3

MATRICES AND ARRAYS

By now, you have a solid handle on using vectors in R. A *matrix* is simply several vectors stored together. Whereas the size of a vector is described by its length, the size of a matrix is specified by a number of rows and a number of columns. You can also create higher-dimensional structures that are referred to as *arrays*. In this chapter, we'll begin by looking at how to work with matrices before increasing the dimension to form arrays.

3.1 Defining a Matrix

The matrix is an important mathematical construct, and it's essential to many statistical methods. You typically describe a matrix A as an $m \times n$ matrix; that is, A will have exactly m rows and n columns. This means A will have a total of mn entries, with each entry $a_{i,j}$ having a unique position given by its specific row ($i = 1, 2, \ldots, m$) and column ($j = 1, 2, \ldots, n$).

You can therefore express a matrix as follows:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$$

To create a matrix in R, use the aptly named `matrix` command, providing the entries of the matrix to the `data` argument as a vector:

```
R> A <- matrix(data=c(-3,2,893,0.17),nrow=2,ncol=2)
R> A
     [,1]   [,2]
[1,]   -3 893.00
[2,]    2   0.17
```

You must make sure that the length of this vector matches exactly with the number of desired rows (`nrow`) and columns (`ncol`). You can elect not to supply `nrow` and `ncol` when calling `matrix`, in which case R's default behavior is to return a single-column matrix of the entries in `data`. For example, `matrix(data=c(-3,2,893,0.17))` would be identical to `matrix(data=c(-3,2,893,0.17),nrow=4,ncol=1)`.

3.1.1 Filling Direction

It's important to be aware of how R fills up the matrix using the entries from `data`. Looking at the previous example, you can see that the 2×2 matrix `A` has been filled in a *column-by-column* fashion when reading the `data` entries from left to right. You can control how R fills in data using the argument `byrow`, as shown in the following examples:

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=FALSE)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Here, I've instructed R to provide a 2×3 matrix containing the digits 1 through 6. By using the optional argument `byrow` and setting it to `FALSE`, you explicitly tell R to fill this 2×3 structure in a column-wise fashion, by filling each column before moving to the next, reading the `data` argument vector from left to right. This is R's default handling of the `matrix` function, so if the `byrow` argument isn't supplied, the software will assume `byrow=FALSE`. Figure 3-1 illustrates this behavior.

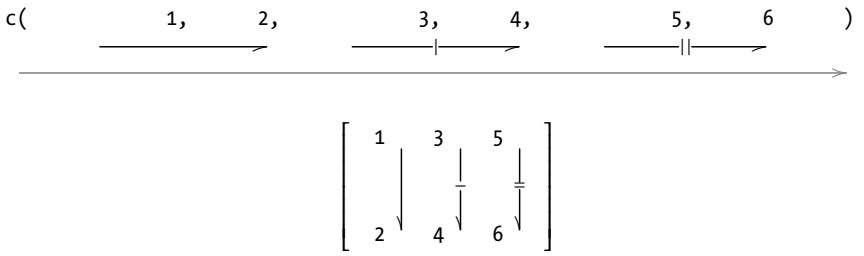

Figure 3-1: Filling a 2 × 3 matrix in a column-wise fashion with `byrow=FALSE` *(R default)*

Now, let's repeat the same line of code but set `byrow=TRUE`.

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

The resulting 2 × 3 structure has now been filled in a row-wise fashion, as shown in Figure 3-2.

