CHAPTER 6

Matrices and Arrays

THE OBJECTIVES OF THIS CHAPTER ARE TO:

- Introduce you to ways of creating and manipulating matrices.
- Introduce you to matrix operations.
- Introduce you to character strings and facilities for handling them.
- Demonstrate some uses of matrices by examples in:
  - Population dynamics.
  - Markov processes.
  - Linear equations.
- Introduce MATLAB's sparse matrix facilities.

As we have already seen, the name MATLAB stands for MATrix LABoratory, because the MATLAB system is designed specially to work with data arranged in the form of *matrices* (2-D arrays). The word 'matrix' has two distinct meanings in this chapter:

1. An arrangement of data in rows and columns, e.g., a table.
2. A mathematical object, for which particular mathematical operations are defined, e.g., 'matrix' multiplication.

We begin this chapter (in Sections 6.6.1–6.6.3) by looking at matrices in the first sense, summarizing and extending what we learned about them in Chapter 2. We then go on to look briefly at the mathematical operations on matrices. Subsequently, we examine how these operations can be applied in a number of widely differing areas, such as systems of linear equations, population dynamics, and Markov processes.

6.1 MATRICES

6.1.1 A concrete example

A ready-mix concrete company has three factories (S1, S2 and S3) which must supply three building sites (D1, D2 and D3). The costs, in some suitable cur-

CONTENTS

Essential MATLAB for Engineers and Scientists. DOI:10.1016/B978-0-08-100877-5.00007-4

rency, of transporting a load of concrete from any factory to any site are given by the following cost table:

	D1	D2	D3
S1	3	12	10
S2	17	18	35
S3	7	10	24

The factories can supply 4, 12 and 8 loads per day, respectively, and the sites require 10, 9 and 5 loads per day, respectively. The real problem is to find the cheapest way to satisfy the demands at the sites, but we are not considering that here.

Suppose the factory manager proposes the following transportation scheme (each entry represents the number of loads of concrete to be transported along that particular route):

	D1	D2	D3
S1	4	0	0
S2	6	6	0
S3	0	3	5

This sort of scheme is called a *solution* to the transportation problem. The cost table (and the solution) can then be represented by tables **C** and **X**, say, where c_{ij} is the entry in row i and column j of the cost table, with a similar convention for **X**.

To compute the cost of the above solution, each entry in the solution table must be multiplied by the corresponding entry in the cost table. (This operation is not to be confused with the mathematical operation of matrix multiplication, which is discussed later.) We therefore want to calculate

$$3 \times 4 + 12 \times 0 + \cdots + 24 \times 5.$$

To do this calculation in MATLAB, enter **C** and **X** as matrices from the command line, with a semi-colon at the end of each row:

```
c = [3 12 10; 17 18 35; 7 10 24];
x = [4 0 0; 6 6 0; 0 3 5];
```

and then find the *array* product of `c` and `x`:

```
total = c .* x
```

which gives

```
 12     0     0
102   108     0
  0    30   120
```

The command

```
sum(total)
```

then returns a vector where each element is the sum of each column of `total`:

```
114   138   120
```

Summing this in turn, i.e., `sum(sum( total ))`, gives the final answer of 372.

6.1.2 Creating matrices

As we saw above, use a semi-colon to indicate the end of a row when entering a matrix. Bigger matrices can be constructed from smaller ones, e.g., the statements

```
a = [1 2; 3 4];
x = [5 6];
a = [a; x]
```

result in

```
a =
     1     2
     3     4
     5     6
```

Instead of a semi-colon, you can use a line-feed (**Enter**) to indicate the end of a row.

6.1.3 Subscripts

Individual elements of a matrix are referenced with two subscripts, the first for the row, and the second for the column, e.g., `a(3,2)` for `a` above returns 6.

Alternatively, and less commonly, a single subscript may be used. In this case you can think of the matrix as being 'unwound' column by column. So `a(5)` for the above example returns 4.

If you refer to a subscript which is out of range, e.g., `a(3,3)` for `a` above, you will get an error message. However, if you assign a value to an element with a subscript which is out of range the matrix is enlarged to accommodate the new element, for example, the assignment

```
a(3,3) = 7
```

will add a third column to `a` with 0s everywhere except at `a(3,3)`.

6.1.4 Transpose

The statements

```
a = [1:3; 4:6]
b = a'
```

result in

```
a =
     1     2     3
     4     5     6

b =
     1     4
     2     5
     3     6
```

The transpose operator `'` (apostrophe) turns rows into columns and vice versa.

6.1.5 The colon operator

- The colon operator is extremely powerful, and provides for very efficient ways of handling matrices, e.g., if `a` is the matrix

```
a =
     1     2     3
     4     5     6
     7     8     9
```

the statement

```
a(2:3,1:2)
```

results in

```
4     5
7     8
```

(returns second and third rows, first and second columns), the statement

```
a(3,:)
```

results in

```
7     8     9
```

(returns third row), and the statement

```
a(1:2,2:3) = ones(2)
```

results in

```
a =
     1     1     1
     4     1     1
     7     8     9
```

(replaces the 2-by-2 submatrix composed of the first and second row and the second and third column with a square matrix of 1s).
Essentially, what is happening in the above examples is that the colon operator is being used to create vector subscripts. However, a colon by itself in place of a subscript denotes *all* the elements of the corresponding row or column. So `a(3,:)` means all elements in the third row.
This feature may be used, for example, to construct tables. Suppose we want a table `trig` of the sines and cosines of the angles 0° to 180° in steps of 30°. The following statements achieve this:

```
x = [0:30:180]';
trig(:,1) = x;
trig(:,2) = sin(pi/180*x);
trig(:,3) = cos(pi/180*x);
```

- You can use vector subscripts to get more complicated effects, e.g.,

```
a(:,[1 3]) = b(:,[4 2])
```

replaces the first and third columns of `a` by the fourth and second columns of `b` (`a` and `b` must have the same number of rows).
- The colon operator is ideal for the sort of row operations performed in Gauss reduction (a technique of numerical mathematics), for example, if `a` is the matrix

```
a =
     1    -1     2
     2     1    -1
     3     0     1
```

the statement

```
a(2,:) = a(2,:) - a(2,1)*a(1,:)
```

subtracts the first row multiplied by the first element in the second row from the second row, resulting in

```
a =
     1    -1     2
     0     3    -5
     3     0     1
```

(the idea being to get a zero immediately underneath `a(1,1)`).

