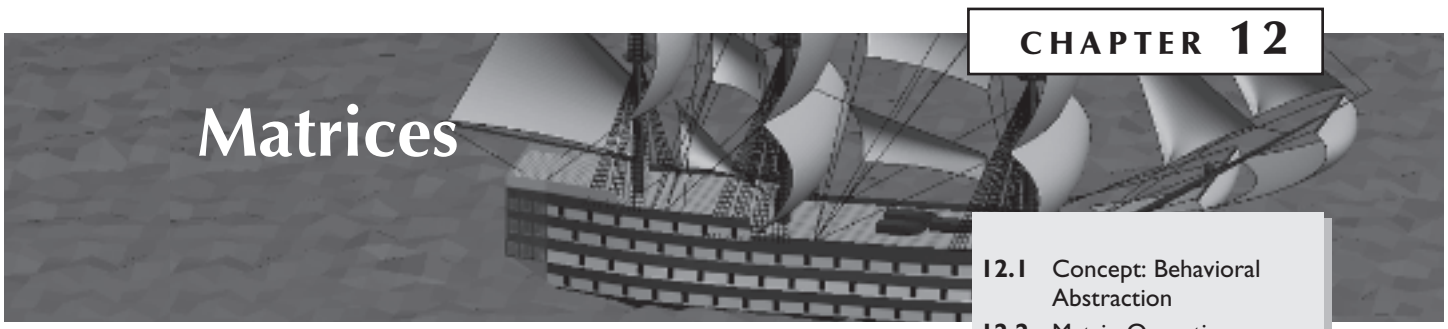
CHAPTER 12

Matrices



Chapter Objectives

This chapter shows matrices as logical extensions of arrays. You will learn about two specialized operations performed with matrices:

- Multiplication for coordinate rotation
- Division for solving simultaneous equations

Although the matrix operations that are the subject of this chapter can be performed on pairs of vectors or arrays that meet certain criteria, when using these operations, we tend to refer to the data objects as matrices. In most mathematical discussions, the words "matrix" and "array" can be used interchangeably, and rightly so, because they store data in exactly the same form. Moreover, almost all of the operations we can perform on an array can also be performed on a matrix—logical operations, concatenation, slicing, and most of the arithmetic operations behave identically. The fact that some of the mathematical operations are defined differently gives us a chance to think about an important concept that is usually well hidden within the MATLAB language definition.

12.1 Concept: Behavioral Abstraction

Recall the following concepts:

- *Abstraction* is the ability to ignore specific details and generalize the description of an entity
- *Data abstraction* is the specific example of abstraction that we first considered whereby we could treat vectors of data (and later other collections like structures and arrays) as single entities rather than enumerating their elements individually
- *Procedural abstraction* are functions that collect multiple operations into a form; once they are developed, we can overlook the specific details and treat them as a "black box," much as we treat built-in functions

Behavioral abstraction combines data and procedural abstraction, encapsulating not only collections of data, but also the operations that are legal to perform on that data. One might argue that this is a new, irrelevant concept best ignored until "we just have to!" However, consider the rules we have had to establish for what we can and cannot do with data collections we have seen so far. For example, am I able to add two arrays together? Yes, but only if they have the same number of rows and columns, or if one of them is scalar. Can I add two character strings? Almost the same answer, except that each string is first converted to a numerical quantity and the result is a vector of numbers and not a string. Can I add two cell arrays? No.

So at least some, and maybe all, data collections also "understand" the set of operations that are permitted on the data. This encapsulation of data and operations is the essence of behavioral abstraction. Therefore, we distinguish arrays from matrices not by the data they collect, but by the operations that are legal to perform on them.

12.2 Matrix Operations

The arithmetic operations that differ between arrays and matrices are multiplication, division, and exponentiation.

12.2.1 Matrix Multiplication

Previously, when we considered multiplying two arrays, we called this scalar multiplication, and it had the following typical array operation characteristics:

- Either the two arrays must be the same size, or one of them must be scalar
- The multiplication was indicated with the `.*` operator

- The result was an array with the same size as the larger original array
- Each element of the result was the product of the corresponding elements in the original two arrays

This is best illustrated in Figure 12.1. Scalar division and exponentiation have the same constraints.

Matrix multiplication, on the other hand, performed using the normal * operator, is an entirely different logical operation, as shown in Figure 12.2. The logical characteristics of matrix multiplication are as follows:

- The two matrices do not have to be the same size. The requirements are either:
  - One of the matrices is a scalar, in which case the matrix operation reduces to a scalar multiply.
  - The number of columns in the first matrix must equal the number of rows in the second. We refer to these as the **inner dimensions.** The result is a new matrix with the column count of the first matrix and the row count of the second.
- If, as illustrated, `A` is an `m` × `n` matrix and `B` is an `n` × `p` matrix, the result of `A * B` is an `m` × `p` matrix.
- The item at `(i, j)` in the result matrix is the sum of the scalar product of the `ith` row of `A` and the `jth` column of `B`.

$$A_{(mxn)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} .^{*} \; B_{(mxn)} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & & \ddots & \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} a_{11} \times b_{11} & a_{12} \times b_{12} & \cdots & a_{1n} \times b_{1n} \\ a_{21} \times b_{21} & a_{22} \times b_{22} & \cdots & a_{2n} \times b_{2n} \\ \vdots & & \ddots & \\ a_{m1} \times b_{m1} & a_{m2} \times b_{m2} & \cdots & a_{mn} \times b_{mn} \end{bmatrix}$$

