CHAPTER 2

Array Based Computing

2.1 Introduction

Matrices have become an integrated part of numerical computation for dealing with large quantities of data. For a two-dimensional matrix, elements have unique row and column indices through which you can access them. Rows and columns can be attributed to different properties under study. For example, if you measure the temperature and pressure at four corners of the square, the *x, y* coordinates associated with the corner points can be assigned to row and column numbers. Now the experimental data can be simply represented as a matrix. In this way, you can fit data for two properties as a matrix and then use these matrices for numerical calculations.

As an example, suppose an element of a row is defined as 1 if a compound is a conductor, it's 2 if it is a semiconductor, and it's 3 if it is an insulator. Then, a row vector (a matrix composed of only one row) `[1 0 0 3 2 1 3 0 1 0 3 2 1]` has information about 13 compounds. In electrical conductivity experiments, this row vector (a 13×1 matrix) can be utilized as input. In this way, you need to model the system in terms of matrix formulation to be solved using MATLAB.

© Sandeep Nagar 2017
S. Nagar, *Introduction to MATLAB for Engineers and Scientists*,
https://doi.org/10.1007/978-1-4842-3189-0_2

MATLAB defines a data object for dealing with matrices. They are called *arrays*. Using different properties of this object, you can define various kinds of matrices. Built-in functions for matrix operations make it easier for a programmer to deal with large amounts of data by arranging it as a matrix in the desired format and performing array operations. This chapter explores the various options for defining and manipulating arrays.

Since MATLAB was made for matrix manipulation, it has a large set of built-in functions and a robust environment to define and work with matrices.

2.2 Arrays and Vectors

Instead of just pointing to a single number, a variable name can also point to a sequential set of numbers, called an *array*. The following example shows how this can be achieved:

```
>> a = [1,2,3,4,5]
a =

1   2   3   4   5

>> a1 = [10,11,12,13,14]
a1 =

10   11    12   13    14
>> matrix22 = [1,2;3,4]
matrix22 =

1     2
3     4
```

```
>> matrix33 = [1,2,3;4,5,6;7,8,9]
matrix33 =

1     2     3
4     5     6
7     8     9
>> size(a)
ans =

1   5

>> size(matrix22)
ans =

2   2

>> size(matrix33)
ans =

3   3
```

As seen in the example code, an array can be understood as a matrix consisting of rows and columns. Thus, you can make a desired sized matrix. For example, `matrix22` is a 2×2 and `matrix33` is a 3×3 matrix, whereas `a` is a 1×5 matrix. The first number listed while defining the size indicates the number of rows, whereas the second number indicates the number of columns. It is also important to note that the comma `(,)` operator operates by defining the *next element* in the same row, whereas the semicolon `(;)` operator defines the numbers in the next line/row. A matrix is defined within the brackets of the type `[ ]` (commonly called *square brackets*).

If the number of elements in each row/column do not match, you get an error message:

```
>> right33 = [1,2,3;4,5,6;7,8,9]
right33 =

1   2   3
4   5   6
7   8   9

>> wrong33 = [2,3;4,5,6;7,8,9]
Dimensions of matrices being concatenated are
not consistent.
>> wrong33 = [1,2,3;4,5,6;8,9]
Dimensions of matrices being concatenated are
not consistent.
```

2.3 Creating Arrays from Other Arrays

Multi-dimensional arrays can be created from other multi-dimensional arrays too, as explained here:

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

a =

1     2     3
4     5     6

>> B = [(1:3);(4:6);(7:9)]

```

```
B =

1     2     3
4     5     6
7     8     9

>> c = [a;B]

c =

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

>>>c = [a,B]

Error using horzcat
Dimensions of matrices being concatenated are not
consistent.
```

Here, the matrix a has elements 1, 2, 3 in the first row. Then a row separator (;) defines the next row of elements as 4, 5, 6. Similarly, matrix B has the rows defined by these commands:

- (1 : 3) results in (1, 2, 3)
- (4 : 6) results in (4, 5, 6)
- (7 : 9) results in (178, 9)

Note the MATLAB variable names are case sensitive, so a is not the same as A. A new matrix called c is created by *vertically concatenating* the

matrices a and B. The resultant matrix c is made of elements of a stacked on top of elements of B. The c=[a, B] command yields an error because the dimensions of a and B are not consistent for horizontal concatenation.

Horizontal concatenation can instead be easily performed in the following ways in this example:

```
>> a = 1:3

a =

1     2     3

>> A = [a,a]

A =

1     2     3     1     2     3
```

For multidimensional arrays, use this code:

```
>> a = 1:4

a =

1     2     3     4

>> A = [a;a]

A =

1     2     3     4
1     2     3     4
```

```
>> AA = [A,A]

AA =

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

2.3.1 Appending Rows and Columns

When an entire row or column of a matrix needs to be appended, you must consider only one thing—the size of new matrix must match the row and column requirements. As an example, define an array `A, B, D` with sizes (2×2), (1×2), and (2×1), respectively. The row matrix `B` can be inserted as a row of `A` and the column matrix `D` can be inserted as a row of `A`, as shown here:

```
>> A = [1,2;3,4]

A =

1     2
3     4

>> B = [5,6]

B =

5     6

>> size(A)

```

```
ans =

2     2

>> size(B)

ans =

1     2

>> C = [A;B]

C =

1     2
3     4
5     6

>> size(C)

ans =

3     2

>> D = [5;6]

D =

5
6
```

```
>> size(D)

ans =

2     1

>> E = [A,D]

E =

1     2     5
3     4     6

>> size(E)

ans =

2     3
```

2.3.2 Deleting a Row and/or Column of a Matrix

Rows and columns can be deleted by assigning null matrices [ ] to them. For example, `(1,:)=[]` deletes the first row and `(:,1)=[]` deletes the first column of a matrix, as shown here:

```
>> A = rand(3,3)

A =

0.8147    0.9134    0.2785
0.9058    0.6324    0.5469
0.1270    0.0975    0.9575

```

```
>> A(1,:) =[]

A =

0.9058    0.6324    0.5469
0.1270    0.0975    0.9575

>> A(:,1) =[]

A =

0.6324    0.5469
0.0975    0.9575
```

2.3.3 Concatenation Along a Dimension

Concatenation of two matrices along a dimension can be obtained using `cat(dim, A, B, ...)`, where `dim` presents the dimension and `A` and `B` are the input matrices. Its usage is shown here:

```
>> A = [1,2;3,4]
A =
1    2
3    4
>> B = [5,6;7,8]
B =
5    6
7    8
>> cat(1,A,B)
ans =
1    2
3    4
```

```
13  5    6
14  7    8
15  >> cat(2,A,B)
16  ans =
17  1    2    5    6
18  3    4    7    8
19  >> C = cat(3,A,B)
20  ans(:,:,1) =
21  1    2
22  3    4
23  ans(:,:,2) =
24  5    6
25  7    8
26  >>> size(C)
27  ans =
28  2    2    2
```

When `cat(1,A,B)` is entered at the command prompt, `A` and `B` are concatenated row-wise and `cat(2,A,B)` performs concatenation column-wise. In case of `cat(3,A,B)`, a new matrix is created whose first element of the third dimension is the matrix `A` and the second element is the matrix `B`.

2.3.4 Selecting the Data Type of Elements

Elements of an array can be any data type, as explained in Chapter 1. All elements of an array can be set to a particular data type using the commands shown here:

```
1  >> x = uint32([1,65535])
2  x =
3
4  1x2 uint32 row vector
5
```

```
 6  1   65535
 7
 8  >> x = uint64([1,65535])
 9  x =
10
11  1x2 uint64 row vector
12
13  1   65535
14
15  >> x = int16([1,65535])
16  x =
17
18  1x2 int6 row vector
19
20  1   32767
21
22  >> x = int32([1,65535])
23  x =
24
25  1x2 int32 row vector
26
27  1   65535
28
29  >> x = int64([1,65535])
30  x =
31
32  1x2 int64 row vector
33
34  1   65535
35
```

```
>> x = single([1,65535])
x =

1x2 single row vector

1       65535

>> x = double([1,65535])
x =

1       65535

>> x = single([1.0,65535e10])
x =

1x2 single row vector

1.0e+14*

0.0000    6.5535

>> x = double([1.0,65535e10])
x =

1.0e+14*

0.0000    6.5535
```