- The keyword `end` refers to the *last* row or column of an array, for example, if `r` is a row vector, the statement

```
sum(r(3:end))
```

returns the sum of all the elements of `r` from the third one to the last one.

- The colon operator may also be used as a single subscript, in which case it behaves differently if it is on the right-hand or left-hand side of an assignment. On the right-hand side of an assignment, `a(:)` gives all the elements of `a` strung out *by columns* in one long column vector, e.g., if

```
a =
     1     2
     3     4
```

the statement

```
b = a(:)
```

results in

```
b =
     1
     3
     2
     4
```

However, on the left-hand side of an assignment, `a(:)` *reshapes* a matrix. `a` must already exist. Then `a(:)` denotes a matrix with the same dimensions (shape) as `a`, but with new contents taken from the right-hand side. In other words, the matrix on the right-hand side is reshaped into the shape of `a` on the left-hand side. Some examples may help. If

```
b =
     1     2     3
     4     5     6
```

and

```
a =
     0     0
     0     0
     0     0
```

the statement

```
a(:) = b
```

results in

```
a =
     1     5
     4     3
     2     6
```

(the contents of `b` are strung out into one long column and then fed into `a` by columns). As another example, the statement

```
a(:) = 1:6
```

(with `a` as above) results in

```
a =
     1     4
     2     5
     3     6
```

Reshaping can also be done with the `reshape` function. See `help`.

- Whether the colon operator appears on the right- or left-hand side of an assignment, the matrices or submatrices on each side of the assignment have the same shape (except in the special case below).
- As a special case, a single colon subscript may be used to replace *all* the elements of a matrix with a scalar, e.g.,

```
a(:) = -1
```

6.1.6 Duplicating rows and columns: tiling

Sometimes it is useful to generate a matrix where all the rows or columns are the same. This can be done with the `repmat` function as follows. If `a` is the row vector

```
a =
     1     2     3
```

the statement

```
repmat(a, [3 1])
```

results in

```
ans =
     1     2     3
     1     2     3
     1     2     3
```

In `help`'s inimitable phraseology, this statement produces a 3-by-1 'tiling' of copies of `a`. You can think of `a` as a 'strip' of three tiles stuck to self-adhesive backing. The above statement tiles a floor with three rows and one column of such a strip of tiles.

There is an alternative syntax for `repmat`:

```
repmat(a, 3, 1)
```

An interesting example of this process appears at the end of this section in the loan repayment problem.

6.1.7 Deleting rows and columns

Use the colon operator and the empty array to delete entire rows or columns, for example,

```
a(:,2) = [ ]
```

deletes the second column of `a`.

You can't delete a single element from a matrix while keeping it a matrix, so a statement like

```
a(1,2) = [ ]
```

results in an error. However, using the single subscript notation you can delete a sequence of elements from a matrix and reshape the remaining elements into a row vector, for example, if

```
a =
     1     2     3
```

```
     4     5     6
     7     8     9
```

the statement

```
a(2:2:6) = [ ]
```

results in

```
a =
     1     7     5     3     6     9
```

Get it? (First unwind `a` by columns, then remove elements 2, 4 and 6.)

You can use logical vectors to extract a selection of rows or columns from a matrix, for example, if `a` is the original 3-by-3 matrix defined above, the statement

```
a(:, logical([1 0 1]))
```

results in

```
ans =
     1     3
     4     6
     7     9
```

(first and third columns extracted). The same effect is achieved with

```
a(:, [1 3])
```

6.1.8 Elementary matrices

There is a group of functions to generate 'elementary' matrices which are used in a number of applications. See `help elmat`.

For example, the functions `zeros`, `ones` and `rand` generate matrices of 1s, 0s and random numbers, respectively. With a single argument n, they generate $n \times n$ (square) matrices. With two arguments n and m they generate $n \times m$ matrices. (For very large matrices `repmat` is usually faster than `ones` and `zeros`.)

The function `eye(n)` generates an $n \times n$ *identity* matrix, i.e., a matrix with 1s on the main 'diagonal', and 0s everywhere else, e.g., the statement

```
eye(3)
```

results in

```
ans =
     1     0     0
     0     1     0
     0     0     1
```

(The original version of MATLAB could not use the more obvious name `I` for the identity since it did not distinguish between upper- and lowercase letters and `i` was the natural choice for the imaginary unit number.)

As an example, `eye` may be used to construct a *tridiagonal* matrix as follows. The statements

```
a = 2 * eye(5);
a(1:4, 2:5) = a(1:4, 2:5) - eye(4);
a(2:5, 1:4) = a(2:5, 1:4) - eye(4)
```

result in

```
a =
     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     2
```

Incidentally, if you work with large tridiagonal matrices, you should have a look at the sparse matrix facilities available in MATLAB via the help browser.

6.1.9 Specialized matrices

The following are functions that could be used to generate arbitrary matrices to investigate matrix operations when you cannot think of one to generate yourself. They are special matrices that were discovered by mathematicians, the motivation for their discovery and other details about them you are *not* expected to understand.

