Chapter 2

Arrays and Matrices

The heart and soul of the MATLAB software is linear algebra. In fact, "MATLAB" was originally a contraction of "matrix laboratory." More so than any other language, MATLAB encourages and expects you to make heavy use of arrays, vectors, and matrices.

Some jargon: An **array** is a collection of numbers, called **elements** or **entries**, referenced by one or more indices running over different index sets. In MATLAB, the index sets are always sequential integers starting with 1. The **dimension** of the array is the number of indices needed to specify an element. The **size** of an array is a list of the sizes of the index sets.

A **matrix** is a two-dimensional array with special rules for addition, multiplication, and other operations. It represents a mathematical linear transformation. The two dimensions are called **rows** and **columns**. A **vector** is a matrix for which one dimension has only the index 1: a **row vector** has only one row, and a **column vector** has only one column.

Although an array is more general and less mathematical than a matrix, the terms are often used interchangeably. What's more, in MATLAB there is really no formal distinction. The commands in this chapter are sorted according to the array/matrix distinction, but MATLAB will let you mix them freely as long as the syntax is defined. The idea—here, as elsewhere—is that MATLAB keeps the language simple, natural, and succinct. It's up to you to stay out of trouble.

2.1 Building arrays and matrices

The most straightforward way to construct a small array is by enclosing its elements in square brackets. Use spaces or commas to separate columns, and use semicolons or new lines to separate rows:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9

>> b = [0;1;0]
b =
     0
     1
     0
```

Information about size and dimension is stored with the array:[3]

```
>> size(A)
ans =
     3     3

>> size(b)
ans =
     3     1

>> nd = [ ndims(A), ndims(b) ]
nd =
     2     2
```

Notice that there is really no such thing as a one-dimensional array in MATLAB. Vectors are technically two-dimensional, with a trivial dimension. The distinction between row and column vectors is often, but not always, important. Table 2.1 lists more commands for obtaining information about an array.

Table 2.1. *Matrix/array information commands.*

`size`	size in each dimension
`length`	size of longest dimension (especially for vectors)
`ndims`	number of dimensions
`find`	indices of nonzero elements

[3] Because of this, array sizes are not usually passed explicitly to functions as they are in Fortran.

Arrays can be built out of other arrays, as long as the sizes are compatible:

```
>> [A b]
ans =
     1     2     3     0
     4     5     6     1
     7     8     9     0

>> [A;b]
??? Error using ==> vertcat
All rows in the bracketed expression must have the
same number of columns.

>> B = [ [1 2;3 4] [5;6] ]
B =
     1     2     5
     3     4     6
```

One important special array is the **empty matrix**, which is entered as `[]`.

Direct bracket constructions are suitable only for small matrices. For larger ones, there are many useful functions, some of which are shown in Table 2.2. For example:

```
>> [eye(3)  diag([1 2 3])]
ans =
     1     0     0     1     0     0
     0     1     0     0     2     0
     0     0     1     0     0     3

>> [zeros(2,3) ones(2,4)]
ans =
     0     0     0     1     1     1     1
     0     0     0     1     1     1     1
```

These and similar functions are sufficient to create most useful arrays, removing the need for a doubly nested loop in row and column indices. An especially important array constructor is the **colon** operator:

```
>> i1 = 1:8
i1 =
     1     2     3     4     5     6     7     8

>> i2 = 0:2:10
i2 =
     0     2     4     6     8    10
```

```
>> i3 = 1:-.5:-1
i3 =
    1.0000    0.5000         0   -0.5000   -1.0000
```

The format is `first:step:last`. The result is always a row vector, or the empty matrix if `last` is less than `first`.

Table 2.2. *Commands for building arrays and matrices.*

`:`	linearly spaced vector
`eye`	identity matrix
`zeros`	all zeros
`ones`	all ones
`diag`	diagonal matrix (or, extract a diagonal)
`toeplitz`	constant on each diagonal
`triu`	upper triangle
`tril`	lower triangle
`rand, randn`	random entries
`linspace`	linearly spaced vector
`cat`	concatenate along a given dimension
`repmat`	duplicate a vector across a dimension

2.2 Referencing elements

It is frequently necessary to access one or more of the elements of a matrix. Each dimension is given a single index or vector of indices. The result is a block extracted from the matrix. The colon is often a useful way to construct array indices. Here are some examples, using the definitions above:

```
>> A(2,3)       % single element
ans =
     6

>> b(2)         % one index for a vector
ans =
     1

>> b([1 3])     % multiple elements
ans =
     0
     0

>> A(1:2,2:3)   % a submatrix
ans =
     2     3
     5     6
```

In many situations, one wants to specify indices relative to the beginning or end of the array, or to include all possible indices. A few examples show how this is done:

```
>> B(1,2:end)          % first row, all but first column
ans =
     2     5

>> A(1:end-1,end)      % all but last row, last column
ans =
     3
     6

>> B(:,3)              % all rows of column 3
ans =
     5
     6

>> b(:,[1 1 1 1])      % multiple copies of a column
ans =
     0     0     0     0
     1     1     1     1
     0     0     0     0
```

A colon by itself as an index is understood to mean "all entries in this dimension." Observe from the last example above that an indexing result need not be a subset of the original array.

It's quite natural to access elements of a vector using a single subscript. However, any array is trivially equivalent to a vector, because it is stored linearly in memory, varying over the first dimension, then the second, and so on. (Think of the columns of an array being stacked on top of each other.) Hence a single subscript can be used for any array, automatically "flattening" it. See the documentation on `sub2ind` and `ind2sub` for more details.

```
>> A
A =
     1     2     3
     4     5     6
     7     8     9

>> A([1 2 3 4])
ans =
     1     4     7     2
```

```
>> A(:)
ans =
     1
     4
     7
     2
     5
     8
     3
     6
     9
```

The output of access by a single index is in the same shape as the index. The idiom `A(:)`, however, is always a column vector.

Subscript referencing can be used on either side of assignments. An array is resized automatically if you delete elements or make assignments outside the current size as shown below. (Any new undefined elements are set to zero.)

```
>> C = rand(2,5)
C =
    0.2610    0.6103    0.0642    0.2259    0.0736
    0.8007    0.8741    0.1214    0.6702    0.0380

>> C(1,:) = ones(1,5)
C =
    1.0000    1.0000    1.0000    1.0000    1.0000
    0.8007    0.8741    0.1214    0.6702    0.0380

>> C(:,4) = -1     % expand scalar into the submatrix
C =
    1.0000    1.0000    1.0000   -1.0000    1.0000
    0.8007    0.8741    0.1214   -1.0000    0.0380

>> C(:,2) = []     % delete elements
C =
    1.0000    1.0000   -1.0000    1.0000
    0.8007    0.1214   -1.0000    0.0380

>> C(3,3) = 3      % grow the array and fill with zeros
C =
    1.0000    1.0000   -1.0000    1.0000
    0.8007    0.1214   -1.0000    0.0380
         0         0    3.0000         0
```

Automatic resizing and scalar expansions can be highly convenient, but they can also cause hard-to-find mistakes and even subtle performance penalties (see section 6.1).

Table 2.3. *Relational operators.*

==	equal to	~=	not equal to
<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to

A different kind of array indexing is **logical indexing**. Logical indices usually arise from a **relational operator** (see Table 2.3). The result of applying a relational operator is a **logical array**, whose elements are 0 and 1 with interpretation as "false" and "true." Using a logical array as an index selects those values where the index is 1. A logical index can be used to select in just one dimension, or (more commonly) used as a single index in the flat-indexing model.