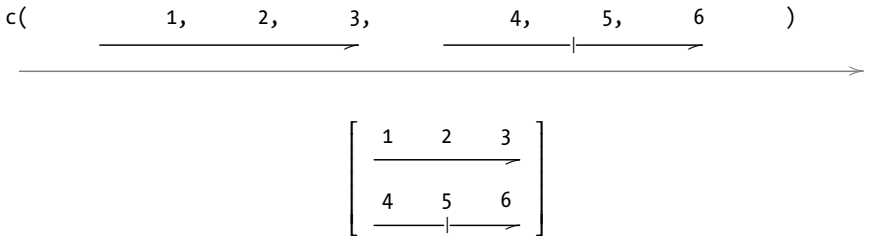

Figure 3-2: Filling a 2 × 3 matrix in a row-wise fashion with `byrow=TRUE`

3.1.2 Row and Column Bindings

If you have multiple vectors of equal length, you can quickly build a matrix by binding together these vectors using the built-in R functions, `rbind` and `cbind`. You can either treat each vector as a row (by using the command `rbind`) or treat each vector as a column (using the command `cbind`). Say you have the two vectors `1:3` and `4:6`. You can reconstruct the 2 × 3 matrix in Figure 3-2 using `rbind` as follows:

```
R> rbind(1:3,4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Here, rbind has bound together the vectors as two rows of a matrix, with the top-to-bottom order of the rows matching the order of the vectors supplied to rbind. The same matrix could be constructed as follows, using cbind:

```
R> cbind(c(1,4),c(2,5),c(3,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Here, you have three vectors each of length 2. You use cbind to glue together these three vectors in the order they were supplied, and each vector becomes a column of the resulting matrix.

3.1.3 Matrix Dimensions

Another useful function, dim, provides the dimensions of a matrix stored in your workspace.

```
R> mymat <- rbind(c(1,3,4),5:3,c(100,20,90),11:13)
R> mymat
     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    5    4    3
[3,]  100   20   90
[4,]   11   12   13

R> dim(mymat)
[1] 4 3
R> nrow(mymat)
[1] 4
R> ncol(mymat)
[1] 3
R> dim(mymat)[2]
[1] 3
```

Having defined a matrix mymat using rbind, you can confirm its dimensions with dim, which returns a vector of length 2; dim always supplies the number of rows first, followed by the number of columns. You can also use two related functions: nrow (which provides the number of rows only) and ncol (which provides the number of columns only). In the last command shown, you use dim and your knowledge of vector subsetting to extract the same result that ncol would give you.

3.2 Subsetting

Extracting and subsetting elements from matrices in R is much like extracting elements from vectors. The only complication is that you now have an additional dimension. Element extraction still uses the square-bracket

operator, but now it must be performed with both a row *and* a column position, given strictly in the order of [*row*,*column*]. Let's start by creating a 3×3 matrix, which I'll use for the examples in this section.

```
R> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

To tell R to "look at the third row of `A` and give me the element from the second column," you execute the following:

```
R> A[3,2]
[1] 105.5
```

As expected, you're given the element at position `[3,2]`.

3.2.1 Row, Column, and Diagonal Extractions

To extract an entire row or column from a matrix, you simply specify the desired row or column number and leave the other value blank. It's important to note that *you must still include* the comma that separates the row and column numbers—this is how R distinguishes between a request for a row and a request for a column. The following returns the second column of `A`:

```
R> A[,2]
[1]  91.0   0.1 105.5
```

The following examines the first row:

```
R> A[1,]
[1]  0.3 91.0 -4.2
```

Note that whenever an extraction (or deletion, covered in a moment) results in a single value, single row, or single column, R will always return stand-alone vectors comprised of the requested values. You can also perform more complicated extractions, for example requesting whole rows or columns, or multiples rows or columns, where the result must be returned as a new matrix of the appropriate dimensions. Consider the following subsets:

```
R> A[2:3,]
     [,1]  [,2] [,3]
[1,]  4.5   0.1  8.2
[2,] 55.3 105.5 27.9

R> A[,c(3,1)]
     [,1] [,2]
```

```
[1,] -4.2  0.3
[2,]  8.2  4.5
[3,] 27.9 55.3

R> A[c(3,1),2:3]
      [,1] [,2]
[1,] 105.5 27.9
[2,]  91.0 -4.2
```

The first command returns the second and third rows of `A`, and the second command returns the third and first columns of `A`. The last command accesses the third and first rows of `A`, in that order, and from those rows it returns the second and third column elements.

You can also identify the values along the diagonal of a square matrix (that is, a matrix with an equal number of rows and columns) using the `diag` command.