`pascal(n)` generates a Pascal matrix of order n. Technically, this is a symmetric positive definite matrix with entries made up from *Pascal's triangle*, e.g.,

```
pascal(4)
```

results in

```
ans =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

`magic(n)` generates an $n \times n$ *magic square*.

MATLAB has a number of other functions which generate special matrices, such as `gallery`, `hadamard`, `hankel`, `hilb`, `toeplitz`, `vander`, etc. See `help elmat`.

6.1.10 Using MATLAB functions with matrices

When a MATLAB mathematical or trigonometric function has a matrix argument, the function operates on every element of the matrix, as you would expect. However, many of the other MATLAB functions operate on matrices *column by column*, e.g., if

```
a =
     1     0     1
     1     1     1
     0     0     1
```

the statement

```
all(a)
```

results in

```
ans =
     0     0     1
```

For each *column* of `a` where all the elements are true (non-zero) `all` returns 1, otherwise it returns 0. `all` therefore returns a logical vector when it takes a matrix argument. To test if all the elements of `a` are true, use `all` twice. In this example, the statement

```
all(all(a))
```

returns 0 because some of the elements of `a` are 0. On the other hand the statement

```
any(a)
```

returns

```
ans =
     1     1     1
```

because each column of `a` has at least one non-zero element, and `any(any(a))` returns 1, since `a` itself has at least one non-zero element.

If you are not sure whether a particular function operates columnwise or element by element on matrices, you can always request `help`.

6.1.11 Manipulating matrices

Here are some functions for manipulating matrices. See `help` for details.

`diag` extracts or creates a diagonal.

`fliplr` flips from left to right.

`flipud` flips from top to bottom.

`rot90` rotates.

`tril` extracts the lower triangular part, e.g., the statement

```
tril(pascal(4))
```

results in

```
ans =
     1     0     0     0
     1     2     0     0
     1     3     6     0
     1     4    10    20
```

`triu` extracts the upper triangular part.

6.1.12 Array (element-by-element) operations on matrices

The array operations discussed in Chapter 2 all apply to matrices as well as vectors. For example, if `a` is a matrix, `a * 2` multiplies each element of `a` by 2, and if

```
a =
     1     2     3
     4     5     6
```

the statement

```
a .^ 2
```

results in

```
ans =
     1     4     9
    16    25    36
```

6.1.13 Matrices and `for`

If

```
a =
     1     2     3
     4     5     6
     7     8     9
```

the statements

```
for v = a
  disp(v')
end
```

result in

```
1     4     7
2     5     8
3     6     9
```

What happens in this most general form of the `for` statement is that the index `v` takes on the value of each *column* of the matrix expression `a` in turn. This provides a neat way of processing all the columns of a matrix. You can do the same with the rows if you transpose `a`, e.g., the statements

```
for v = a'
  disp(v')
end
```

display the rows of `a` one at a time. Get it?

6.1.14 Visualization of matrices

Matrices can be visualized graphically in MATLAB. This subject is discussed briefly in Chapter 9, with illustrations.

6.1.15 Vectorizing nested `for`s: loan repayment tables

If a regular fixed payment P is made n times a year to repay a loan of amount A over a period of k years, where the nominal annual interest rate is r, P is given by

$$P = \frac{rA(1 + r/n)^{nk}}{n[(1 + r/n)^{nk} - 1]}.$$

We would like to generate a table of repayments for a loan of \$1 000 over 15, 20 or 25 years, at interest rates that vary from 10% to 20% per annum, in steps of 1%. Since P is directly proportional to A in the above formula, the repayments of a loan of any amount can be found by simple proportion from such a table.

The conventional way of handling this is with 'nested' `for`s. The `fprintf` statements are necessary in order to get the output for each interest rate on the same line (see MATLAB Help):

```
A = 1000;                % amount borrowed
n = 12;                  % number of payments per year

for r = 0.1 : 0.01 : 0.2
  fprintf( '%4.0f%', 100 * r );
  for k = 15 : 5 : 25
    temp = (1 + r/n) ^ (n*k);
    P = r * A * temp / (n * (temp - 1));
    fprintf( '%10.2f', P );
  end;
  fprintf( '\n' );       % new line
end;
```

Some sample output (with headings edited in):

```
rate %    15 yrs     20 yrs     25 yrs

  10      10.75       9.65       9.09
  11      11.37      10.32       9.80
  ...
  19      16.83      16.21      15.98
  20      17.56      16.99      16.78
```

However, we saw in Chapter 2 that `for` loops can often be vectorized, saving a lot of computing time (and also providing an interesting intellectual challenge!). The inner loop can easily be vectorized; the following code uses only one `for`:

```
...
for r = 0.1 : 0.01 : 0.2
  k = 15 : 5 : 25;
  temp = (1 + r/n) .^ (n*k);
  P = r * A * temp / n ./ (temp - 1);
  disp( [100 * r, P] );
end;
```

Note the use of the array operators.

The really tough challenge, however, is to vectorize the outer loop as well. We want a table with 11 rows and 3 columns. Start by assigning values to `A` and `n` (from the command line):

```
A = 1000;
n = 12;
```

Then generate a column vector for the interest rates:

```
r = [0.1:0.01:0.2]'
```

Now change this into a table with 3 columns each equal to `r`:

```
r = repmat(r, [1 3])
```

The matrix `r` should look like this:

```
0.10          0.10          0.10
0.11          0.11          0.11
...
0.19          0.19          0.19
0.20          0.20          0.20
```

Now do a similar thing for the repayment periods `k`. Generate the row vector

```
k = 15:5:25
```

and expand it into a table with 11 rows each equal to k:

```
k = repmat(k, [11 1])
```

This should give you

```
15     20     25
15     20     25
...
15     20     25
15     20     25
```

The formula for P is a little complicated, so let's do it in two steps:

```
temp = (1 + r/n) .^ (n * k);
P = r * A .* temp / n ./ (temp - 1)
```

Finally, you should get this for P:

```
10.75          9.65          9.09
11.37         10.32          9.80
...
16.83         16.21         15.98
17.56         16.99         16.78
```

This works, because of the way the tables r and k have been constructed, and because the MATLAB array operations are performed element by element. For example, when the calculation is made for P in row 2 and column 1, the array operations pick out row 2 of r (all 0.11) and column 1 of k (all 15), giving the correct value for P (11.37).

The nested for way might be easier to program, but this way is certainly more interesting (and quicker to execute).

6.1.16 Multi-dimensional arrays

MATLAB arrays can have more than two dimensions. For example, suppose you create the matrix

