

2.4 Introduction

Matrices are the basic elements of the MATLAB environment. A matrix is a two-dimensional array consisting of m rows and n columns. Special cases are *column vectors* ($n = 1$) and *row vectors* ($m = 1$).

In this section we will illustrate how to apply different *operations* on matrices. The following topics are discussed: vectors and matrices in MATLAB, the inverse of a matrix, determinants, and matrix manipulation.

MATLAB supports two types of operations, known as *matrix operations* and *array operations*. Matrix operations will be discussed first.

2.5 Matrix generation

Matrices are fundamental to MATLAB. Therefore, we need to become familiar with matrix generation and manipulation. Matrices can be generated in several ways.

2.5.1 Entering a vector

A vector is a special case of a matrix. The purpose of this section is to show how to create vectors and matrices in MATLAB. As discussed earlier, an array of dimension $1 \times n$ is called a *row* vector, whereas an array of dimension $m \times 1$ is called a *column* vector. The elements of vectors in MATLAB are enclosed by square brackets and are separated by spaces or by commas. For example, to enter a row vector, `v`, type

```
>> v = [1 4 7 10 13]
v =
     1     4     7    10    13
```

Column vectors are created in a similar way, however, semicolon (;) must separate the components of a column vector,

```
>> w = [1;4;7;10;13]
w =
     1
     4
     7
    10
    13
```

On the other hand, a *row* vector is converted to a *column* vector using the *transpose* operator. The *transpose* operation is denoted by an apostrophe or a single quote (').

```
>> w = v'
w =
     1
     4
     7
    10
    13
```

Thus, $v(1)$ is the first element of vector \mathbf{v} , $v(2)$ its second element, and so forth.

Furthermore, to access *blocks* of elements, we use MATLAB's colon notation ($:$). For example, to access the first three elements of \mathbf{v} , we write,

```
>> v(1:3)
ans =
     1     4     7
```

Or, all elements from the third through the last elements,

```
>> v(3:end)
ans =
     7    10    13
```

where **end** signifies the *last* element in the vector. If \mathbf{v} is a vector, writing

```
>> v(:)
```

produces a column vector, whereas writing

```
>> v(1:end)
```

produces a row vector.

2.5.2 Entering a matrix

A matrix is an array of numbers. To type a matrix into MATLAB you must

- begin with a square bracket, `[`
- separate elements in a row with spaces or commas `(,)`
- use a semicolon `(;)` to separate rows
- end the matrix with another square bracket, `]`.

Here is a typical example. To enter a matrix **A**, such as,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (2.1)$$

type,

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

MATLAB then displays the 3×3 matrix as follows,

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

Note that the use of semicolons (;) here is different from their use mentioned earlier to suppress output or to write multiple commands in a single line.

Once we have entered the matrix, it is automatically stored and remembered in the *Workspace*. We can refer to it simply as matrix **A**. We can then view a particular element in a matrix by specifying its location. We write,

```
>> A(2,1)  
ans =  
      4
```

A(2,1) is an element located in the second row and first column. Its value is 4.

2.5.3 Matrix indexing

We select elements in a matrix just as we did for vectors, but now we need two indices. The element of row i and column j of the matrix **A** is denoted by **A(i,j)**. Thus, **A(i,j)** in MATLAB refers to the element A_{ij} of matrix **A**. The *first* index is the *row* number and the *second* index is the *column* number. For example, **A(1,3)** is an element of *first* row and *third* column. Here, **A(1,3)=3**.

Correcting any entry is easy through indexing. Here we substitute **A(3,3)=9** by **A(3,3)=0**. The result is

```
>> A(3,3) = 0  
A      =  
      1      2      3  
      4      5      6  
      7      8      0
```

Single elements of a matrix are accessed as $A(i, j)$, where $i \geq 1$ and $j \geq 1$. Zero or negative subscripts are not supported in MATLAB.

2.5.4 Colon operator

The colon operator will prove very useful and understanding how it works is the key to efficient and convenient usage of MATLAB. It occurs in several different forms.

Often we must deal with matrices or vectors that are too large to enter one element at a time. For example, suppose we want to enter a vector x consisting of points $(0, 0.1, 0.2, 0.3, \dots, 5)$. We can use the command

```
>> x = 0:0.1:5;
```

The row vector has 51 elements.