Figure 12.1 *Matrix dot multiply*

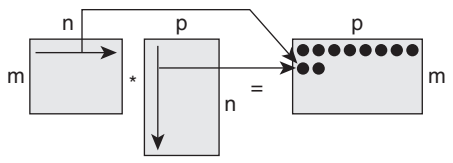


Figure 12.2 *Mechanics of matrix multiplication*

- Whereas with scalar multiplication `A .* B` gives the same result as `B .* A`, this is not the case with matrix multiplication. In fact, if `A * B` works, `B * A` will not work unless both matrices are square, and even then the results are different. (Proof of this can be derived immediately from Figure 12.3 by eliminating the third row and column and exchanging `a` for `b`. All four terms of the result of `A * B` are different from `B * A`.)
- Whereas with scalar multiplication the original array `A` can be recovered by dividing the result by `B`, this is not the case with matrix multiplication unless both matrices are square.
- The **identity matrix,** sometimes given the symbol I_n, is a square matrix with `n` rows and `n` columns that is zero everywhere except on its major diagonal, which contains the value 1. I_n has the special property that when pre-multiplied by any matrix `A` with `n` columns, or post-multiplied with any matrix `A` with `n` rows, the result is `A`. We will need this property to derive matrix division below. (The built-in function `eye(...)` generates the identity matrix.)

Figure 12.3 illustrates the mathematics for the case where a 3×2 matrix is multiplied by a 2×3 matrix, resulting in a 3×3 matrix.

$$A_{(mxn)} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{m1} & a_{m2} \end{bmatrix} * B_{(mxn)} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} (a_{11} \times b_{11} + a_{12} \times b_{21}) & (a_{11} \times b_{12} + a_{12} \times b_{22}) & (a_{11} \times b_{13} + a_{12} \times b_{23}) \\ (a_{21} \times b_{11} + a_{22} \times b_{21}) & (a_{21} \times b_{12} + a_{22} \times b_{22}) & (a_{21} \times b_{13} + a_{22} \times b_{23}) \\ (a_{31} \times b_{11} + a_{32} \times b_{21}) & (a_{31} \times b_{12} + a_{32} \times b_{22}) & (a_{31} \times b_{13} + a_{32} \times b_{23}) \end{bmatrix}$$

Figure 12.3 *Matrix multiplication*

12.2.2 Matrix Division

Matrix division is the logical process of reversing the effects of a matrix multiplication. The goal is as follows: given $A_{n\times n}$, $B_{n\times p}$, and $C_{n\times p}$, where `C = A * B`, we wish to define the mathematical equivalent of `C/A` that will result in `B`.

Since `C = A * B`, we are actually searching for some matrix $K_{n\times n}$ by which we can multiply each side of the above equation:

```
K * C = K * A * B
```

This multiplication would accomplish the division we desire if `K * A` were to result in I_n, the identity matrix. If this were the case, pre-multiplying `C` by `K` would result in I_n `* B`, or simply `B` by the definition of I_n above. The matrix `K` is referred to as the inverse of `A`, or A^{-1}. The algebra for computing this inverse is messy but well defined. In fact, Gaussian Elimination to solve linear simultaneous equations accomplishes the same thing. The MATLAB language defines both functions (`inv(A)`) and operators ("back divide," `\`) that accomplish this. However, two things should be noted:

- This inverse does not exist for all matrices—if any two rows or columns of a matrix are linearly related, the matrix is ***singular*** and does not have an inverse
- Only non-singular, square matrices have an inverse (just as a set of linear equations is soluble only if there are as many equations as there are unknown variables)

12.2.3 Matrix Exponentiation

For completeness, we mention here that matrix operations include exponentiation. However, this does not suggest that one would encounter $A_{n\times n}$`^`$B_{n\times n}$ in the scope of our applications. Rather, our usage of matrix exponentiation will be confined to A^k where `k` is any non-zero integer value. The result for positive `k` is accomplished by multiplying `A` by itself `k` times (using matrix multiplication). The result for negative `k` is accomplished by inverting A^{-k}. (There is, in fact, meaning in matrix exponentials with non-scalar exponents, but this involves advanced concepts with eigen values and eigenvectors and is beyond the scope of this text.)

12.3 Implementation

In this section, we see how MATLAB implements matrix multiplication and division. However, since applications that require matrix exponentiation A^k where k is anything but a scalar quantity are beyond the scope of this text, we will not look at its implementation in MATLAB.

12.3.1 Matrix Multiplication

Matrix multiplication is accomplished by using the "normal" multiplication symbol, as illustrated in Exercise 12.1.

