <- sum(rw[1:d]) / d
   return(if(maj > 0.5) 1 else 0)
}
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    1    1    0
[2,]    1    1    1    1    0
[3,]    1    0    0    1    1
[4,]    0    1    1    1    0
> apply(x,1,copymaj,3)
[1] 1 1 0 1
> apply(x,1,copymaj,2)
[1] 0 1 0 0
```

Here, the values 3 and 2 form the actual arguments for the formal argument `d` in `copymaj()`. Let's look at what happened in the case of row 1 of `x`. That row consisted of (1,0,1,1,0), the first `d` elements of which were (1,0,1). A majority of those three elements were 1s, so `copymaj()` returned a 1, and thus the first element of the output of `apply()` was a 1.

Contrary to common opinion, using `apply()` will generally not speed up your code. The benefits are that it makes for very compact code, which may be easier to read and modify, and you avoid possible bugs in writing code for looping. Moreover, as R moves closer and closer to parallel processing, functions like `apply()` will become more and more important. For example, the `clusterApply()` function in the `snow` package gives R some parallel-processing capability by distributing the submatrix data to various network nodes, with each node basically applying the given function on its submatrix.

3.3.2 Extended Example: Finding Outliers

In statistics, *outliers* are data points that differ greatly from most of the other observations. As such, they are treated either as suspect (they might be erroneous) or unrepresentative (such as Bill Gates's income among the incomes of the citizens of the state of Washington). Many methods have been devised to identify outliers. We'll build a very simple one here.

Say we have retail sales data in a matrix `rs`. Each row of data is for a different store, and observations within a row are daily sales figures. As a simple (undoubtedly overly simple) approach, let's write code to identify the most

deviant observation for each store. We'll define that as the observation furthest from the median value for that store. Here's the code:

```
1  findols <- function(x) {
2     findol <- function(xrow) {
3         mdn <- median(xrow)
4         devs <- abs(xrow-mdn)
5         return(which.max(devs))
6     }
7     return(apply(x,1,findol))
8  }
```

Our call will be as follows:

```
findols(rs)
```

How will this work? First, we need a function to specify in our `apply()` call.

Since this function will be applied to each row of our sales matrix, our description implies that it needs to report the index of the most deviant observation in a given row. Our function `findol()` does that, in lines 4 and 5. (Note that we've defined one function within another here, a common practice if the inner function is short.) In the expression `xrow-mdn`, we are subtracting a number that is a one-element vector from a vector that generally will have a length greater than 1. Thus, recycling is used to extend `mdn` to conform with `xrow` before the subtraction.

Then in line 5, we use the R function `which.max()`. Instead of finding the maximum value in a vector, which the `max()` function does, `which.max()` tells us *where* that maximum value occurs—that is, the *index* where it occurs. This is just what we need.

Finally, in line 7, we ask R to apply `findol()` to each row of `x`, thus producing the indices of the most deviant observation in each row.

3.4 Adding and Deleting Matrix Rows and Columns

Technically, matrices are of fixed length and dimensions, so we cannot add or delete rows or columns. However, matrices can be *reassigned*, and thus we can achieve the same effect as if we had directly done additions or deletions.

3.4.1 Changing the Size of a Matrix

Recall how we reassign vectors to change their size:

```
> x
[1] 12  5 13 16  8
> x <- c(x,20)  # append 20
> x
[1] 12  5 13 16  8 20
```

```
> x <- c(x[1:3],20,x[4:6])  # insert 20
> x
[1] 12  5 13 20 16  8 20
> x <- x[-2:-4]  # delete elements 2 through 4
> x
[1] 12 16  8 20
```

In the first case, `x` is originally of length 5, which we extend to 6 via concatenation and then reassignment. We didn't literally change the length of `x` but instead created a new vector from `x` and then assigned `x` to that new vector.

NOTE *Reassignment occurs even when you don't see it, as you'll see in Chapter 14. For instance, even the innocuous-looking assignment `x[2] <- 12` is actually a reassignment.*

Analogous operations can be used to change the size of a matrix. For instance, the `rbind()` (*row bind*) and `cbind()` (*column bind*) functions let you add rows or columns to a matrix.