```
>> B = floor( 5*rand(2,4) )
B =
     0     1     2     1
     3     2     0     4

>> B>2
ans =
     0     0     0     0
     1     0     0     1

>> B(ans)
ans =
     3
     4

>> B(B==0) = NaN
B =
   NaN     1     2     1
     3     2   NaN     4
```

A less direct way of accomplishing the same thing is to use `find`, which returns the flat-index locations of nonzeros in any array, including a logical one:

```
>> find(B>2)
ans =
     2
     8

>> B(ans)
ans =
     3
     4
```

The disadvantage of `find` in this context is that you cannot easily refer to the complementary set of elements, whereas the ~ operator does complementation for logical arrays.

You can also create logical indices by hand, but you must explicitly cast them as such. Note carefully the difference between these two cases:

```
>> b = [1 2 3];
>> b([1 1 1])            % first element, three copies
ans =
     1     1     1

>> b(logical([1 1 1]))   % every element
ans =
     1     2     3
```

2.3 Matrix operations

The arithmetic operators +,-, *, ^ are interpreted in a matrix sense:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> D = [0 0 1; 0 0 1; 0 0 1];
>> A - 3*D
ans =
     1     2     0
     4     5     3
     7     8     6

>> Asq = A^2
Asq =
    30    36    42
    66    81    96
   102   126   150
>> b = [0; 1; -1];
>> B = [1 -1 1; 1 1 1];
>> Ab = A*b
Ab =
    -1
    -1
    -1

>> AB = A*B
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

```
>> BA = B*A
BA =
     4     5     6
    12    15    18
```

The apostrophe or single-quote mark ' produces the complex-conjugate transpose (i.e., the Hermitian or adjoint) of a matrix:

```
>> zero = A*B'-(B*A')'
zero =
     0     0
     0     0
     0     0

>> inner = b'*b
inner =
     2

>> outer = b*b'
outer =
     0     0     0
     0     1    -1
     0    -1     1
```

A special operator, the backslash \, is used to solve linear systems of equations:

```
>> A = magic(3);
>> x = A\b
x =
   -0.2083
   -0.0833
    0.2917

>> b-A*x
ans =
  1.0e-015 *

         0
    0.2220
    0.2220
```

(The last result gives 10^{-15} as a factor scaling the entire vector shown.) Mathematically, when A is invertible and B is any matrix of compatible size, `A\B` is equivalent (given exact arithmetic) to $A^{-1}B$. Similarly, `B/A` is equal to BA^{-1} if defined.

Table 2.4. *Functions from linear algebra.*

`\`	solve linear system (or least squares)
`rank`	rank
`det`	determinant
`norm`	norm (2-norm, by default)
`expm`	matrix exponential
`lu`	LU factorization (Gaussian elimination)
`qr`	QR factorization
`chol`	Cholesky factorization
`eig`	eigenvalue decomposition
`svd`	singular value decomposition

Several key functions related to linear algebra are listed in Table 2.4. There are many others. See section 7.1 for tips on using some of these functions.

2.4 Array operations

Array operations act identically on each element of an array. For arrays of identical size, the operations `+` and `-` are the same as for matrices. When the operands have different sizes, the operation typically returns an error, with one major exception: a scalar is silently "expanded" to match the size of a matching array operand:

```
>> b+2
ans =
     2
     3
     1

>> A-3
ans =
    -2    -1     0
     1     2     3
     4     5     6
```

Thus, the latter case is *not* interpreted in the mathematical matrix sense of $A - 3I$ for the identity matrix I.

The operators `*`,`'`,`^`, and `/` have matrix interpretations. To get elementwise behavior appropriate for an array, precede the operator with a dot:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> C = [1 3 -1; 2 4 0; 6 0 1];
>> A.*C                  % array multiplication
ans =
     1     6    -3
     8    20     0
    42     0     9
```

```
>> A*C                    % matrix multiplication
ans =
    23    11     2
    50    32     2
    77    53     2

>> A./A
ans =
     1     1     1
     1     1     1
     1     1     1

>> 1./A
ans =
    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
    0.1429    0.1250    0.1111

>> (C+1i)'
ans =
   1.0000 - 1.0000i  2.0000 - 1.0000i  6.0000 - 1.0000i
   3.0000 - 1.0000i  4.0000 - 1.0000i        0 - 1.0000i
  -1.0000 - 1.0000i        0 - 1.0000i  1.0000 - 1.0000i

>> (C+1i).'
ans =
   1.0000 + 1.0000i  2.0000 + 1.0000i  6.0000 + 1.0000i
   3.0000 + 1.0000i  4.0000 + 1.0000i        0 + 1.0000i
  -1.0000 + 1.0000i        0 + 1.0000i  1.0000 + 1.0000i
```