```
a = [1:2; 3:4]
```

You can add a third dimension to a with

```
a(:,:,2) = [5:6; 7:8]
```

MATLAB responds with

```
a(:,:,1) =
     1     2
     3     4
a(:,:,2) =
     5     6
     7     8
```

It helps to think of the 3-D array `a` as a series of 'pages', with a matrix on each page. The third dimension of `a` numbers the pages. This is analogous to a spreadsheet with multiple sheets: each sheet contains a table (matrix).

You can get into hyperpages with higher dimensions if you like!

See `help datatypes` for a list of special multi-dimensional array functions.

6.2 MATRIX OPERATIONS

We have seen that *array* operations are performed element by element on matrices. However, *matrix* operations, which are fundamental to MATLAB, are defined differently in certain cases, in the mathematical sense.

Matrix addition and subtraction are defined in the same way as the equivalent array operations, i.e., element by element. Matrix multiplication, however, is quite different.

6.2.1 Matrix multiplication

Matrix *multiplication* is probably the most important matrix operation. It is used widely in such areas as network theory, solution of linear systems of equations, transformation of co-ordinate systems, and population modeling, to name but a very few. The rules for multiplying matrices look a little weird if you've never seen them before, but will be justified by the applications that follow.

When two matrices **A** and **B** are multiplied together in this sense, their product is a third matrix **C**. The operation is written as

$\mathbf{C} = \mathbf{AB}$,

and the general element c_{ij} of **C** is formed by taking the *scalar product* of the ith row of **A** with the jth column of **B**. (The scalar product of two *vectors* **x** and **y** is $x_1y_1 + x_2y_2 + \ldots$, where x_i and y_i are the components of the vectors.) It follows that **A** and **B** can only be successfully multiplied (in that order) if the number of columns in **A** is the same as the number of rows in **B**.

The general definition of matrix multiplication is as follows: If **A** is an $n \times m$ matrix and **B** is an $m \times p$ matrix, their product **C** will be an $n \times p$ matrix such that the general element c_{ij} of **C** is given by

$$c_{ij} = \sum_{k=1}^{m} a_{ik}\, b_{kj}.$$

Note that in general **AB** is not equal to **BA** (matrix multiplication is not *commutative*).

Example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 15 & 14 \end{bmatrix}$$

Since a vector is simply a one-dimensional matrix, the definition of matrix multiplication given above also applies when a vector is multiplied by an appropriate matrix, e.g.,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 8 \\ 18 \end{bmatrix}.$$

The operator * is used for matrix multiplication, as you may have guessed. For example, if

```
a =
     1     2
     3     4
```

and

```
b =
     5     6
     0    -1
```

the statement

```
c = a * b
```

results in

```
c =
     5     4
    15    14
```

Note the important difference between the *array* operation `a .* b` (evaluate by hand and check with MATLAB) and the *matrix* operation `a * b`.

To multiply a matrix by a vector in that order, the vector must be a column vector. So if

```
b = [2 3]'
```

the statement

```
c = a * b
```

results in

```
c =
     8
    18
```

6.2.2 Matrix exponentiation

The matrix operation $\mathbf{A}^2$ means $\mathbf{A} \times \mathbf{A}$, where $\mathbf{A}$ must be a square matrix. The operator `^` is used for matrix exponentiation, e.g., if

```
a =
     1     2
     3     4
```

the statement

```
a ^ 2
```

results in

```
ans =
     7    10
    15    22
```

which is the same as `a * a` (try it).

Again, note the difference between the array operation `a .^ 2` (evaluate by hand and check with MATLAB) and the matrix operation `a ^ 2`.

6.3 OTHER MATRIX FUNCTIONS

Here are some of MATLAB's more advanced matrix functions.

`det` determinant.

`eig` eigenvalue decomposition.

`expm` matrix exponential, i.e., $e^{\mathbf{A}}$, where $\mathbf{A}$ is a matrix. The matrix exponential may be used to evaluate analytical solutions of linear ordinary differential equations with constant coefficients.

`inv` inverse.

`lu` LU factorization (into lower and upper triangular matrices).

`qr` orthogonal factorization.

`svd` singular value decomposition.

6.4 POPULATION GROWTH: LESLIE MATRICES

Our first application of matrices is in population dynamics.

Suppose we want to model the growth of a population of rabbits, in the sense that given their number at some moment, we would like to estimate the size of the population in a few years' time. One approach is to divide the rabbit population up into a number of age classes, where the members of each age class are one time unit older than the members of the previous class, the time unit being whatever is convenient for the population being studied (days, months, etc.).

If X_i is the size of the ith age class, we define a *survival factor* P_i as the proportion of the ith class that survive to the $(i+1)$th age class, i.e. the proportion that 'graduate'. F_i is defined as the *mean fertility* of the ith class. This is the mean number of newborn individuals expected to be produced during one time interval by each member of the ith class at the beginning of the interval (only females count in biological modeling, since there are always enough males to go round!).

Suppose for our modified rabbit model we have three age classes, with X_1, X_2 and X_3 members, respectively. We will call them young, middle-aged and old-aged for convenience. We will take our time unit as one month, so X_1 is the number that were born during the current month, and which will be considered as youngsters at the end of the month. X_2 is the number of middle-aged rabbits at the end of the month, and X_3 the number of oldsters. Suppose the youngsters cannot reproduce, so that $F_1 = 0$. Suppose the fertility rate for middle-aged rabbits is 9, so $F_2 = 9$, while for oldsters $F_3 = 12$. The probability of survival from youth to middle-age is one third, so $P_1 = 1/3$, while no less

than half the middle-aged rabbits live to become oldsters, so $P_2 = 0.5$ (we are assuming for the sake of illustration that all old-aged rabbits die at the end of the month—this can be corrected easily). With this information we can quite easily compute the changing population structure month by month, as long as we have the population breakdown to start with.

If we now denote the current month by t, and next month by $(t+1)$, we can refer to this month's youngsters as $X_1(t)$, and to next month's as $X_1(t+1)$, with similar notation for the other two age classes. We can then write a scheme for updating the population from month t to month $(t+1)$ as follows:

$$X_1(t+1) = F_2X_2(t) + F_3X_3(t),$$
$$X_2(t+1) = P_1X_1(t),$$
$$X_3(t+1) = P_2X_2(t).$$

We now define a population vector $\mathbf{X}(t)$, with three components, $X_1(t)$, $X_2(t)$, and $X_3(t)$, representing the three age classes of the rabbit population in month t. The above three equations can then be rewritten as

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}_{(t+1)} = \begin{bmatrix} 0 & F_2 & F_3 \\ P_1 & 0 & 0 \\ 0 & P_2 & 0 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}_t$$

where the subscript at the bottom of the vectors indicates the month. We can write this even more concisely as the matrix equation

$$\mathbf{X}(t+1) = \mathbf{L}\,\mathbf{X}(t), \tag{6.1}$$

where $\mathbf{L}$ is the matrix

$$\begin{bmatrix} 0 & 9 & 12 \\ 1/3 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix}$$

in this particular case. $\mathbf{L}$ is called a *Leslie matrix*. A population model can always be written in the form of Equation (6.1) if the concepts of age classes, fertility, and survival factors, as outlined above, are used.

Now that we have established a matrix representation for our model, we can easily write a script using matrix multiplication and repeated application of Equation (6.1):

$$\mathbf{X}(t+2) = \mathbf{L}\,\mathbf{X}(t+1),$$
$$\mathbf{X}(t+3) = \mathbf{L}\,\mathbf{X}(t+2), \text{ etc.}$$

We will assume to start with that we have one old (female) rabbit, and no others, so $X_1 = X_2 = 0$, and $X_3 = 1$. Here is the script:

```
% Leslie matrix population model
n = 3;
L = zeros(n);      % all elements set to zero
L(1,2) = 9;
L(1,3) = 12;
L(2,1) = 1/3;
L(3,2) = 0.5;
x = [0 0 1]';     % remember x must be a column vector!

for t = 1:24
  x = L * x;
  disp( [t x' sum(x)] )  % x' is a row
end
```

The output, over a period of 24 months (after some editing), is:

Month	Young	Middle	Old	Total
1	12	0	0	12
2	0	4	0	4
3	36	0	2	38
4	24	12	0	36
5	108	8	6	122
...				
22	11184720	1864164	466020	13514904
23	22369716	3728240	932082	27030038
24	44739144	7456572	1864120	54059836

It so happens that there are no 'fractional' rabbits in this example. If there are any, they should be kept, and not rounded (and certainly not truncated). They occur because the fertility rates and survival probabilities are averages.

If you look carefully at the output you may spot that after some months the total population doubles every month. This factor is called the *growth factor*, and is a property of the particular Leslie matrix being used (for those who know about such things, it's the *dominant eigenvalue* of the matrix). The growth factor is 2 in this example, but if the values in the Leslie matrix are changed, the long-term growth factor changes too (try it and see).

Figure 6.1 shows how the total rabbit population grows over the first 15 months. To draw this graph yourself, insert the line

```
p(t) = sum(x);
```

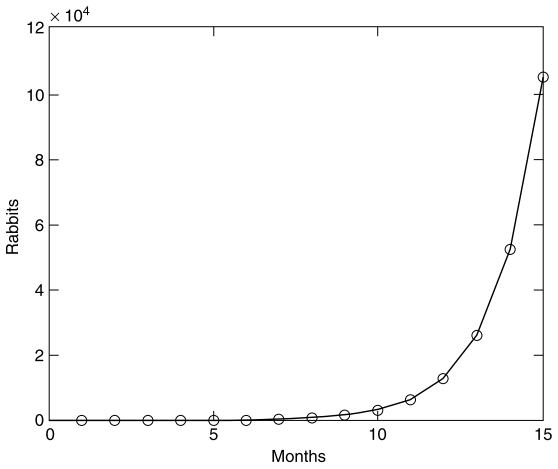

FIGURE 6.1 Total rabbit population over 15 months.

in the `for` loop after the statement `x = L * x;`, and run the program again. The vector `p` will contain the total population at the end of each month. Then enter the commands

```
plot(1:15, p(1:15)), xlabel('months'), ylabel('rabbits')
hold, plot(1:15, p(1:15),'o')
```

The graph demonstrates *exponential* growth. If you plot the population over the full 24-month period, you will see that the graph gets much steeper. This is a feature of exponential growth.

You probably didn't spot that the numbers in the three age classes tend to a limiting ratio of 24:4:1. This can be demonstrated very clearly if you run the model with an initial population structure having this limiting ratio. The limiting ratio is called the *stable age distribution* of the population, and again it is a property of the Leslie matrix (in fact, it is the *eigenvector* belonging to the dominant eigenvalue of the matrix). Different population matrices lead to different stable age distributions.

The interesting point about this is that a given Leslie matrix always eventually gets a population into the *same* stable age distribution, which increases eventually by the *same* growth factor each month, *no matter what the initial population breakdown is*. For example, if you run the above model with any other initial population, it will always eventually get into a stable age distribution of 24:4:1 with a growth factor of 2 (try it and see).

See `help eig` if you're interested in using MATLAB to compute eigenvalues and eigenvectors.

6.5 MARKOV PROCESSES

Often a process that we wish to model may be represented by a number of possible *discrete* (i.e. discontinuous) states that describe the outcome of the process. For example, if we are spinning a coin, then the outcome is adequately represented by the two states 'heads' and 'tails' (and nothing in between). If the process is random, as it is with spinning coins, there is a certain probability of being in any of the states at a given moment, and also a probability of changing from one state to another. If the probability of moving from one state to another depends on the present state only, and not on any previous state, the process is called a *Markov chain*. The progress of the myopic sailor in Chapter 13 is an example of such a process. Markov chains are used widely in such diverse fields as biology and business decision making, to name just two areas.

6.5.1 A random walk

This example is a variation on the random walk simulation of Chapter 13. A street has six intersections. A short-sighted student wanders down the street. His home is at intersection 1, and his favorite internet cafe at intersection 6. At each intersection other than his home or the cafe he moves in the direction of the cafe with probability 2/3, and in the direction of his home with probability 1/3. In other words, he is twice as likely to move towards the cafe as towards his home. He never wanders down a side street. If he reaches his home or the cafe, he disappears into them, never to re-appear (when he disappears we say in Markov jargon that he has been *absorbed*).

We would like to know: what are the chances of him ending up at home or in the cafe, if he starts at a given corner (other than home or the cafe, obviously)? He can clearly be in one of six states, with respect to his random walk, which can be labeled by the intersection number, where state 1 means *Home* and state 6 means *Cafe*. We can represent the probabilities of being in these states by a six-component *state vector* $\mathbf{X}(t)$, where $X_i(t)$ is the probability of him being at intersection i at moment t. The components of $\mathbf{X}(t)$ must sum to 1, since he has to be in one of these states.

We can express this Markov process with the following *transition probability matrix*, $\mathbf{P}$, where the rows represent the next state (i.e. corner), and the columns represent the present state:

	Home	2	3	4	5	Cafe
Home	1	1/3	0	0	0	0
2	0	0	1/3	0	0	0
3	0	2/3	0	1/3	0	0
4	0	0	2/3	0	1/3	0
5	0	0	0	2/3	0	0
Cafe	0	0	0	0	2/3	1

The entries for *Home-Home* and *Cafe-Cafe* are both 1 because he stays there with certainty.

Using the probability matrix **P** we can work out his chances of being, say, at intersection 3 at moment $(t+1)$ as

$X_3(t+1) = 2/3X_2(t) + 1/3X_4(t)$.

To get to 3, he must have been at either 2 or 4, and his chances of moving from there are 2/3 and 1/3, respectively.

Mathematically, this is identical to the Leslie matrix problem. We can therefore form the new state vector from the old one each time with a matrix equation:

$\mathbf{X}(t+1) = \mathbf{P}\,\mathbf{X}(t)$.

If we suppose the student starts at intersection 2, the initial probabilities will be (0; 1; 0; 0; 0; 0). The Leslie matrix script may be adapted with very few changes to generate future states:

```
n = 6;
P = zeros(n);          % all elements set to zero

for i = 3:6
  P(i,i-1) = 2/3;
  P(i-2,i-1) = 1/3;
end

P(1,1) = 1;
P(6,6) = 1;
x = [0 1 0 0 0 0]';    % remember x must be a column vector!

for t = 1:50
  x = P * x;
  disp( [t x'] )
end
```

Edited output:

```
Time      Home          2          3          4          5       Cafe

   1    0.3333          0     0.6667          0          0          0
   2    0.3333     0.2222          0     0.4444          0          0
   3    0.4074          0     0.2963          0     0.2963          0
   4    0.4074     0.0988          0     0.2963          0     0.1975
   5    0.4403          0     0.1646          0     0.1975     0.1975
   6    0.4403     0.0549          0     0.1756          0     0.3292
   7    0.4586          0     0.0951          0     0.1171     0.3292
   8    0.4586     0.0317          0     0.1024          0     0.4073
  ...
  20    0.4829     0.0012          0     0.0040          0     0.5119
  ...
  40    0.4839     0.0000          0     0.0000          0     0.5161
  ...
  50    0.4839     0.0000          0     0.0000          0     0.5161
```

By running the program for long enough, we soon find the limiting probabilities: he ends up at home about 48% of the time, and at the cafe about 52% of the time. Perhaps this is a little surprising; from the transition probabilities, we might have expected him to get to the cafe rather more easily. It just goes to show that you should never trust your intuition when it comes to statistics!

Note that the Markov chain approach is *not* a simulation: one gets the *theoretical* probabilities each time (this can all be done mathematically, without a computer).

6.6 LINEAR EQUATIONS

A problem that often arises in scientific applications is the solution of a system of linear equations, e.g.,

$$3x + 2y - z = 10 \tag{6.2}$$

$$-x + 3y + 2z = 5 \tag{6.3}$$

$$x - y - z = -1. \tag{6.4}$$

MATLAB was designed to solve a system like this directly and very easily, as we shall now see.

If we define the matrix of coefficients, **A**, as

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

and the vectors of unknowns, **x**, and the right hand side, **b**, as

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix},$$

we can write the above system of three equations in matrix form as

$$\begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix},$$

or even more concisely as the single matrix equation

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{6.5}$$

The solution may then be written as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}, \tag{6.6}$$

where $\mathbf{A}^{-1}$ is the matrix inverse of $\mathbf{A}$ (i.e., the matrix which when multiplied by $\mathbf{A}$ gives the identity matrix $\mathbf{I}$).

6.6.1 MATLAB's solution

To see how MATLAB solves this system, first recall that the *left division* operator `\` may be used on scalars, i.e., `a \ b` is the same as `b / a` if `a` and `b` are scalars. However, it can also be used on vectors and matrices, in order to solve linear equations. Enter the following statements on the command line to solve Equations (6.2)–(6.4):

```
A = [3 2 -1; -1 3 2; 1  -1  -1];
b = [10 5 -1]';
x = A \ b
```

This should result in

```
x =
   -2.0000
    5.0000
   -6.0000
```

In terms of our notation, this means that the solution is $x = -2$, $y = 5$, $z = -6$.

You can think of the matrix operation `A \ b` as '`b` divided by `A`', or as 'the inverse of `A` multiplied by `b`', which is essentially what Equation (6.6) means. A colleague of mine reads this operation as '`A` under `b`', which you may also find helpful.

You may be tempted to implement Equation (6.6) in MATLAB as follows:

```
x = inv(A) * b
```

since the function `inv` finds a matrix inverse directly. However, `A \ b` is actually more accurate and efficient; see **MATLAB Help: Functions — Alphabetical List** and click on `inv`.

6.6.2 The residual

Whenever we solve a system of linear equations numerically we need to have some idea of how accurate the solution is. The first thing to check is the *residual*, defined as

```
r = A*x - b
```

where `x` is the result of the operation `x = A \ b`. Theoretically the residual `r` should be zero, since the expression `A * x` is supposed to be equal to `b`, according to Equation (6.5), which is the equation we are trying to solve. In our example here the residual is (check it)

```
r =
  1.0e-015 *
         0
    0.8882
    0.6661
```

This seems pretty conclusive: all the elements of the residual are less than 10^{-15} in absolute value. Unfortunately, there may still be problems lurking beneath the surface, as we shall see shortly.

We will, however, first look at a situation where the residual turns out to be far from zero.

6.6.3 Over-determined systems

When we have more equations than unknowns, the system is called *over-determined*, e.g.,

$$\begin{aligned} x - y &= 0 \\ y &= 2 \\ x &= 1. \end{aligned}$$

Surprisingly, perhaps, MATLAB gives a solution to such a system. If