In Exercise 12.1 we make the following observations:

- Entry 1 creates a 2×3 matrix, A
- Entry 2 creates a 3×1 matrix, B, a column vector
- Entry 3 indicates that this multiplication is legal because the columns in A match the rows in B
- Entry 4 shows that, likewise, it is legal to multiply a 1×2 vector by a 2×3 matrix
- Entry 5 creates an identity matrix
- Entry 6 shows that pre-multiplying A by this is legal because the inner dimensions match

Exercise 12.1 Matrix multiply

```
1. >> A = [2 5 7; 1 3 42]
A =
     2     5     7
     1     3    42
2. >> B = [1 2 3]'
B =
     1
     2
     3
3. >> A * B
ans =
    33
   133
4. >> (1:2) * A
ans =
     4    11    91
5. >> I2 = eye(2)
I2 =
     1     0
     0     1
6. >> I2 * A
ans =
     2     5     7
     1     3    42
7. >> A*I2
??? Error using ==> mtimes
Inner matrix dimensions must agree.
8. >> A*eye(3)
ans =
     2     5     7
     1     3    42
```

- Entry 7 shows that post-multiplying A by I_2 does not work because the inner dimensions do not match
- Entry 8 uses I_3 to post-multiply legally

12.3.2 Matrix Division

Matrix division is accomplished in a number of ways, all of which appear to work, but some give the wrong answer. Returning to the division problem described in Section 12.2.2, we know that `A is a` square matrix of side n, and B and C have n rows, and `C = A * B`. If we are actually given the matrices A and B, we can compute B in one of the following ways:

- `B = inv(A) * C` —using the MATLAB `inv(...)` function to compute the inverse of B
- `B = A \ C`—"back dividing" B into C to produce the same result
- `B = C / A`—apparently performing the same operation, ***but giving different answers***

Technical Insight 12.1

According to the MATLAB language help system, the third way really computes (C'\A')', which can only work if C is also square.

The order in which the matrix multiply is done affects the value of the result; therefore, care must be taken to ensure that the appropriate inversion or division is used. Study the results of Exercise 12.2 carefully.

Exercise 12.2 Matrix divide

```
>> A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> B = [1 26 24; 9 22 20; 5 12 16]
B =
     1    26    24
     9    22    20
     5    12    16
>> AB = A * B
AB =
    47   302   308
    83   272   284
    95   326   308
>> BA = B * A
BA =
   182   347   236
   218   299   248
   140   209   146
```

continued on next page

```
>> AB * inv(B)
ans =
    8    1    6
    3    5    7
    4    9    2
>> AB / B
ans =
    8    1    6
    3    5    7
    4    9    2
>> B \ BA
ans =
    8    1    6
    3    5    7
    4    9    2
>> BA / B
ans =
   -4.3000   29.2000  -15.3000
   -9.9667   27.5333   -3.9667
   -5.7333   20.7667   -8.2333
```

In Exercise 12.2 we make the following observations:

- Entries 1 and 2 construct two 3×3 matrices, `A` and `B`
- Entries 3 and 4 pre-multiply and post-multiply `B` and `A`; recall that we expect this to produce different answers
- Entry 5 shows that since we defined `inv(B)` as that function that produces the result `B*inv(B)=I`, this should produce a matrix with the same values as `A`
- Entry 6 reveals that normal division by `B` should also produce a matrix with the same values as `A`
- Entry 7 shows that back dividing `B` into `BA` should also produce a matrix equal to `A`
- Entry 8 verifies that dividing `BA` by `B` works but does not return the matrix `A`

12.4 Rotating Coordinates

A common use for matrix multiplication is for rotating coordinates in two or three dimensions. Previously we have seen the ability to rotate a complete picture by changing the viewing angle. We can move and scale items on a plot by adding coordinate offsets or multiplying them by scalar quantities. However, frequently the need arises to rotate the coordinates of a graphical object by some angle. We can use matrix multiplication to rotate individual items in a picture in two or three dimensions.

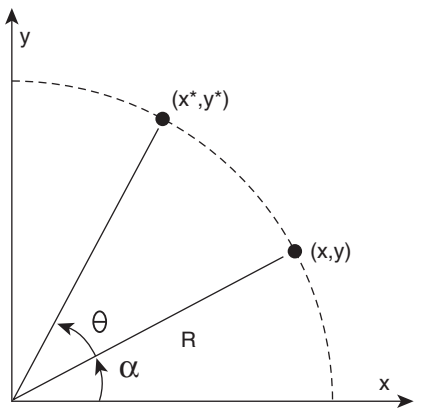


Figure 12.4 *Rotating Cartesian coordinates*

12.4.1 2-D Rotation

The mathematics implementing rotation in two dimensions is relatively straightforward, as shown in Figure 12.4. If the original point location P is (x, y) and you wish to find the point P* (x*, y*) that is the result of rotating P by the angle θ about the origin of coordinates, the mathematics are as follows:

```
x* = x cosθ - y sinθ
y* = x sinθ + y cosθ
```

which can be expressed as the matrix equation:

```
P* = A * P
```

where A is found by:

```
A = [cosθ -sinθ
     sinθ  cosθ]
```

To rotate the x-y coordinates of a graphic object in the x-y plane about some point, P, other than the origin, you would do as follows:

1. Translate the object so that P is at the origin by subtracting P from all the object's coordinates
2. Perform the rotation by multiplying each coordinate by the rotation matrix shown above
3. Translate the rotated object back to P by adding P to all the rotated coordinates

Rotating a Line Listing 12.1 illustrates a simple script to rotate a line about the origin.