```
R> diag(x=A)
[1]  0.3  0.1 27.9
```

This returns a vector with the elements along the diagonal of `A`, starting at `A[1,1]`.

3.2.2 Omitting and Overwriting

To delete or omit elements from a matrix, you again use square brackets, but this time with negative indexes. The following provides `A` without its second column:

```
R> A[,-2]
     [,1] [,2]
[1,]  0.3 -4.2
[2,]  4.5  8.2
[3,] 55.3 27.9
```

The following removes the first row from `A` and retrieves the third and second column values, in that order, from the remaining two rows:

```
R> A[-1,3:2]
     [,1]  [,2]
[1,]  8.2   0.1
[2,] 27.9 105.5
```

The following produces `A` without its first row and second column:

```
R> A[-1,-2]
     [,1] [,2]
[1,]  4.5  8.2
[2,] 55.3 27.9
```

Lastly, this deletes the first row and then deletes the second and third columns from the result:

```
R> A[-1,-c(2,3)]
[1]  4.5 55.3
```

Note that this final operation leaves you with only the last two elements of the first column of `A`, so this result is returned as a stand-alone vector rather than a matrix.

To overwrite particular elements, or entire rows or columns, you identify the elements to be replaced and then assign the new values, as you did with vectors in Section 2.3.3. The new elements can be a single value, a vector of the same length as the number of elements to be replaced, or a vector whose length evenly divides the number of elements to be replaced. To illustrate this, let's first create a copy of `A` and call it `B`.

```
R> B <- A
R> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

The following overwrites the second row of `B` with the sequence `1`, `2`, and `3`:

```
R> B[2,] <- 1:3
R> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  1.0   2.0  3.0
[3,] 55.3 105.5 27.9
```

The following overwrites the second column elements of the first and third rows with `900`:

```
R> B[c(1,3),2] <- 900
R> B
     [,1] [,2] [,3]
[1,]  0.3  900 -4.2
[2,]  1.0    2  3.0
[3,] 55.3  900 27.9
```

Next, you replace the third column of `B` with the values in the third *row* of `B`.

```
R> B[,3] <- B[3,]
R> B
     [,1] [,2]  [,3]
```

```
[1,]  0.3  900  55.3
[2,]  1.0    2 900.0
[3,] 55.3  900  27.9
```

To try R's vector recycling, let's now overwrite the first and third column elements of rows 1 and 3 (a total of four elements) with the two values `-7` and `7`.

```
R> B[c(1,3),c(1,3)] <- c(-7,7)
R> B
     [,1] [,2] [,3]
[1,]   -7  900   -7
[2,]    1    2  900
[3,]    7  900    7
```

The vector of length 2 has replaced the four elements *in a column-wise fashion*. The replacement vector `c(-7,7)` overwrites the elements at positions (1,1) and (3,1), in that order, and is then repeated to overwrite (1,3) and (3,3), in that order.

To highlight the role of index order on matrix element replacement, consider the following example:

```
R> B[c(1,3),2:1] <- c(65,-65,88,-88)
R> B
     [,1] [,2] [,3]
[1,]   88   65   -7
[2,]    1    2  900
[3,]  -88  -65    7
```

The four values in the replacement vector have overwritten the four specified elements, again in a column-wise fashion. In this case, because I specified the first and second columns in reverse order, the overwriting proceeded accordingly, filling the second column before moving to the first. Position (1,2) is matched with `65`, followed by (3,2) with `-65`; then (1,1) becomes `88`, and (3,1) becomes `-88`.

If you just want to replace the diagonal of a square matrix, you can avoid explicit indexes and directly overwrite the values using the `diag` command.

```
R> diag(x=B) <- rep(x=0,times=3)
R> B
     [,1] [,2] [,3]
[1,]    0   65   -7
[2,]    1    0  900
[3,]  -88  -65    0
```

Exercise 3.1

a. Construct and store a 4×2 matrix that's filled row-wise with the values 4.3, 3.1, 8.2, 8.2, 3.2, 0.9, 1.6, and 6.5, in that order.

b. Confirm the dimensions of the matrix from (a) are 3×2 if you remove any one row.

c. Overwrite the second column of the matrix from (a) with that same column sorted from smallest to largest.

d. What does R return if you delete the fourth row and the first column from (c)? Use `matrix` to ensure the result is a single-column matrix, rather than a vector.

e. Store the bottom four elements of (c) as a new 2×2 matrix.

f. Overwrite, in this order, the elements of (c) at positions $(4,2)$, $(1,2)$, $(4,1)$, and $(1,1)$ with $-\frac{1}{2}$ of the two values on the diagonal of (e).