In addition, most elementary functions, such as `sin`, `exp`, etc., act elementwise:

```
>> cos(pi*C)
ans =
    -1    -1    -1
     1     1     1
     1     1    -1

>> exp(C)
ans =
    2.7183   20.0855    0.3679
    7.3891   54.5982    1.0000
  403.4288    1.0000    2.7183
```

Note that in this last case the matrix exponential function $e^C = I + C + C^2/2! + C^3/3! + \cdots$, defined for square matrices, is a completely different animal! For it, use `expm`, not `exp`.

Elementwise operators are often useful in functional expressions. Consider evaluating a Taylor polynomial approximation to $\sin(t)$:

```
>> t = (0:0.25:1)*pi/2
t =
         0    0.3927    0.7854    1.1781    1.5708

>> t - t.^3/6 + t.^5/120
ans =
         0    0.3827    0.7071    0.9245    1.0045
```

This is easier and clearer than writing a loop for the calculation. (See section 6.2.) However, because polynomials are so common, they have special status and commands of their own. A polynomial is represented by a vector of its coefficients in decreasing degree order, and it is best evaluated using `polyval`:

```
>> polyval( [1/120,0,-1/6,0,1,0], t )
ans =
         0    0.3827    0.7071    0.9245    1.0045
```

The final zero in the coefficient vector is very important here, since leaving it out would shift all of the coefficients down in degree by one. Leading zeros, on the other hand, have no effect.

Occasionally it is useful to do logical elementwise operations on arrays. The `|`, `&`, and `~` operators perform logical elementwise OR, AND, and NOT, respectively. Note, however, that a different form of OR and AND may be preferred in some statements, as explained in section 3.3.

```
>> (t>0) & (t<1)
ans =
     0     1     1     0     0
```

Another kind of array operation works in parallel along one dimension of the array, returning a result that is one dimension smaller:

```
>> C = [1 3 -1; 2 4 0; 6 0 1];

>> sum(C,1)
ans =
     9     7     0

>> sum(C,2)
ans =
     3
     6
     7
```

Other functions that behave this way are shown in Table 2.5.

Table 2.5. *Dimension-reducing functions.*

max	sum	mean	any
min	diff	median	all
sort	prod	std	

2.5 Sparse matrices

It's natural to think of a matrix as a rectangular table of numbers. However, many real-world matrices are both extremely large and very **sparse**, meaning that most entries are zero.[4] In such cases it's wasteful, or just impossible, to store every entry. Instead, one should take advantage of sparsity by storing only the nonzero entries and their locations. MATLAB has a `sparse` data type for this purpose. The `sparse` and `full` commands convert back and forth between the two available internal representations:

```
>> A = vander(1:3);
>> sparse(A)
ans =
   (1,1)         1
   (2,1)         4
   (3,1)         9
   (1,2)         1
   (2,2)         2
   (3,2)         3
   (1,3)         1
   (2,3)         1
   (3,3)         1

>> full(ans)
ans =

     1     1     1
     4     2     1
     9     3     1
```

Sparsifying a standard full matrix is usually not the right way to create a sparse matrix—you should avoid creating very large full matrices, even temporarily. One alternative is to give `sparse` the raw data required by the format. (This is the functional inverse of the `find` command.)