2.5.5 Linear spacing

On the other hand, there is a command to generate linearly spaced vectors: `linspace`. It is similar to the colon operator `(:)`, but gives direct control over the number of points. For example,

```
y = linspace(a,b)
```

generates a row vector y of 100 points linearly spaced between and including a and b .

```
y = linspace(a,b,n)
```

generates a row vector y of n points linearly spaced between and including a and b . This is useful when we want to divide an interval into a number of subintervals of the same length. For example,

```
>> theta = linspace(0,2*pi,101)
```

divides the interval $[0, 2\pi]$ into 100 equal subintervals, then creating a vector of 101 elements.

2.5.6 Colon operator in a matrix

The colon operator can also be used to pick out a certain row or column. For example, the statement $A(m:n, k:l)$ specifies rows m to n and column k to l . Subscript expressions refer to portions of a matrix. For example,

```
>> A(2,:)
ans =
     4     5     6
```

is the second row elements of A.

The colon operator can also be used to extract a sub-matrix from a matrix A.

```
>> A(:,2:3)
ans =
     2     3
     5     6
     8     0
```

A(:,2:3) is a sub-matrix with the last two columns of A.

A row or a column of a matrix can be deleted by setting it to a *null* vector, [].

```
>> A(:,2)=[]
ans =
     1     3
     4     6
     7     0
```

2.5.7 Creating a sub-matrix

To extract a *submatrix* B consisting of rows 2 and 3 and columns 1 and 2 of the matrix A, do the following

```
>> B = A([2 3],[1 2])
B =
     4     5
     7     8
```

To interchange rows 1 and 2 of A, use the vector of row indices together with the colon operator.

```
>> C = A([2 1 3], :)
C =
     4     5     6
     1     2     3
     7     8     0
```

It is important to note that the *colon operator* (:) stands for *all columns* or *all rows*. To create a vector version of matrix A, do the following

```
>> A(:)
ans =
     1
     2
     3
     4
     5
     6
     7
     8
     0
```

The submatrix comprising the intersection of rows **p** to **q** and columns **r** to **s** is denoted by **A(p:q,r:s)**.

As a special case, a colon (:) as the row or column specifier covers all entries in that row or column; thus

- **A(:,j)** is the **j**th column of **A**, while
- **A(i,:)** is the **i**th row, and
- **A(end,:)** picks out the last row of **A**.

The keyword **end**, used in **A(end,:)**, denotes the last index in the specified dimension. Here are some examples.

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

```
>> A(2:3,2:3)
ans =
     5     6
     8     9
```

```
>> A(end:-1:1,end)
ans =
     9
     6
     3
```

```
>> A([1 3],[2 3])
ans =
     2     3
     8     9
```

2.5.8 Deleting row or column

To delete a row or column of a matrix, use the *empty vector* operator, `[]`.

```
>> A(3,:) = []
A =
     1     2     3
     4     5     6
```

Third row of matrix A is now deleted. To restore the third row, we use a technique for creating a matrix

```
>> A = [A(1,:);A(2,:);[7 8 0]]
A =
     1     2     3
     4     5     6
     7     8     0
```

Matrix A is now restored to its original form.

2.5.9 Dimension

To determine the *dimensions* of a matrix or vector, use the command `size`. For example,

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

means 3 rows and 3 columns.

Or more explicitly with,

```
>> [m,n]=size(A)
```

2.5.10 Continuation

If it is not possible to type the entire input on the same line, use consecutive periods, called an ellipsis ..., to signal continuation, then continue the input on the next line.

```
B = [4/5      7.23*tan(x)      sqrt(6); ...
     1/x^2    0                3/(x*log(x)); ...
     x-7      sqrt(3)          x*sin(x)];
```

Note that *blank* spaces around +, −, = signs are optional, but they improve readability.

2.5.11 Transposing a matrix

The *transpose* operation is denoted by an apostrophe or a single quote ('). It flips a matrix about its main diagonal and it turns a row vector into a column vector. Thus,

```
>> A'
ans =
     1     4     7
     2     5     8
     3     6     0
```

By using linear algebra notation, the transpose of $m \times n$ real matrix **A** is the $n \times m$ matrix that results from interchanging the rows and columns of **A**. The transpose matrix is denoted A^T .