```
A = [1 -1; 0 1; 1 0];
b = [0 2 1]';
```

the statement

```
x = A \ b
```

results in

```
x =
    1.3333
    1.6667
```

The residual `r = A*x - b` is now

```
r =
   -0.3333
   -0.3333
    0.3333
```

What happens in this case is that MATLAB produces the *least squares best fit*. This is the value of `x` which makes the *magnitude* of `r`, i.e.,

$$\sqrt{\texttt{r(1)}^2 + \texttt{r(2)}^2 + \texttt{r(3)}^3},$$

as small as possible. You can compute this quantity (0.5774) with `sqrt(r'*r)` or `sqrt(sum(r .* r))`. There is a nice example of fitting a decaying exponential function to data with a least squares fit in **MATLAB Help: Mathematics: Matrices and Linear Algebra: Solving Linear Equations: Overdetermined Systems**.

6.6.4 Under-determined systems

If there are fewer equations than unknowns, the system is called *under-determined*. In this case there are an infinite number of solutions; MATLAB will find one which has zeros for some of the unknowns.

The equations in such a system are the constraints in a linear programming problem.

6.6.5 Ill conditioning

Sometimes the coefficients of a system of equations are the results of an experiment, and may be subject to error. We need in that case to know how sensitive the solution is to the experimental errors. As an example, consider the system

$$10x + 7y + 8z + 7w = 32$$
$$7x + 5y + 6z + 5w = 23$$

$$8x + 6y + 10z + 9w = 33$$

$$7x + 5y + 9z + 10w = 31$$

Use matrix left division to show that the solution is $x = y = z = w = 1$. The residual is exactly zero (check it), and all seems well.

However, if we change the right-hand side constants to 32.1, 22.9, 32.9 and 31.1, the 'solution' is now given by $x = 6, y = -7.2, z = 2.9, w = -0.1$. The residual is very small.

A system like this is called *ill-conditioned*, meaning that a small change in the coefficients leads to a large change in the solution. The MATLAB function `rcond` returns the *condition estimator*, which tests for ill conditioning. If `A` is the coefficient matrix, `rcond(A)` will be close to zero if `A` is ill-conditioned, but close to 1 if it is well-conditioned. In this example, the condition estimator is about 2×10^{-4}, which is pretty close to zero.

Some authors suggest the rule of thumb that a matrix is ill-conditioned if its determinant is small compared to the entries in the matrix. In this case the determinant of `A` is 1 (check with the function `det`) which is about an order of magnitude smaller than most of its entries.

6.6.6 Matrix division

Matrix left division, `A\B`, is defined whenever `B` has as many rows as `A`. This corresponds formally to `inv(A)*B` although the result is obtained without computing the inverse explicitly. In general,

```
x = A \ B
```

is a solution to the system of equations defined by $\mathbf{A}\mathbf{x} = \mathbf{B}$.

If `A` is square, matrix left division is done using Gauss elimination.

If `A` is not square the over- or under-determined equations are solved in the least squares sense. The result is an $m \times n$ matrix `X`, where m is the number of columns of `A` and n is the number of columns of `B`.

Matrix right division, `B/A`, is defined in terms of matrix left division such that `B/A` is the same as `(A'\B')'`. So with `a` and `b` defined as for Equations (6.2)–(6.4), this means that

```
x = (b'/a')'
```

gives the same solution, doesn't it? Try it, and make sure you can see why.

Sometimes the least squares solutions computed by \ or / for over- or under-determined systems can cause surprises, since you can legally divide one *vector* by another. For example, if

```
a = [1 2];
b = [3 4];
```

the statement

```
a / b
```

results in

```
ans =
    0.4400
```

This is because `a/b` is the same as `(b'\ a')'`, which is formally the solution of $\mathbf{b}' x' = \mathbf{a}'$. The result is a scalar, since `a'` and `b'` each have one column, and is the least squares solution of

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

With this under your belt, can you explain why

```
a \ b
```

gives

```
ans =
         0         0
    1.5000    2.0000
```

(try writing the equations out in full)?

A complete discussion of the algorithms used in solving simultaneous linear equations may be found in the online documentation under **MATLAB: Reference: MATLAB Function Reference: Alphabetical List of Functions: Arithmetic Operators** `+ - * / \ ^ '`.

6.7 SPARSE MATRICES

Matrices can sometimes get very large, with thousands of entries. Very large matrices can occupy huge amounts of memory, and processing them can take up a lot of computer time. For example, a system of n simultaneous linear equations requires n^2 matrix entries, and the computing time to solve them is proportional to n^3.

However, some matrices have relatively few *non-zero* entries. Such matrices are called *sparse* as opposed to *full*. The father of modern numerical linear algebra, J.H. Wilkinson, has remarked that a matrix is sparse if 'it has enough zeros that it pays to take advantage of them'. MATLAB has facilities for exploiting sparsity, which have the potential for saving huge amounts of memory and processing time. For example, the matrix representation of a certain type of partial differential equation (a 5-point Laplacian) on a square 64×64 grid is a 4096×4096 element matrix with 20 224 non-zero elements. The sparse form of this matrix in MATLAB occupies only 250 kB of memory, whereas the full version of the same matrix would occupy 128 MB, which is way beyond the limits of most desktop computers. The solution of the system $\mathbf{Ax} = \mathbf{b}$ using sparse techniques is about 4000 times faster than solving the full case, i.e., 10 s instead of 12 hours!

In this section (which you can safely skip, if you want to) we will look briefly at how to create sparse matrices in MATLAB. For a full description of sparse matrix techniques consult **MATLAB Help: Mathematics: Sparse Matrices**.

First an example, then an explanation. The transition probability matrix for the random walk problem in Section 6.5 is a good candidate for sparse representation. Only ten of its 36 elements are non-zero. Since the non-zeros appear only on the diagonal and the sub- and super-diagonals, a matrix representing more intersections would be even sparser. For example, a 100×100 representation of the same problem would have only 198 non-zero entries, i.e., 1.98%.

To represent a sparse matrix all that MATLAB needs to record are the non-zero entries with their row and column indices. This is done with the `sparse` function. The transition matrix of Section 6.5 can be set up as a sparse matrix with the statements