[4]For instance, the link structure of the Web can be described by an adjacency matrix in which a_{ij} is nonzero if page j links to page i. Obviously, any page links to a negligible fraction of all Web pages!

```
>> sparse(1:4,8:-2:2,[2 3 5 7])
ans =
   (4,2)        7
   (3,4)        5
   (2,6)        3
   (1,8)        2
```

Alternatively, you can create an empty sparse matrix with space to hold a specified number of nonzeros, and then fill it in using standard subscript assignments. Another useful sparse building command is `spdiags`, which builds along the diagonals of the matrix:

```
>> M = ones(6,1)*[-20 Inf 10]
M =
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10

>> full( spdiags( M,[-2 0 1],6,6 ) )
ans =
   Inf    10     0     0     0     0
     0   Inf    10     0     0     0
   -20     0   Inf    10     0     0
     0   -20     0   Inf    10     0
     0     0   -20     0   Inf    10
     0     0     0   -20     0   Inf
```

The `nnz` command tells how many nonzeros are in a given sparse matrix. Since it's impractical to view directly all the entries (even just the nonzeros) of a realistically sized sparse matrix, the `spy` command helps by producing a plot in which the locations of nonzeros are shown. For instance, `spy(bucky)` shows the pattern of bonds among the 60 carbon atoms in a buckyball.

MATLAB has a lot of built-in ability to work intelligently with sparse matrices. The arithmetic operators `+`, `-`, `*`, and `^` use sparse-aware algorithms and produce sparse results when applied to sparse inputs. The backslash `\` uses sparse-appropriate matrix algorithms automatically as well. There are also functions for the iterative solution of linear equations, eigenvalues, and singular values that exploit sparsity well. See section 7.2.

Exercises

2.1. Let A be a random matrix generated by `rand(8)`. Find the maximum values (a) in each column, (b) in each row, and (c) overall. Also (d) use `find` to find the row and column indices of all elements that are larger than 0.25.

2.2. A *magic square* is an $n \times n$ matrix in which each integer $1, 2, \ldots, n^2$ appears once and for which all the row, column, and diagonal sums are identical. MATLAB has a command `magic` that returns magic squares. Check its output at a few sizes and use MATLAB to verify the summation property. (The antidiagonal sum will be the trickiest. Look for help on how to "flip" a matrix.)

2.3. Are the following true or false? Assume A is a generic $n \times n$ matrix.

(a) `A^(-1)` equals `1/A`

(b) `A.^(-1)` equals `1./A`

2.4. Suppose `p` is a row vector of polynomial coefficients. What does this line do?

```
(length(p)-1:-1:0) .* p
```

2.5. (a) Look up `diag` in the online help and use it (more than once) to build the 16×16 matrix

$$D = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}.$$

(b) Now read about `toeplitz` and use it to build D.

(c) Use `toeplitz` and whatever else you need to build

$$\begin{bmatrix} 1 & 2 & 3 & \cdots & 8 \\ 0 & 1 & 2 & \cdots & 7 \\ & & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 2 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{8} \\ \frac{1}{2} & 1 & \frac{1}{2} & \cdots & \frac{1}{7} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \frac{1}{7} & \frac{1}{6} & \ddots & 1 & \frac{1}{2} \\ \frac{1}{8} & \frac{1}{7} & \cdots & \frac{1}{2} & 1 \end{bmatrix}.$$

The second case looks best in `format rat`.

2.6. Suppose A is any matrix. What does this statement do?

```
A( 1:size(A,1)+1:end )
```

2.7. (a) Suppose A is a matrix whose entries are all positive numbers. Write one line that will multiply each column of A by a scalar so that, in the resulting matrix, every column sums to 1.

(b) Try this more difficult variation: Suppose that A may have zero entries, and leave a column of A that sums to zero unchanged.

2.8. Find a MATLAB one-line expression to create the $n \times n$ matrix A satisfying

$$a_{ij} = \begin{cases} 1 & \text{if } i - j \text{ is prime,} \\ 0 & \text{otherwise.} \end{cases}$$

2.9. Suppose we represent a standard deck of playing cards by a vector `v` containing one copy of each integer from 1 to 52. Show how to "shuffle" `v` by rearranging its contents in random order. (Note: One very easy answer to this problem can be found if you look hard enough.)

2.10. Let `B=bucky`, and make a series of `spy` plots of B^2, B^3, etc. to see the phenomenon of *fill-in*: many operations, including multiplication, increase the density of nonzeros. Can you see why the (i, j) entry of B^n is the number of paths of length n between nodes i and j? What fill-in do you see with `inv(B)`?