Listing 12.1 Script to rotate a line

```
1. pts =    [3, 10
2.           1, 3];
3. plot(pts(1,:), pts(2,:))
4. axis ([0 10 0 10]), axis equal
5. hold on
6. for angle = 0.05:0.05:1
7.     A = [ cos(angle), -sin(angle); sin(angle), cos(angle) ];
8.     pr = A * pts;
9.     plot(pr(1,:), pr(2,:))
10. end
```

In Listing 12.1:

Lines 1 and 2: Considering the form of the rotation equations, we need to define the points where the x values are in the first row and the y values are in the second row.

Line 3: Plots the line in its original location from (3, 1) to (10, 3).

Lines 4 and 5: Fix the axes at a suitable size.

Line 6: Iterates across a selection of angles (in radians).

Line 7: Computes the rotation matrix.

Line 8: Rotates the original line by the current angle.

Line 9: Plots the rotated line.

Figure 12.5 shows the plot resulting from this script.

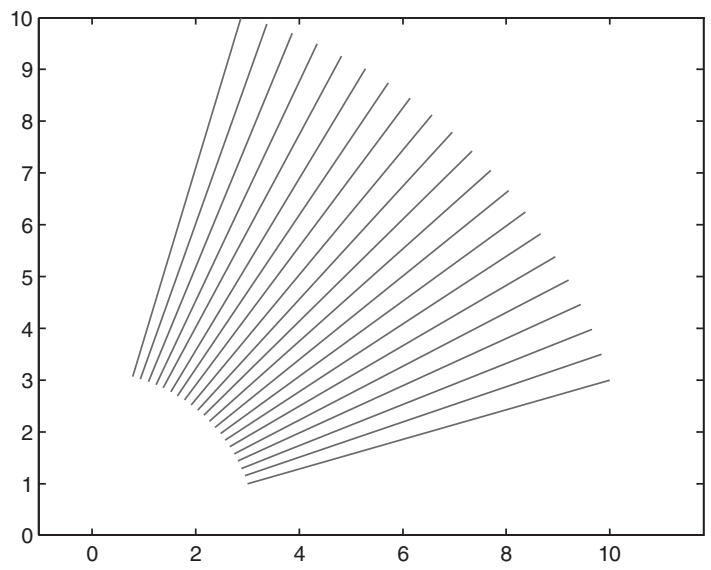


Figure 12.5 *Line rotations*

Listing 12.2 Simulating stars

```
 1. nst = 20; th = 0;
 2. for ndx = 1:nst
 3.     pos(ndx,:) = rand(1,2)*10;
 4.     scale(ndx) = rand(1,1) * .9 + .1;
 5.     rate(ndx) = rand(1,1) * 3 + 1;
 6. end
 7. while true
 8.     for str = 1:nst
 9.             star(pos(str,:), ...    % location
10.                     scale(str), ... % scale
11.                     th, ...         % basic angle
12.                     rate(str))      % angle multiplier
13.     end
14.     colormap autumn
15.     axis equal; axis([-.5 10.5 -.5 10.5])
16.     axis off; hold off
17.     th = mod(th + .1, 20*pi);
18.     pause(0.1)
19. end
```

Twinkling Stars As a second example, consider the problem of simulating twinkling stars. One way to accomplish this is to draw two triangles for each star rotating in opposite directions. The script shown in Listing 12.2 accomplishes this.

In Listing 12.2:

Line 1: Sets the number of stars and the initial rotation angle.

Lines 2–6: Establish the location, size, and rotation speed of each star.

Lines 7–19: Continue drawing until interrupted by Ctrl-C.

Lines 8–13: Draw each star at the current rotation (see Listing 12.3 for the `star(...)` function).

Line 14: Chooses a color map with yellow as the first color.

Lines 15 and 16: Show the normal display environment setup.

Line 17: Updates the angle of rotation.

Line 18: Waits 1/10 sec for the figure to be displayed. Without this, the computation would be continuous and the user would never see the result.

In Listing 12.3:

Line 1: Draws one star at location `[pt(1), pt(2)]` with scale `sc`, rotation speed `v`, and angle `th`.

Lines 2–4: Invoke the helper function `triangle(...)` to draw two triangles rotating in opposite directions.

Line 6: Function to draw one triangle with the following parameters: `up`, with values `1` for upright and `-1` for point down; `th`, the scaled

Listing 12.3 Drawing one star

```
1. function star(pt, sc, v, th)
   % draw a star at pt(1), pt(2),
   % scaled with sc, at angle v*th
2.     triangle(1, v*th, pt, sc)
3.     hold on
4.     triangle(-1, v*th, pt, sc)
5. end
6. function triangle( up, th, pt, sc )
7.     pts = [-.5   .5     0   -.5;    % x values
8.             -.289 -.289 .577 -.289]; % y values
       % rotation matrix
9.     A = sc * [cos(th), -sin(th); sin(th), cos(th)];
10.    thePts = A * pts;
11.    fill( thePts(1,:) + pt(1), ...
12.          up*thePts(2,:) + pt(2), 1);
13. end
```

rotation angle; and `pt` and `sc`, which are passed directly through from the `star(...)` function.

Lines 7 and 8: Are coordinates of an equilateral triangle.

Line 9: Computes the rotation matrix and applies the scaling factor.

Line 10: Rotates and scales the points of the triangle.

Lines 11 and 12: Call the function `fill(...)` to fill the triangle, offsetting the `x` and `y` coordinates by the original location of the triangle, and scaling `y` by the `up` multiplier to invert the triangle if necessary.