3.3 Matrix Operations and Algebra

You can think of matrices in R from two perspectives. First, you can use these structures purely as a computational tool in programming to store and operate on results, as you've seen so far. Alternatively, you can use matrices for their mathematical properties in relevant calculations, such as the use of matrix multiplication for expressing regression model equations. This distinction is important because the mathematical behavior of matrices is not always the same as the more generic data handling behavior. Here I'll briefly describe some special matrices, as well as some of the most common mathematical operations involving matrices, and the corresponding functionality in R. If the mathematical behavior of matrices isn't of interest to you, you can skip this section for now and refer to it later as needed.

3.3.1 Matrix Transpose

For any $m \times n$ matrix A, its *transpose*, $A^\top$, is the $n \times m$ matrix obtained by writing either its columns as rows or its rows as columns.

Here's an example:

$$\text{If } A = \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix}, \text{ then } A^\top = \begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix}.$$

In R, the transpose of a matrix is found with the function `t`. Let's create a new matrix and then transpose it.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> A
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
R> t(A)
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
```

If you "transpose the transpose" of A, you'll recover the original matrix.

```
R> t(t(A))
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
```

3.3.2 Identity Matrix

The *identity matrix* written as I_m is a particular kind of matrix used in mathematics. It's a square $m \times m$ matrix with ones on the diagonal and zeros elsewhere.

Here's an example:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can create an identity matrix of any dimension using the standard `matrix` function, but there's a quicker approach using `diag`. Earlier, I used `diag` on an existing matrix to extract or overwrite its diagonal elements. You can also use it as follows:

```
R> A <- diag(x=3)
R> A
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Here you see `diag` can be used to easily produce an identity matrix. To clarify, the behavior of `diag` depends on what you supply to it as its argument `x`. If, as earlier, `x` is a matrix, `diag` will retrieve the diagonal elements of the matrix. If `x` is a single positive integer, as is the case here, then `diag` will produce the identity matrix of the corresponding dimension. You can find more uses of `diag` on its help page.

3.3.3 Scalar Multiple of a Matrix

A scalar value is just a single, univariate value. Multiplication of any matrix A by a scalar value a results in a matrix in which every individual element is multiplied by a.

Here's an example:

$$2 \times \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 10 & 4 \\ 12 & 2 & 8 \end{bmatrix}$$

R will perform this multiplication in an element-wise manner, as you might expect. Scalar multiplication of a matrix is carried out using the standard arithmetic $*$ operator.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> a <- 2
R> a*A
     [,1] [,2] [,3]
[1,]    4   10    4
[2,]   12    2    8
```

3.3.4 Matrix Addition and Subtraction

Addition or subtraction of two matrices of equal size is also performed in an element-wise fashion. Corresponding elements are added or subtracted from one another, depending on the operation.

Here's an example:

$$\begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 8.1 \\ 3 & 8.2 \\ 6 & -9.8 \end{bmatrix} = \begin{bmatrix} 4 & -2.1 \\ 2 & -7.2 \\ -4 & 13.8 \end{bmatrix}$$

You can add or subtract any two equally sized matrices with the standard + and - symbols.

```
R> A <- cbind(c(2,5,2),c(6,1,4))
R> A
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
R> B <- cbind(c(-2,3,6),c(8.1,8.2,-9.8))
R> B
     [,1] [,2]
[1,]   -2  8.1
[2,]    3  8.2
[3,]    6 -9.8
R> A-B
```