2.5.12 Concatenating matrices

Matrices can be made up of sub-matrices. Here is an example. First, let's recall our previous matrix **A**.

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

The new matrix **B** will be,

```
>> B = [A 10*A; -A [1 0 0; 0 1 0; 0 0 1]]
B =
     1     2     3    10    20    30
```

4	5	6	40	50	60
7	8	9	70	80	90
-1	-2	-3	1	0	0
-4	-5	-6	0	1	0
-7	-8	-9	0	0	1

2.5.13 Matrix generators

MATLAB provides functions that generates elementary matrices. The matrix of zeros, the matrix of ones, and the identity matrix are returned by the functions `zeros`, `ones`, and `eye`, respectively.

Table 2.4: Elementary matrices

<code>eye(m,n)</code>	Returns an m-by-n matrix with 1 on the main diagonal
<code>eye(n)</code>	Returns an n-by-n square identity matrix
<code>zeros(m,n)</code>	Returns an m-by-n matrix of zeros
<code>ones(m,n)</code>	Returns an m-by-n matrix of ones
<code>diag(A)</code>	Extracts the diagonal of matrix A
<code>rand(m,n)</code>	Returns an m-by-n matrix of random numbers

For a complete list of *elementary matrices* and *matrix manipulations*, type `help elmat` or `doc elmat`. Here are some examples:

```
1.      >> b=ones(3,1)
      b
      =
      1
      1
      1
```

Equivalently, we can define `b` as `>> b=[1;1;1]`

```
2.      >> eye(3)
      ans
      =
      1     0     0
      0     1     0
      0     0     1
```

```
3.      >> c=zeros(2,3)
      c
      =
      0     0     0
```

0 0 0

In addition, it is important to remember that the three elementary operations of *addition* (+), *subtraction* (−), and *multiplication* (*) apply also to matrices whenever the dimensions are *compatible*.

Two other important matrix generation functions are **rand** and **randn**, which generate matrices of (pseudo-)random numbers using the same syntax as **eye**.

In addition, matrices can be constructed in a block form. With **C** defined by **C** = [1 2; 3 4], we may create a matrix **D** as follows

```
>> D = [C zeros(2); ones(2) eye(2)]
D =
     1     2     0     0
     3     4     0     0
     1     1     1     0
     1     1     0     1
```

2.5.14 Special matrices

MATLAB provides a number of special matrices (see Table 2.5). These matrices have interesting properties that make them useful for constructing examples and for testing algorithms. For more information, see MATLAB documentation.

Table 2.5: Special matrices

hilb	Hilbert matrix
invhilb	Inverse Hilbert matrix
magic	Magic square
pascal	Pascal matrix
toeplitz	Toeplitz matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

2.6 Exercises

NOTE: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.

Chapter 3

Array operations and Linear equations

3.1 Array operations

MATLAB has two different types of arithmetic operations: matrix arithmetic operations and array arithmetic operations. We have seen matrix arithmetic operations in the previous lab. Now, we are interested in array operations.

3.1.1 Matrix arithmetic operations

As we mentioned earlier, MATLAB allows arithmetic operations: $+$, $-$, $*$, and $^$ to be carried out on matrices. Thus,

$A+B$ or $B+A$	is valid if A and B are of the same size
$A*B$	is valid if A 's number of column equals B 's number of rows
A^2	is valid if A is square and equals $A*A$
$\alpha*A$ or $A*\alpha$	multiplies each element of A by α

3.1.2 Array arithmetic operations

On the other hand, array arithmetic operations or *array operations* for short, are done *element-by-element*. The period character, $.$, distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition ($+$) and subtraction ($-$), the character pairs $(.+)$ and $(.-)$ are not used. The list of array operators is shown below in Table 3.2. If A and B are two matrices of the same size with elements $A = [a_{ij}]$ and $B = [b_{ij}]$, then the command

<code>.*</code>	Element-by-element multiplication
<code>./</code>	Element-by-element division
<code>.^</code>	Element-by-element exponentiation

Table 3.1: Array operators

```
>> C = A.*B
```

produces another matrix \mathbf{C} of the same size with elements $c_{ij} = a_{ij}b_{ij}$. For example, using the same 3×3 matrices,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

we have,

```
>> C = A.*B
C      =
    10    40    90
   160   250   360
   490   640   810
```