The results of this script are shown in Figure 12.6.

12.4.2 3-D Rotation

The mathematics implementing rotation in three dimensions is a natural extension of the 2-D rotation case. We present here a simple way to make this extension. The 2-D rotation in Section 12.4.1 that rotates by the angle θ in the x-y plane is actually rotating about the z-axis. If P* and P are now 3-D coordinates, we can rotate P by an angle θ about the z-axis with the equation:

$P^* = R_z * P$

where R_z is computed as

```
Rz =    [ cosθ,    -sinθ,    0
          sinθ,     cosθ,    0
             0,       0,     1]
```

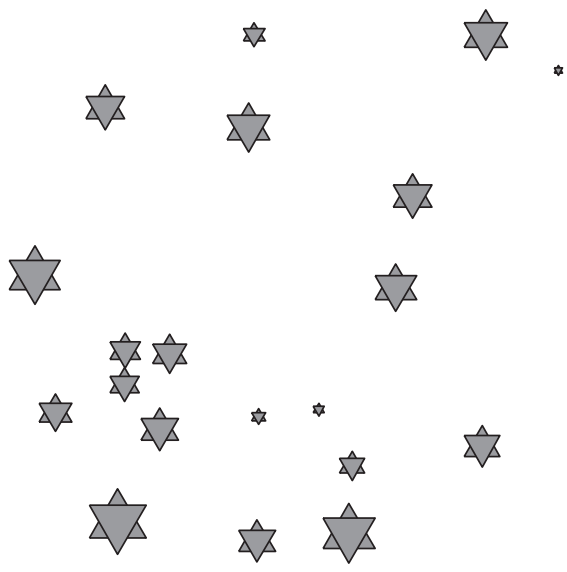


Figure 12.6 *Stars*

Similarly, we can develop matrices R_x and R_y that rotate about the x and y axes by angles ϕ and ψ, respectively.

```
Rx =    [    1,      0,         0
             0,     cosϕ,     -sinϕ
             0,     sinϕ,     cosϕ]
Ry =    [  cosψ,     0,       sinψ
             0,      1,         0
          -sinψ,     0,        cosψ]
P* = Rx * Ry * Rz * P
```

An example of a script to rotate the solid cube drawn in Chapter 11 is shown in Listing 12.4. The major problem with rotating solid objects is that the coordinates of the object are defined as arrays of points. However, the rotation matrices need each set of coordinates in single rows. To accomplish this, we will use the `reshape(...)` function to transform the coordinates to and from the row vectors necessary for the coordinate rotation.

In Listing 12.4:

Lines 1–12: Build the coordinates of the cube centered at the origin.

Lines 13 and 14: Determine the length of the linearized row vector for the `reshape(...)` function.

Lines 15 and 16: Set up the three rotation angle parameters—the initial values and the increments.

Lines 17 and 18: Repeat the drawing loop until the variable `go` is reset.

Listing 12.4 Rotating a solid cube

```
 1. xx = [ 0  0  0  0  0;
 2.       -1 -1  1  1 -1;
 3.       -1 -1  1  1 -1;
 4.        0  0  0  0  0]
 5. yy = [ 0  0  0  0  0;
 6.       -1  1  1 -1 -1;
 7.       -1  1  1 -1 -1;
 8.        0  0  0  0  0]
 9. zz = [ 1  1  1  1  1;
10.        1  1  1  1  1;
11.       -1 -1 -1 -1 -1;
12.       -1 -1 -1 -1 -1]
13. [r c] = size(xx);
14. ln = r*c; % length of reshaped vector
15. th = 0; ph = 0; ps = 0;
16. dth = 0.05; dph = 0.03; dps = 0.01;
17. go = true
18. while go
19.     surf(xx+4, yy, zz)
20.     shading interp; colormap autumn
21.     hold on; alpha(0.5)
22.     Rz = [cos(th) -sin(th)  0
23.           sin(th)  cos(th)  0
24.            0        0       1];
25.     Ry = [cos(ph)   0   -sin(ph)
26.            0        1       0
27.           sin(ph)   0 cos(ph)];
28.     Rx = [ 1        0       0
29.            0      cos(ps) -sin(ps)
30.            0      sin(ps) cos(ps)];
31.     P(1,:) = reshape(xx, 1, ln);
32.     P(2,:) = reshape(yy, 1, ln);
33.     P(3,:) = reshape(zz, 1, ln);
34.     Q = Rx*Ry*Rz*P;
35.     qx = reshape(Q(1,:), r, c);
36.     qy = reshape(Q(2,:), r, c);
37.     qz = reshape(Q(3,:), r, c);
38.     surf(qx, qy, qz)
39.     shading interp
40.     axis equal; axis off; hold off
41.     axis([-2 6 -2 2 -2 2])
42.     lightangle(40, 65); alpha(0.5)
43.     th = th+dth; ph = ph+dph; ps = ps+dps;
44.     go = ps < pi/4
45.     pause(0.03)
46. end
```

Lines 19–21: Draw one cube not rotated four units down the x-axis.
Lines 22–30: Set up the rotation matrices.
Lines 31–33: Reshape the x, y, and z arrays into linear form.
Line 34: Performs the rotation.