```
     [,1] [,2]
[1,]    4 -2.1
[2,]    2 -7.2
[3,]   -4 13.8
```

3.3.5 Matrix Multiplication

In order to *multiply* two matrices A and B of size $m \times n$ and $p \times q$, it must be true that $n = p$. The resulting matrix $A \cdot B$ will have the size $m \times q$. The elements of the product are computed in a row-by-column fashion, where the value at position $(AB)_{i,j}$ is computed by element-wise multiplication of the entries in row i of A by the entries in column j of B, summing the result.

Here's an example:

$$\begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & -3 \\ -1 & 1 \\ 1 & 5 \end{bmatrix}$$
$$= \begin{bmatrix} 2_{\times}3 + 5_{\times}(-1) + 2_{\times}1 & 2_{\times}(-3) + 5_{\times}(1) + 2_{\times}5 \\ 6_{\times}3 + 1_{\times}(-1) + 4_{\times}1 & 6_{\times}(-3) + 1_{\times}(1) + 4_{\times}5 \end{bmatrix}$$
$$= \begin{bmatrix} 3 & 9 \\ 21 & 3 \end{bmatrix}$$

Note that, in general, multiplication of appropriately sized matrices (denoted, say, with C and D) is not commutative; that is, $CD \neq DC$.

Unlike addition, subtraction, and scalar multiplication, matrix multiplication is not a simple element-wise calculation, and the standard $*$ operator cannot be used. Instead, you must use R's matrix product operator, written with percent symbols as `%*%`. Before you try this operator, let's first store the two example matrices and check to make sure the number of columns in the first matrix matches the number of rows in the second matrix using `dim`.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> dim(A)
[1] 2 3
R> B <- cbind(c(3,-1,1),c(-3,1,5))
R> dim(B)
[1] 3 2
```

This confirms the two matrices are compatible for multiplication, so you can proceed.

```
R> A%*%B
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

You can show that matrix multiplication is noncommutative using the same two matrices. Switching the order of multiplication gives you an entirely different result.

```
R> B%*%A
     [,1] [,2] [,3]
[1,]  -12   12   -6
[2,]    4   -4    2
[3,]   32   10   22
```

3.3.6 Matrix Inversion

Some square matrices can be *inverted*. The inverse of a matrix A is denoted A^{-1}. An invertible matrix satisfies the following equation:

$$AA^{-1} = I_m$$

Here's an example of a matrix and its inverse:

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & -0.5 \\ -2 & 1.5 \end{bmatrix}$$

Matrices that are not invertible are referred to as *singular*. Inverting a matrix is often necessary when solving equations with matrices and has important practical ramifications. There are several different approaches to matrix inversion, and these calculations can become extremely computationally expensive as you increase the size of a matrix. We won't go into too much detail here, but if you're interested, see Golub and Van Loan (1989) for formal discussions.

For now, I'll just show you the R function `solve` as one option for inverting a matrix.

```
R> A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
R> A
     [,1] [,2]
[1,]    3    1
[2,]    4    2
R> solve(A)
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