Line 15 shows that if the element is set to `int16`, then it can store a maximum value of 32767, regardless of being commanded to store a value bigger than that. Hence, it becomes supremely important to understand

the data type of the elements beforehand, in order to avoid errors in numerical calculations. Keep in mind that storing very small numbers in larger numbers of bits is a waste of memory. (Line 46 displays that the number 1, which is stored as a *double precision floating point number,* occupies 64 bits, where essentially 63 bits except the last one are all zeros!)

2.4 Arithmetic Operations on Arrays

Operating on arrays involves two aspects:

- Operating on two or more arrays
- Element-wise operations

All arithmetic operators (such as +, -, *, /, %, ^, etc.) can be used in both cases. When you need to do element-wise operation, then a `.` (dot) is placed before the operator. The element-wise operators become `.+`, `.-`, `.*`, `./`, `.%`, and `.^`. This will become more clear in following example.

```
 1  >> a = [1,2;3,4]
 2  a =
 3
 4  1    2
 5  3    4
 6
 7  >> b = [5,6;7,8]
 8  b =
 9
10  5    6
11  7    8
12
```

```
>> a+b
ans =

6    8
10   12

>> 2.+a
ans =

3    4
5    6

>> -10.+b
ans =

-5   -4
-3   -2
```

When a and b are matrices to be added/subtracted, their elements are added/subtracted to elements in the same position. For this reason, the size of the two matrices should be same. On the other hand, when you write `2.+a`, you add the number 2 to each of the elements individually. This can be done regardless of the size and is implemented uniformly on all the elements of the matrix.

2.5 Built-In Functions

A host of built-in functions provide facilities to calculate properties of arrays for quick computation. This includes:

- Summing all elements using `sum()` function.
- Finding the product of all elements of an array using `prod()`.

- Finding the length of array using `length()`.
- Finding the mean of array elements using the `mean()` function.
- Finding the maximum and minimum amongst an element of an array using `max()` and `min()` of an array.
- Finding a particular element as per a logical expression using the `find()` function.
- The rounding elements are as follows:
  - Rounding the elements of an array to the nearest integer toward zero using the `fix()` function.
  - Rounding the elements of an array to the nearest integer toward $-\infty$ using the `floor()` function.
  - Rounding the elements of an array to the nearest integer toward $+\infty$ using the `ceil()` function.
  - Rounding the elements of an array to the nearest integer using the `rounding()` function.
- Sorting the elements of an array using `sort()` in ascending or descending order.

Their usage is demonstrated here:

```
>> A = 1:5
A =
1     2     3     4     5
>> sum(A)
ans =
15
>> prod(A)
ans =
120
```

```
>> length(A)
ans =
5
>> mean(A)
ans =
3
>> max(A)
ans =
5
>> min(A)
ans =
1
>> find(A>4)
ans =
5
>> find(A<4)
ans =
1     2     3
>> A= -1.1:0.5:1.1
A =
-1.1000   -0.6000   -0.1000    0.4000    0.9000
>> fix(A)
ans =
-1     0     0     0     0
>> floor(A)
ans =
-2    -1    -1     0     0
>> ceil(A)
ans =
-1    0      0     1     1
```

```
>> round(A)
ans =
-1   -1    0    0    1
>> A = [2,4.4,2,7,0,-2]
A =
2.0000    4.4000    2.0000    7.0000         0   -2.0000
>> sort(A,'ascend')
ans =
-2.0000         0    2.0000    2.0000    4.4000    7.0000
>> sort(A,'descend')
ans =
7.0000    4.4000    2.0000    2.0000         0   -2.0000
```

2.6 Matrix Algebra

Arithmetic on matrices can be placed into two classes:

- Algebraic operations (covered in Chapter 2)
- Matrix operations

2.6.1 Algebraic Operations on Matrices

Algebraic operations on matrices involve *element-wise operations*. For example:

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

```
>> a+2
ans =
3     4
5     6
7     8
```

Note that a defines a 3×2 matrix so the a+2 command performs element-wise addition of a with a number 2. Computationally, this is done by creating a 3×2 matrix with all its elements as the number 2 and adding them.