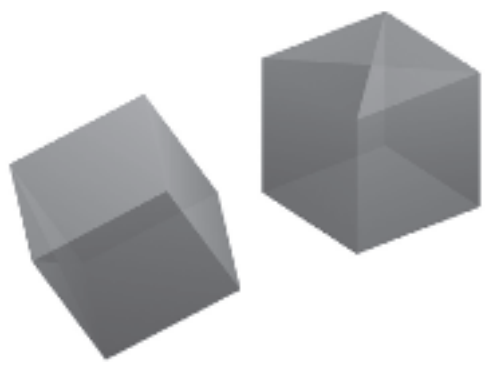


Figure 12.7 *Solid cubes*

Lines 35–37: Recover the original array shapes.
Lines 38–42: Draw the rotated cube.
Line 43: Updates the rotation angles.
Line 44: Shows the terminating condition.
Line 45: Pauses to give the figure time to draw.

The results after running this script are shown in Figure 12.7. Notice that the mechanization of the top face has caused a "wrapped parcel" effect on the light reflections off that surface.

12.5 Solving Simultaneous Linear Equations

A common use for matrix division is solving simultaneous linear equations. To be solvable, simultaneous linear equations must be expressed as N independent equations involving N unknown variables, xi. They are usually expressed in the following form:

$$
\begin{aligned}
A_{11}\,x_1 + A_{12}\,x_2 + \ldots + A_{1N}\,x_N &= c_1 \\
A_{21}\,x_1 + A_{22}\,x_2 + \ldots + A_{2N}\,x_N &= c_2 \\
\cdot \quad\quad \cdot \quad\quad\quad\quad \cdot \quad\quad \cdot & \\
\cdot \quad\quad \cdot \quad\quad\quad\quad \cdot \quad\quad \cdot & \\
A_{N1}\,x_1 + A_{N2}\,x_2 + \ldots + A_{NN}\,x_N &= c_N
\end{aligned}
$$

In matrix form, they can be expressed as follows:

$$A_{N\times N} = X_{N\times 1} = C_{N\times 1}$$

from which, since all of the values in A and C are constants, we can immediately solve for the column vector X by back division:

```
X = A\C
```

or by using the matrix inverse function:

```
X = inv(A) * C
```

12.5.1 Intersecting Lines

A typical example of a simultaneous equation problem might take the following form. Consider two straight lines on a plot with the following general form:

$A_{11}\,x + A_{12}y = c_1$
$A_{21}\,x + A_{22}y = c_2$

These lines intersect at some point P (x, y) that is the solution to both of these equations. The equations can be rewritten in matrix form as follows:

```
A * V = c
```

where `c` is the column vector $[c_1\ c_2]'$ and `v` is the required result, the column vector `[x y]'`. The solution is obtained by matrix division as follows:

```
V = A \ c
```

Recall that back divide, like the `inv(...)` function, will fail to produce a result if the matrix is singular, that is, has two rows or columns that have a linear relationship. In the specific example of two intersecting lines, this singularity occurs when the two lines are parallel, in which case there is no

Listing 12.5 Plotting line intersections

```
    % equations are y = m1 x + c1
    % y = m2 x + c2
    % in matrix form:
    % [ -m1 1; * [xp; = [c1
    % -m2 1 ] yp] c2]
 1. ax = [-0.5 6]; ay = [-4.5 18];
    % plot the two lines
 2. m1 = 3; c1 = -2;
 3. y1 = m1*ax + c1;
 4. m2 = -2; c2 = 9;
 5. y2 = m2*ax + c2;
 6. plot(ax, y1)
 7. hold on
 8. plot(ax, y2, 'b—')
    % solve for the intersection point
 9. A = [-m1 1; -m2 1];
10. c = [c1; c2];
11. P = A\c;
    % draw intersection identification lines
12. ix = P(1); iy = P(2);
13. plot([ix ix], [0 iy*1.2], 'r:')
14. plot([0 ix*1.2],[iy iy], 'r:')
    % draw the axes
15. plot(ax, [0 0], 'k');
16. axis([ax ay])
17. plot([0 0], ay, 'k');
18. legend({'Line 1','Line 2','Intersect'}, ...
19.         'Location','NorthWest' )
```

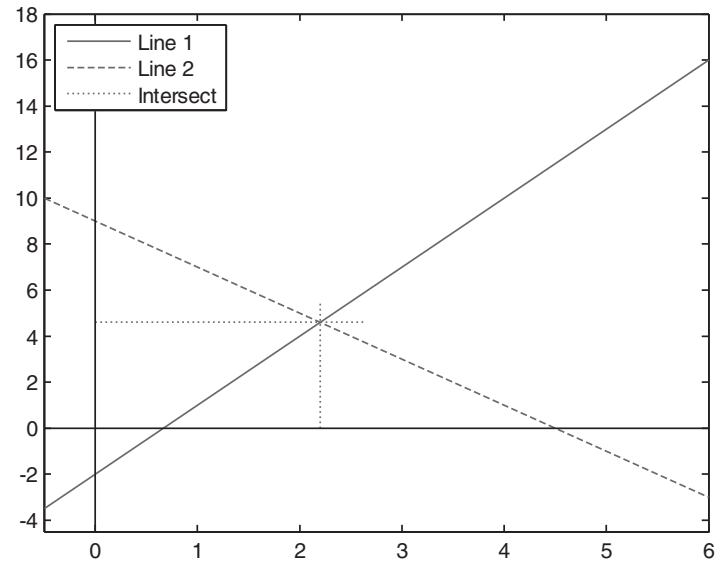


Figure 12.8 *Lines intersecting*

point of intersection. Listing 12.5 shows the solution to a pair of simultaneous equations.