You can also verify that the product of these two matrices (using matrix multiplication rules) results in the 2×2 identity matrix.

```
R> A%*%solve(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Exercise 3.2

a. Calculate the following:

$$\frac{2}{7}\left(\begin{bmatrix}1 & 2\\ 2 & 4\\ 7 & 6\end{bmatrix} - \begin{bmatrix}10 & 20\\ 30 & 40\\ 50 & 60\end{bmatrix}\right)$$

b. Store these two matrices:

$$A = \begin{bmatrix}1\\ 2\\ 7\end{bmatrix} \qquad B = \begin{bmatrix}3\\ 4\\ 8\end{bmatrix}$$

Which of the following multiplications are possible? For those that are, compute the result.

i. $A \cdot B$
ii. $A^\top \cdot B$
iii. $B^\top \cdot (A \cdot A^\top)$
iv. $(A \cdot A^\top) \cdot B^\top$
v. $[(B \cdot B^\top) + (A \cdot A^\top) - 100I_3]^{-1}$

c. For

$$A = \begin{bmatrix}2 & 0 & 0 & 0\\ 0 & 3 & 0 & 0\\ 0 & 0 & 5 & 0\\ 0 & 0 & 0 & -1\end{bmatrix},$$

confirm that $A^{-1} \cdot A - I_4$ provides a 4×4 matrix of zeros.

3.4 Multidimensional Arrays

Just as a matrix (a "rectangle" of elements) is the result of increasing the dimension of a vector (a "line" of elements), the dimension of a matrix can be increased to get more complex data structures. In R, vectors and matrices can be considered special cases of the more general *array*, which is how I'll refer to these types of structures when they have more than two dimensions.

So, what's the next step up from a matrix? Well, just as a matrix is considered to be a collection of vectors of equal length, a three-dimensional array can be considered to be a collection of equally dimensioned matrices,

providing you with a rectangular prism of elements. You still have a fixed number of rows and a fixed number of columns, as well as a new third dimension called a *layer*. Figure 3-3 illustrates a three-row, four-column, two-layer ($3 \times 4 \times 2$) array.

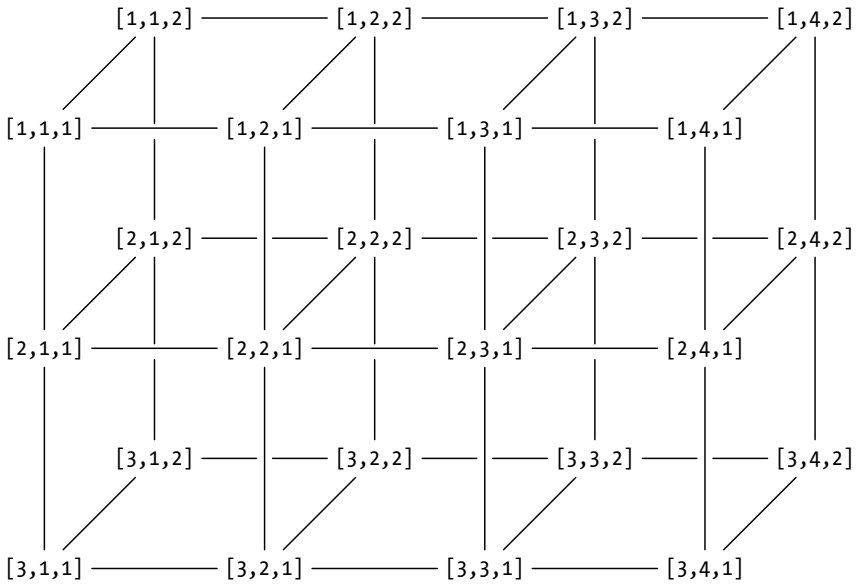

Figure 3-3: A conceptual diagram of a 3 × 4 × 2 array. The index of each element is given at the corresponding position. These indexes are provided in the strict order of `[row,column,layer]`.

3.4.1 Definition

To create these data structures in R, use the `array` function and specify the individual elements in the `data` argument as a vector. Then specify size in the `dim` argument as another vector with a length corresponding to the number of dimensions. Note that `array` fills the entries of each layer with the elements in `data` in a strict column-wise fashion, starting with the first layer. Consider the following example:

```
R> AR <- array(data=1:24,dim=c(3,4,2))
R> AR
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2
```

```
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

This gives you an array of the same size as in Figure 3-3—each of the two layers constitutes a 3 × 4 matrix. In this example, note the order of the dimensions supplied to `dim`: `c(`*`rows`*`,`*`columns`*`,`*`layers`*`)`. Just like a single matrix, the product of the dimension sizes of an array will yield the total number of elements. As you increase the dimension further, the `dim` vector must be extended accordingly. For example, a four-dimensional array is the next step up and can be thought of as *blocks* of three-dimensional arrays. Suppose you had a four-dimensional array comprised of three copies of `AR`, the three-dimensional array just defined. This new array can be stored in R as follows (once again, the array is filled column-wise):

```
R> BR <- array(data=rep(1:24,times=3),dim=c(3,4,2,3))
R> BR
, , 1, 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2, 1

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 1, 2

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2, 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 1, 3
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2, 3

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

With `BR` you now have three copies of `AR`. Each of these copies is split into its two layers so R can print the object to the screen. As before, the rows are indexed by the first digit, the columns by the second digit, and the layers by the third digit. The new fourth digit indexes the blocks.

3.4.2 Subsets, Extractions, and Replacements

Even though high-dimensional objects can be difficult to conceptualize, R indexes them consistently. This makes extracting elements from these structures straightforward now that you know how to subset matrices—you just have to keep using commas in the square brackets as separators of the dimensions being accessed. This is highlighted in the examples that follow.

Suppose you want the second row of the second layer of the previously created array `AR`. You just enter these exact dimensional locations of `AR` in square brackets.

```
R> AR[2,,2]
[1] 14 17 20 23
```

The desired elements have been extracted as a vector of length 4. If you want specific elements from this vector, say the third and first, in that order, you can call the following:

```
R> AR[2,c(3,1),2]
[1] 20 14
```

Again, this literal method of subsetting makes dealing with even high-dimensional objects in R manageable.

An extraction that results in multiple vectors will be presented as columns in the returned matrix. For example, to extract the first rows of both layers of `AR`, you enter this:

```
R> AR[1,,]
     [,1] [,2]
[1,]    1   13
```

```
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

The returned object has the first rows of each of the two matrix layers. However, it has returned each of these vectors as a *column* of the single returned matrix. As this example shows, when multiple vectors are extracted from an array, they will be returned as columns by default. This means extracted rows will not necessarily be returned as rows.