To raise a scalar to a power, we use for example the command `10^2`. If we want the operation to be applied to each element of a matrix, we use `.^2`. For example, if we want to produce a new matrix whose elements are the square of the elements of the matrix \mathbf{A} , we enter

```
>> A.^2
ans     =
     1     4     9
    16    25    36
    49    64    81
```

The relations below summarize the above operations. To simplify, let's consider two vectors U and V with elements $U = [u_i]$ and $V = [v_j]$.

$$\begin{array}{lll} U.*V & \text{produces} & [u_1v_1 \ u_2v_2 \ \dots \ u_nv_n] \\ U./V & \text{produces} & [u_1/v_1 \ u_2/v_2 \ \dots \ u_n/v_n] \\ U.^V & \text{produces} & [u_1^{v_1} \ u_2^{v_2} \ \dots \ u_n^{v_n}] \end{array}$$

OPERATION	MATRIX	ARRAY
Addition	+	+
Subtraction	−	−
Multiplication	*	.*
Division	/	./
Left division	\	.\
Exponentiation	^	.^

Table 3.2: Summary of matrix and array operations

3.2 Solving linear equations

One of the problems encountered most frequently in scientific computation is the solution of systems of simultaneous linear equations. With matrix notation, a system of simultaneous linear equations is written

$$Ax = b \quad (3.1)$$

where there are as many equations as unknown. A is a given square matrix of order n , b is a given column vector of n components, and x is an unknown column vector of n components.

In linear algebra we learn that the solution to $Ax = b$ can be written as $x = A^{-1}b$, where A^{-1} is the inverse of A .

For example, consider the following system of linear equations

$$\begin{cases} x + 2y + 3z &= 1 \\ 4x + 5y + 6z &= 1 \\ 7x + 8y &= 1 \end{cases}$$

The coefficient matrix A is

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{and the vector} \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

With matrix notation, a system of simultaneous linear equations is written

$$Ax = b \quad (3.2)$$

This equation can be solved for x using linear algebra. The result is $x = A^{-1}b$.

There are typically two ways to solve for x in MATLAB:

1. The first one is to use the matrix inverse, `inv`.

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> b = [1; 1; 1];
>> x = inv(A)*b
x      =
    -1.0000
     1.0000
    -0.0000
```

2. The second one is to use the *backslash* (`\`) operator. The numerical algorithm behind this operator is computationally efficient. This is a numerically reliable way of solving system of linear equations by using a well-known process of Gaussian elimination.

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> b = [1; 1; 1];
>> x = A\b
x      =
    -1.0000
     1.0000
    -0.0000
```

This problem is at the heart of many problems in scientific computation. Hence it is important that we know how to solve this type of problem efficiently.

Now, we know how to solve a system of linear equations. In addition to this, we will see some additional details which relate to this particular topic.

3.2.1 Matrix inverse

Let's consider the same matrix A .

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Calculating the inverse of A manually is probably not a pleasant work. Here the hand-calculation of A^{-1} gives as a final result:

$$A^{-1} = \frac{1}{9} \begin{bmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{bmatrix}$$

In MATLAB, however, it becomes as simple as the following commands:

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> inv(A)
ans =
    -1.7778    0.8889   -0.1111
     1.5556   -0.7778    0.2222
    -0.1111    0.2222   -0.1111
```

which is similar to:

$$A^{-1} = \frac{1}{9} \begin{bmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{bmatrix}$$

and the determinant of A is

```
>> det(A)
ans =
    27
```

For further details on applied numerical linear algebra, see [10] and [11].

3.2.2 Matrix functions

MATLAB provides many matrix functions for various matrix/vector manipulations; see Table 3.3 for some of these functions. Use the online help of MATLAB to find how to use these functions.

det	Determinant
diag	Diagonal matrices and diagonals of a matrix
eig	Eigenvalues and eigenvectors
inv	Matrix inverse
norm	Matrix and vector norms
rank	Number of linearly independent rows or columns

Table 3.3: Matrix functions

3.3 Exercises

NOTE: Due to the teaching class during this Fall Quarter 2005, the *problems* are *temporarily* removed from this section.