In Listing 12.5:

Line 1: Sets the x and y limits of the plot.
Lines 2–8: Plot the original lines.
Line 9: Sets the simultaneous equation matrix.
Line 10: Shows the right-hand side of the equation.
Line 11: Solves the linear equations—P(1) is the x value; P(2) is the y value.
Lines 12–14: Plot the lines identifying the intersection point.
Lines 15–17: Plot the axes.
Lines 18–19: Finish the plot.

Figure 12.8 shows the result of this script.

12.6 Engineering Examples

The following examples illustrate applications of the matrix capabilities discussed in this chapter.

12.6.1 Ceramic Composition

Industrial ceramics plants require mixtures with precise formulations in order to produce products of consistent quality. For example, a factory

Table 12.1 Compound compositions

	Silica	Alumina	CaO	MgO
Feldspar	0.6950	0.1750	0.0080	0.1220
Diatomite	0.8970	0.0372	0.0035	0.0623
Magnesite	0.0670	0.0230	0.0600	0.8500
Talc	0.6920	0.0160	0.0250	0.2670

might require 100 kg of a mix consisting of 67% silica, 5% alumina, 2% calcium oxide, and 26% magnesium oxide. However, the raw material provided is not pure quantities of these materials. Rather, they are delivered as batches of material that consist of the required components in different proportions. Each batch of raw materials is analyzed to determine their composition, and we will need to do the analysis to determine the proportions of the raw materials to mix in order to accomplish the appropriate formulation. The raw materials we will use here are feldspar, diatomite, magnesite, and talc. Table 12.1 illustrates a typical analysis of the composition of these compounds.

For example, if we mixed W_f kg of feldspar, W_d kg of diatomite, W_m kg of magnesite, and W_t kg of talc, the amount of silica would be $0.695\ W_f + 0.897\ W_d + 0.067\ W_m + 0.692\ W_t$. Repeating this equation for the other components produces a matrix equation that reduces to:

```
C = A * W
```

where C is the required composition of the resulting mix, A is a 4×4 matrix showing the results of analyzing the four raw materials, and W is the proportions in which should we mix the raw material to produce the desired result. We find the appropriate amounts of the raw material by solving these equations:

```
W = A\B
```

A script that works this problem is shown in Listing 12.6.

Listing 12.6 Analyzing ceramic composition

```
1. A = [0.6950  0.8970  0.0670  0.6920
2.      0.1750  0.0372  0.0230  0.0160
3.      0.0080  0.0035  0.0600  0.0250
4.      0.1220  0.0623  0.8500  0.2670]
5. B = [67 5 2 26]'
6. W = (inv(A) * B)'
```

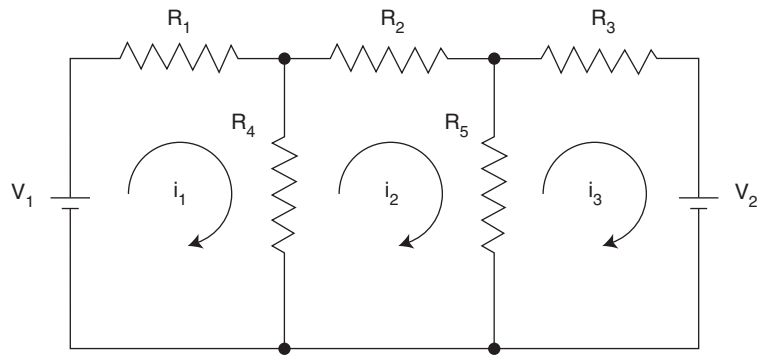


Figure 12.9 *Typical electrical circuit*

In Listing 12.6:

Lines 1–4: Matrix A is the transpose of the original data table.
Line 5: Shows the required composition in kg.
Line 6: Shows the computed weights of the raw materials in kg, which produces the following result:

```
W =
    16.0083 35.3043 15.1766 33.5108
```

12.6.2 Analyzing an Electrical Circuit

Figure 12.9 illustrates a typical electrical circuit with two voltage sources connected to five resistors with three closed loops. The voltages and resistances are given. We are asked to determine the voltage drop across R_1. Solution techniques apply Ohm's Law to the voltage drops around each closed circuit. When this technique is applied, the equations are as follows:

$$V_1 = i_1 * R_1 + (i_1 - i_2) * R_4$$
$$0 = i_2 * R_2 + (i_2 - i_3) * R_5 + (i_2 - i_1) * R_4$$
$$-V_2 = i_3 * R_3 + (i_3 - i_2) * R_5$$

When these three equations are manipulated to isolate the three currents, we have the following matrix equation:

```
V = A * I
```

which can be solved as usual by:

```
I = A \ V
```

The script to accomplish this is shown in Listing 12.7.