```
n = 6;
P = sparse(1, 1, 1, n, n);
P = P + sparse(n, n, 1, n, n);
P = P + sparse(1:n-2, 2:n-1, 1/3, n, n);
P = P + sparse(3:n, 2:n-1, 2/3, n, n)
```

which result in (with `format rat`)

```
P =

   (1,1)        1
   (1,2)      1/3
   (3,2)      2/3
   (2,3)      1/3
   (4,3)      2/3
   (3,4)      1/3
   (5,4)      2/3
   (4,5)      1/3
   (6,5)      2/3
   (6,6)        1
```

Each line of the display of a sparse matrix gives a non-zero entry with its row and column, e.g., 2/3 in row 3 and column 2. To display a sparse matrix in full form, use the function

```
full(P)
```

which results in

```
ans =
      1        1/3        0        0        0        0
      0         0        1/3       0        0        0
      0        2/3        0       1/3       0        0
      0         0        2/3       0       1/3       0
      0         0         0       2/3       0        0
      0         0         0        0       2/3       1
```

(also with `format rat`).

The form of the `sparse` function used here is

```
sparse(rows, cols, entries, m, n)
```

This generates an `m` × `n` sparse matrix with non-zero `entries` having subscripts (`rows`, `cols`) (which may be vectors), e.g., the statement

```
sparse(1:n-2, 2:n-1, 1/3, n, n);
```

(with `n = 6`) creates a 6 × 6 sparse matrix with 4 non-zero elements, being 1/3s in rows 1–4 and columns 2–5 (most of the super-diagonal). Note that repeated

use of `sparse` produces a number of 6×6 matrices, which must be added together to give the final form. Sparsity is therefore preserved by operations on sparse matrices. See `help` for more details of `sparse`.

It's quite easy to test the efficiency of sparse matrices. Construct a (full) identity matrix

```
a = eye(1000);
```

Determine the time it takes to compute `a^2`. Now let us take advantage of the sparseness of `a`. It is an ideal candidate for being represented as a sparse matrix, since only 1000 of its one million elements are non-zero. It is represented in sparse form as

```
s = sparse(1:1000, 1:1000, 1, 1000, 1000);
```

Now check the time it takes to find `a^2`. Use `tic` and `toc` to find out how much faster is the computation.

The function `full(a)` returns the full form of the sparse matrix `a` (without changing the sparse representation of `a` itself). Conversely, `sparse(a)` returns the sparse form of the full matrix `a`.

The function `spy` provides a neat visualization of sparse matrices. Try it on `P`. Then enlarge `P` to about 50×50, and `spy` it.

SUMMARY

- A matrix is a 2-D array. Elements may be referenced in the conventional way with two subscripts. Alternatively, one subscript may be used, in which case the matrix is 'unwound' by columns.
- The colon operator may be used to represent all the rows or columns of a matrix, and also as a single subscript.
- The keyword `end` refers to the last row or column of a matrix.
- Use `repmat` to duplicate rows or columns of a matrix.
- Use the empty array `[ ]` to delete rows or columns of a matrix.
- Arrays may have more than two dimensions. In the case of a 3-D array the third subscript may be thought of as numbering pages, where each page contains a matrix defined by the first two subscripts.
- The matrix operations of multiplication and exponentiation are implemented with the matrix operators `*` and `^`.
- The matrix left division operator `\` is used for solving systems of linear equations directly. Because the matrix division operators `\` and `/` can sometimes produce surprising results with the least squares solution method,

you should always compute the residual when solving a system of equations.

- If you work with large matrices with relatively few non-zero entries you should consider using MATLAB's sparse matrix facilities.

EXERCISES

6.1 Set up any 3×3 matrix `a`. Write some command-line statements to perform the following operations on `a`:
(a) interchange columns 2 and 3;
(b) add a fourth column (of 0s);
(c) insert a row of 1s as the new second row of `a` (i.e. move the current second and third rows down);
(d) remove the second column.

6.2 Compute the limiting probabilities for the student in Section 6.5 when he starts at each of the remaining intersections in turn, and confirm that the closer he starts to the cafe, the more likely he is to end up there.
Compute `P^50` directly. Can you see the limiting probabilities in the first row?

6.3 Solve the equations

$$2x - y + z = 4$$
$$x + y + z = 3$$
$$3x - y - z = 1$$

using the left division operator. Check your solution by computing the residual. Also compute the determinant (`det`) and the condition estimator (`rcond`). What do you conclude?

6.4 This problem, suggested by R.V. Andree, demonstrates ill conditioning (where small changes in the coefficients cause large changes in the solution). Use the left division operator to show that the solution of the system

$$x + 5.000y = 17.0$$
$$1.5x + 7.501y = 25.503$$

is $x = 2$, $y = 3$. Compute the residual.
Now change the term on the right-hand side of the second equation to 25.501, a change of about one part in 12 000, and find the new solution and the residual. The solution is completely different. Also try changing this term to 25.502, 25.504, etc. If the coefficients are subject to experimental errors, the solution is clearly meaningless. Use `rcond` to find the condition estimator and `det` to compute the determinant. Do these values confirm ill conditioning?
Another way to anticipate ill conditioning is to perform a *sensitivity analysis* on the coefficients: change them all in turn by the same small percentage, and observe what effect this has on the solution.

6.5 Use `sparse` to represent the Leslie matrix in Section 6.4. Test your representation by projecting the rabbit population over 24 months.

6.6 If you are familiar with *Gauss reduction* it is an excellent programming exercise to code a Gauss reduction directly with operations on the rows of the augmented coefficient matrix. See if you can write a function

```
x = mygauss(a, b)
```

to solve the general system $\mathbf{Ax} = \mathbf{b}$. Skillful use of the colon operator in the row operations can reduce the code to a few lines!
Test it on $\mathbf{A}$ and $\mathbf{b}$ with random entries, and on the systems in Section 6.6 and Exercise 16.4. Check your solutions with left division.