```
> one
[1] 1 1 1 1
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> cbind(one,z)
[1,]1 1 1 1
[2,]1 2 1 0
[3,]1 3 0 1
[4,]1 4 0 0
```

Here, `cbind()` creates a new matrix by combining a column of 1s with the columns of `z`. We choose to get a quick printout, but we could have assigned the result to `z` (or another variable), as follows:

```
z <- cbind(one,z)
```

Note, too, that we could have relied on recycling:

```
> cbind(1,z)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    1    0
[3,]    1    3    0    1
[4,]    1    4    0    0
```

Here, the 1 value was recycled into a vector of four 1 values.

You can also use the `rbind()` and `cbind()` functions as a quick way to create small matrices. Here's an example:

```
> q <- cbind(c(1,2),c(3,4))
> q
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Be careful with `rbind` and `cbin()`, though. Like creating a vector, creating a matrix is time consuming (matrices are vectors, after all). In the following code, `cbind()` creates a new matrix:

```
z <- cbind(one,z)
```

The new matrix happens to be reassigned to z; that is, we gave it the name z—the same name as the original matrix, which is now gone. But the point is that we did incur a time penalty in creating the matrix. If we did this repeatedly inside a loop, the cumulative penalty would be large.

So, if you are adding rows or columns one at a time within a loop, and the matrix will eventually become large, it's better to allocate a large matrix in the first place. It will be empty at first, but you fill in the rows or columns one at a time, rather than doing a time-consuming matrix memory allocation each time.

You can delete rows or columns by reassignment, too:

```
> m <- matrix(1:6,nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m <- m[c(1,3),]
> m
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

3.4.2 Extended Example: Finding the Closest Pair of Vertices in a Graph

Finding the distances between vertices on a graph is a common example used in computer science courses and is used in statistics/data sciences too. This kind of problem arises in some clustering algorithms, for instance, and in genomics applications.

Here, we'll look at the common example of finding distances between cities, as it is easier to describe than, say, finding distances between DNA strands.

Suppose we need a function that inputs a distance matrix, where the element in row `i`, column `j` gives the distance between city `i` and city `j` and outputs the minimum one-hop distance between cities and the pair of cities that achieves that minimum. Here's the code for the solution:

```
# returns the minimum value of d[i,j], i != j, and the row/col attaining
# that minimum, for square symmetric matrix d; no special policy on ties
mind <- function(d) {
   n <- nrow(d)
   # add a column to identify row number for apply()
   dd <- cbind(d,1:n)
   wmins <- apply(dd[-n,],1,imin)
   # wmins will be 2xn, 1st row being indices and 2nd being values
   i <- which.min(wmins[2,])
   j <- wmins[1,i]
   return(c(d[i,j],i,j))
}

# finds the location, value of the minimum in a row x
imin <- function(x) {
   lx <- length(x)
   i <- x[lx]  # original row number
   j <- which.min(x[(i+1):(lx-1)])
   k <- i+j
   return(c(k,x[k]))
}
```

And here's an example of putting our new function to use:

```
> q
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   12   13    8   20
[2,]   12    0   15   28   88
[3,]   13   15    0    6    9
[4,]    8   28    6    0   33
[5,]   20   88    9   33    0
> mind(q)
[1] 6 3 4
```

The minimum value was 6, located in row 3, column 4. As you can see, a call to `apply()` plays a prominent role.

Our task is fairly simple: We need to find the minimum nonzero element in the matrix. We find the minimum in each row—a single call to `apply()` accomplishes this for all the rows—and then find the smallest value

among those minima. But as you'll see, the code logic becomes rather intricate.

One key point is that the matrix is *symmetric*, because the distance from city `i` to city `j` is the same as from `j` to `i`. So in finding the minimum value in the i^{th} row, we need look at only elements `i+1`, `i+2`,..., *n*, where *n* is the number of rows and columns in the matrix. Note too that this means we can skip the last row of `d` in our call to `apply()` in line 7.

Since the matrix could be large—a thousand cities would mean a million entries in the matrix—we should exploit that symmetry and save work. That, however, presents a problem. In order to go through the basic computation, the function called by `apply()` needs to know the number of the row in the original matrix—knowledge that `apply()` does not provide to the function. So, in line 6, we augment the matrix with an extra column, consisting of the row numbers, so that the function called by `apply()` can take row numbers into account.

The function called by `apply()` is `imin()`, beginning in line 15, which finds the mininum in the row specified in the formal argument `x`. It returns not only the mininum in the given row but also the index at which that minimum occurs. When `imin()` is called on row 1 of our example matrix `q` above, the minimum value is 8, which occurs in index 4. For this latter purpose, the R function `which.min()`, used in line 18, is very handy.

Line 19 is noteworthy. Recall that due to the symmetry of the matrix, we skip the early part of each row, as is seen in the expression `(i+1):(1x-1)` in line 18. But that means that the call to `which.min()` in that line will return the minimum's index *relative* to the range `(i+1):(1x-1)`. In row 3 of our example matrix `q`, we would get an index of 1 instead of 4. Thus, we must adjust by adding `i`, which we do in line 19.

Finally, making proper use of the output of `apply()` here is a bit tricky. Think again of the example matrix `q` above. The call to `apply()` will return the matrix `wmins`:

$$\begin{pmatrix} 4 & 3 & 4 & 5 \\ 8 & 15 & 6 & 33 \end{pmatrix}$$

As noted in the comments, the second row of that matrix contains the upper-diagonal minima from the various rows of `d`, while the first row contains the indices of those values. For instance, the first column of `wmins` gives the information for the first row of `q`, reporting that the smallest value in that row is 8, occurring at index 4 of the row.

Thus line 9 will pick up the number `i` of the row containing the smallest value in the entire matrix, 6 in our `q` example. Line 10 will give us the position `j` in that row where the minimum occurs, 4 in the case of `q`. In other words, the overall minimum is in row `i` and column `j`, information that we then use in line 11.

Meanwhile, row 1 of `apply()`'s output shows the indices within those rows at which the row minima occur. That's great, because we can now find which other city was in the best pair. We know that city 3 is one of them, so we go

to entry 3 in row 1 of the output, finding 4. So, the pair of cities closest to each other is city 3 and city 4. Lines 9 and 10 then generalize this reasoning.

If the minimal element in our matrix is unique, there is an alternate approach that is far simpler:

```
minda <- function(d) {
   smallest <- min(d)
   ij <- which(d == smallest,arr.ind=TRUE)
   return(c(smallest,ij))
}
```

This works, but it does have some possible drawbacks. Here's the key line in this new code:

```
ij <- which(d == smallest,arr.ind=TRUE)
```

It determines the index of the element of `d` that achieves the minimum. The argument `arr.ind=TRUE` specifies that the returned index will be a matrix index—that is, a row and a column, rather than a single vector subscript. Without this argument, `d` would be treated as a vector.

As noted, this new code works only if the minimum is unique. If that is not the case, `which()` will return multiple row/column number pairs, contrary to our basic goal. And if we used the original code and `d` had multiple minimal elements, just one of them would be returned.

Another problem is performance. This new code is essentially making two (behind-the-scenes) loops through our matrix: one to compute `smallest` and the other in the call to `which()`. This will likely be slower than our original code.

Of these two approaches, you might choose the original code if execution speed is an issue or if there may be multiple minima, but otherwise opt for the alternate code; the simplicity of the latter will make the code easier to read and maintain.

3.5 More on the Vector/Matrix Distinction

At the beginning of the chapter, I said that a matrix is just a vector but with two additional attributes: the number of rows and the number of columns. Here, we'll take a closer look at the vector nature of matrices. Consider this example:

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

As z is still a vector, we can query its length:

```
> length(z)
[1] 8
```

But as a matrix, z is a bit more than a vector:

```
> class(z)
[1] "matrix"
> attributes(z)
$dim
[1] 4 2
```

In other words, there actually is a `matrix` *class*, in the object-oriented programming sense. As noted in Chapter 1, most of R consists of S3 classes, whose components are denoted by dollar signs. The `matrix` class has one attribute, named `dim`, which is a vector containing the numbers of rows and columns in the matrix. Classes will be covered in detail in Chapter 9.

You can also obtain `dim` via the `dim()` function:

```
> dim(z)
[1] 4 2
```

The numbers of rows and columns are obtainable individually via the `nrow()` and `ncol()` functions:

```
> nrow(z)
[1] 4
> ncol(z)
[1] 2
```

These just piggyback on `dim()`, as you can see by inspecting the code. Recall once again that objects can be printed in interactive mode by simply typing their names:

```
> nrow
function (x)
dim(x)[1]
```

These functions are useful when you are writing a general-purpose library function whose argument is a matrix. By being able to determine the number of rows and columns in your code, you alleviate the caller of the burden of supplying that information as two additional arguments. This is one of the benefits of object-oriented programming.

3.6 Avoiding Unintended Dimension Reduction

In the world of statistics, dimension reduction is a good thing, with many statistical procedures aimed to do it well. If we are working with, say, 10 variables and can reduce that number to 3 that still capture the essence of our data, we're happy.

However, in R, something else might merit the name *dimension reduction* that we may sometimes wish to avoid. Say we have a four-row matrix and extract a row from it:

```
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> r <- z[2,]
> r
[1] 2 6
```

This seems innocuous, but note the format in which R has displayed `r`. It's a vector format, not a matrix format. In other words, `r` is a vector of length 2, rather than a 1-by-2 matrix. We can confirm this in a couple of ways:

```
> attributes(z)
$dim
[1] 4 2
> attributes(r)
NULL
> str(z)
 int [1:4, 1:2] 1 2 3 4 5 6 7 8
> str(r)
 int [1:2] 2 6
```

Here, R informs us that `z` has row and column numbers, while `r` does not. Similarly, `str()` tells us that `z` has indices ranging in 1:4 and 1:2, for rows and columns, while `r`'s indices simply range in 1:2. No doubt about it—`r` is a vector, not a matrix.

This seems natural, but in many cases, it will cause trouble in programs that do a lot of matrix operations. You may find that your code works fine in general but fails in a special case. For instance, suppose that your code extracts a submatrix from a given matrix and then does some matrix operations on the submatrix. If the submatrix has only one row, R will make it a vector, which could ruin your computation.

Fortunately, R has a way to suppress this dimension reduction: the `drop` argument. Here's an example, using the matrix `z` from above:

```
> r <- z[2,, drop=FALSE]
> r
     [,1] [,2]
[1,]    2    6
> dim(r)
[1] 1 2
```

Now `r` is a 1-by-2 matrix, not a two-element vector.

For these reasons, you may find it useful to routinely include the `drop=FALSE` argument in all your matrix code.

Why can we speak of `drop` as an argument? Because that [is actually a function, just as is the case for operators like +. Consider the following code:

```
> z[3,2]
[1] 7
> "["(z,3,2)
[1] 7
```

If you have a vector that you wish to be treated as a matrix, you can use the `as.matrix()` function, as follows:

```
> u
[1] 1 2 3
> v <- as.matrix(u)
> attributes(u)
NULL
> attributes(v)
$dim
[1] 3 1
```

3.7 Naming Matrix Rows and Columns

The natural way to refer to rows and columns in a matrix is via the row and column numbers. However, you can also give names to these entities. Here's an example:

```
> z
     [,1] [,2]
[1,] 1 3
[2,] 2 4
> colnames(z)
NULL
> colnames(z) <- c("a","b")
```

```
> z
    a b
[1,] 1 3
[2,] 2 4
> colnames(z)
[1] "a" "b"
> z[,"a"]
[1] 1 2
```

As you see here, these names can then be used to reference specific columns. The function `rownames()` works similarly.

Naming rows and columns is usually less important when writing R code for general applications, but it can be useful when analyzing a specific data set.

3.8 Higher-Dimensional Arrays

In a statistical context, a typical matrix in R has rows corresponding to observations, say on various people, and columns corresponding to variables, such as weight and blood pressure. The matrix is then a two-dimensional data structure. But suppose we also have data taken at different times, one data point per person per variable per time. Time then becomes the third dimension, in addition to rows and columns. In R, such data sets are called *arrays*.

As a simple example, consider students and test scores. Say each test consists of two parts, so we record two scores for a student for each test. Now suppose that we have two tests, and to keep the example small, assume we have only three students. Here's the data for the first test:

```
> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   50
```

Student 1 had scores of 46 and 30 on the first test, student 2 scored 21 and 25, and so on. Here are the scores for the same students on the second test:

```
> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   50
```

Now let's put both tests into one data structure, which we'll name `tests`. We'll arrange it to have two "layers"—one layer per test—with three rows

and two columns within each layer. We'll store `firsttest` in the first layer and `secondtest` in the second.

In layer 1, there will be three rows for the three students' scores on the first test, with two columns per row for the two portions of a test. We use R's `array` function to create the data structure:

```
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
```

In the argument `dim=c(3,2,2)`, we are specifying two layers (this is the second 2), each consisting of three rows and two columns. This then becomes an attribute of the data structure:

```
> attributes(tests)
$dim
[1] 3 2 2
```

Each element of `tests` now has three subscripts, rather than two as in the matrix case. The first subscript corresponds to the first element in the `$dim` vector, the second subscript corresponds to the second element in the vector, and so on. For instance, the score on the second portion of test 1 for student 3 is retrieved as follows:

```
> tests[3,2,1]
[1] 48
```

R's print function for arrays displays the data layer by layer:

```
> tests
, , 1

     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

, , 2

     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

Just as we built our three-dimensional array by combining two matrices, we can build four-dimensional arrays by combining two or more three-dimensional arrays, and so on.

One of the most common uses of arrays is in calculating tables. See Section 6.3 for an example of a three-dimensional table.