Listing 12.7 Analyzing an electrical circuit

```
1. R1 = 100; R2 = 200; R3 = 300;
2. R4 = 400; R5 = 500;
3. V1 = 10; V2 = 5;
4. A = [R1+R4   -R4     0
5.      -R4    R2+R4+R5 -R5
6.      0       -R5     R3+R5];
7. B = [V1; 0; -V2];
8. curr = inv(A) * B
9. fprintf('drop across R1 is %6.2f volts\n', ...
10.     curr(1) * R1 );
```

In Listing 12.7:

Lines 1–3: Set up the parameters of the problem.

Lines 4–7: Set up the coefficient matrices.

Lines 8–10: Compute and display the answers.

Running this script produces the following printout:

```
curr =    0.0283
     0.0104
     0.0003
  drop across R1 is 2.83 volts
```

Chapter Summary

This chapter presented two specialized operations performed with matrices:

- Matrix multiplication can be used for 2-D and 3-D coordinate rotations by building the appropriate rotation matrices
- Matrix division can be used for solving simultaneous equations by setting up the equations in the general form B = A * x, where the known matrix A is n × n and the known column vector B is n × 1; the unknown vector x is then found by x = A\B or x = inv(A) * B

Special Characters, Reserved Words, and Functions—2 -D

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`*`	Matrix multiplication	12.2.1
`/`	Matrix division	12.2.2
`\`	Matrix back division	12.2.2
`^`	Matrix exponentiation	12.2.3
`eye(n)`	Computes the identity matrix	12.2.1
`inv(a)`	Computes the inverse of a matrix	12.2.3
`reshape(a,r,c)`	Changes the row/column configuration of the array `a`	12.4.2

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. All MATLAB classes exhibit some form of behavioral abstraction.
2. Matrix multiplication requires that the inner dimensions match.
3. The results of `A * B` and `B * A` are identical.
4. Both `A * A`$^{-1}$ and `A`$^{-1}$ `* A` return the identity matrix.
5. Multiplying `inv(A) * B` is logically equivalent to `B / A`.
6. All sets of simultaneous linear equations can be solved by matrix inversion.

Fill in the Blanks

1. Behavioral abstraction combines _____ abstraction and _____ abstraction.
2. The result of a matrix multiplication is a new matrix with the _____ count of the first matrix and the _____ count of the second.
3. To rotate a graphic object in the x-y plane about some point, P, other than the origin, you first _____, then _____, and then _____.
4. To be soluble, simultaneous linear equations must be expressed as _____ equations involving _____variables, x_i, and _____values.

Programming Projects

1. This is a set of simple matrix manipulations.
   a. Create a five by six matrix, `A`, that contains random numbers between 0 and 10.
   b. Create a six by five matrix, `B`, that contains random numbers between 0 and 10.
   c. Find the inverse of matrix `A*B` and store it in the variable, `C`.
   d. Without iteration, create a new matrix `D` that is the same as `A` except that all values less than 5 are replaced by zero.
   e. Using iteration, create a new matrix `F` that is the same as `A` except that all values less than 5 are replaced by zero.
   f. Create a new matrix `G` that is the matrix `A` with the columns reversed.

For example:

if `A` is `[1 2 3; 3 2 5; 1 7 4]`, `G` should be
`[3 2 1; 5 2 3; 4 7 1]`

g. Find the minimum value among all the elements in `A` and store your answer in the variable `H`.

2. Imagine that world leaders have decided to come up with a single currency for the world. This new currency, called the Eullar, is defined by the following:

   Seven dollars and 3 Euros make 71 Eullars.

   One dollar and 2 Euros make 20 Eullars.

   You are a reputed economist, and your job is to find out the value of a dollar in terms of Eullars.

3. As an enthusiastic and motivated student, you decided to go out and buy plenty of pens for all your classes this semester. This spending spree unfortunately occurred before you realized your engineering classes seldom required the use of "ink." So now, you're left with four different types of pens and no receipt—you only remember the total amount you spent, and not the price of each type of pen. You decide to get together with three of your friends who coincidentally did the same thing as you, buying the same four types of pens and knowing only the total amount. Write a script to find the prices of each type of pen.

Hint:

In order to find the price of each individual pen, you could create a matrix called "pens," where each column represents a different type of pen and each row represents a different person and a column vector `totals` that contains the amount of money each of you spent on the pens.

4. Write a function called `rotateLine` that takes in two vectors, `x` and `y`, of the same length that represent a set or ordered pairs that could be used to plot a line. Your function should also take in a third parameter, `theta`, representing an angle in degrees. Your function should return `xprime` and `yprime` where `xprime` and `yprime` represent the line that is `x` and `y` rotated about the origin by the angle theta.

For example:

```
x = [ 7 7 11 11 7];
y = [-5 -9 -9 -5 -5];
[xprime yprime] = rotateLine(x, y, 90) returns
xprime = [5     9      9    5     5]
yprime = [7     7     11   11     7]
```

5. Write a function named `solveSystem` that has three inputs: two vectors consisting of the coefficients `[a b c]` of two line equations of the form `ax + by = c` and a vector of `x` values
   a. The function should output a vector giving the `x` and `y` values of the point of intersection between the two lines. If the lines are parallel, return the empty vector.
   b. Your function should also plot the two lines using the inputted vector of `x` values as `x`. In addition, on the same graph, plot the intersection point of the two lines. Make the first line blue, the second line red, and the intersection point a magenta diamond. Make sure that you label your plot appropriately.