Turning to the object `BR`, the following gives you the single element of the second row and first column of the matrix in the first layer of the three-dimensional array located in the third block.

```
R> BR[2,1,1,3]
[1] 2
```

Again, you just need to look at the position of the index in the square brackets to know which values you are asking R to return from the array. The following examples highlight this:

```
R> BR[1,,,1]
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

This returns all the values in the first row of the first block. Since I left the column and layer indexes blank in this subset `[1,,,1]`, the command has returned values for all four columns and both layers in that block of `BR`.

Next, the following line returns all the values in the second layer of the array `BR`, composed of three matrices:

```
R> BR[,,2,]
, , 1

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```
, , 3

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

This last example highlights a feature noted earlier, where multiple vectors from `AR` were returned as a matrix. Broadly speaking, if you have an extraction that results in multiple *d*-dimensional arrays, the result will be an array of the next-highest dimension, $d + 1$. In the last example, you extracted multiple (two-dimensional) matrices, and they were returned as a three-dimensional array. This is demonstrated again in the next example:

```
R> BR[3:2,4,,]
, , 1

     [,1] [,2]
[1,]   12   24
[2,]   11   23

, , 2

     [,1] [,2]
[1,]   12   24
[2,]   11   23

, , 3

     [,1] [,2]
[1,]   12   24
[2,]   11   23
```

This extracts the elements at rows 3 and 2 (in that order), column 4, for all layers and for all array blocks. Consider the following final example:

```
R> BR[2,,1,]
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    5    5    5
[3,]    8    8    8
[4,]   11   11   11
```

Here you've asked R to return the entire second rows of the first layers of all the arrays stored in `BR`.

Deleting and overwriting elements in high-dimensional arrays follows the same rules as for stand-alone vectors and matrices. You specify the

dimension positions the same way, using negative indexes (for deletion) or using the assignment operator for overwriting.

You can use the `array` function to create one-dimensional arrays (vectors) and two-dimensional arrays (matrices) should you want to (by setting the `dim` argument to be of length 1 or 2, respectively). Note, though, that vectors in particular may be treated differently by some functions if created with `array` instead of `c` (see the help file `?array` for technical details). For this reason, and to make large sections of code more readable, it's more conventional in R programming to use the specific vector- and matrix-creation functions `c` and `matrix`.

Exercise 3.3

a. Create and store a three-dimensional array with six layers of a 4×2 matrix, filled with a decreasing sequence of values between 4.8 and 0.1 of the appropriate length.

b. Extract and store as a new object the fourth- and first-row elements, in that order, of the second column only of all layers of (a).

c. Use a fourfold repetition of the second row of the matrix formed in (b) to fill a new array of dimensions $2 \times 2 \times 2 \times 3$.

d. Create a new array comprised of the results of deleting the sixth layer of (a).

e. Overwrite the second and fourth row elements of the second column of layers 1, 3, and 5 of (d) with −99.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`matrix`	Create a matrix	Section 3.1, p. 40
`rbind`	Create a matrix (bind rows)	Section 3.1.2, p. 41
`cbind`	Create a matrix (bind columns)	Section 3.1.2, p. 42
`dim`	Get matrix dimensions	Section 3.1.3, p. 42
`nrow`	Get number of rows	Section 3.1.3, p. 42
`ncol`	Get number of columns	Section 3.1.3, p. 42
`[ , ]`	Matrix/array subsetting	Section 3.2, p. 43
`diag`	Diagonal elements/identity matrix	Section 3.2.1, p. 44
`t`	Matrix transpose	Section 3.3.1, p. 47
`*`	Scalar matrix multiple	Section 3.3.3, p. 49
`+, -`	Matrix addition/subtraction	Section 3.3.4, p. 49
`%*%`	Matrix multiplication	Section 3.3.5, p. 50
`solve`	Matrix inversion	Section 3.3.6, p. 51
`array`	Create an array	Section 3.4.1, p. 53