Similarly, some other operations are shown here:

```
>> 2*a
ans =
2     4
6     8
10    12
>> 2-a
ans =
1     0
-1    -2
-3    -4
>> a-2
ans =
-1    0
1     2
3     4
>> a/2
ans =
0.5000    1.0000
1.5000    2.0000
2.5000    3.0000
```

The problem starts with other arithmetic operations. For example, when we want to calculate a^2, this would mean multiplying a with itself, i.e., matrix multiplication. This requires either a square matrix or the inner dimensions to be similar because a matrix of dimension $n \times m$ can only be multiplied with `m×t` and the resultant matrix is of the dimension `n×t`. Hence, the command `a^(2)` will result in an error message, as shown here:

```
1  >> a^2
2  Error using ^
3  Input s must be a scalar and a square matrix.
4  To compute elementwise POWER, use POWER (.^) instead.
```

If we wanted to calculate element-wise squares of matrix a then the last line of the `error message` comes to the rescue. Adding a dot operator to a power operator (`.^`) will direct MATLAB to perform the same operation element-wise.

```
1  >> a.^2
2  ans =
3  1    4
4  9   16
5  25  36
```

On the other hand, multiplication of two matrices is the domain of matrix algebra, discussed next.

2.6.2 Matrix Operations on Matrices

Those who are familiar with matrix algebra know that matrix multiplication and division are not straightforward tasks. A `m×n` matrix can only be multiplied by a `n×t` matrix, which results in `a×c` matrix. This is performed by multiplying elements of rows with elements of columns to get new elements.

```
>> a = rand(2,3)

a =

0.8147     0.1270     0.6324
0.9058     0.9134     0.0975

>> b = rand(3,4)

b =

0.2785     0.9649     0.9572     0.1419
0.5469     0.1576     0.4854     0.4218
0.9575     0.9706     0.8003     0.9157
>> c = rand(2,3)

c =

0.7922     0.6557     0.8491
0.9595     0.0357     0.9340

>> a.*c

ans =

0.6454     0.0833     0.5370
0.8691     0.0326     0.0911
```

Here, the matrices a, b, and c are defined using the rand function (which generates uniformly distributed random numbers between 0 and 1).

Now, `a*b` performs matrix multiplication, whereas `a.*c` performs element-wise multiplication. The requirements for both are as follows:

- For matrix multiplication, the inner dimensions must match.
- For element-wise multiplication, all dimensions must match.

Transpose

A single hash mark (`'`), also called an apostrophe, transposes a matrix (rows become columns and vice versa). Performing division on a matrix involves *matrix inversion*.

```
>> a

a =

1    2
3    4
5    6
>> pinv(a)
ans =

-1.3333    -0.3333    0.6667
1.0833     0.3333   -0.4167
>> b

b =

5    6
7    8
```

```
>> pinv(b)

ans =

-4.0000    3.0000
3.5000   -2.5000
```

Inverse

The inverse of a matrix a, denoted by a^{-1}, is a matrix such that

$$a * a^{-1} = I$$

where I is an *identity matrix*. If the given matrix is a square matrix, then the function `inv()` can be used; otherwise, the function `pinv()` is used. Examples are given here:

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

a =

1     2
3     4
5     6

>> pinv(a)

ans =

-1.3333      -0.3333       0.6667
1.0833       0.3333      -0.4167

```

```
>> pinv(a)*a

ans =

1.0000      0.0000
-0.0000     1.0000

>> a = rand(5,5)

a =

0.9649     0.8003     0.9595     0.6787     0.1712
0.1576     0.1419     0.6557     0.7577     0.7060
0.9706     0.4218     0.0357     0.7431     0.0318
0.9572     0.9157     0.8491     0.3922     0.2769
0.4854     0.7922     0.9340     0.6555     0.0462

>> inv(a)

ans =

  2.5545    -0.3119    -0.0173    -0.4492    -1.9962
 -4.9167    -0.1095     0.8740     2.7919     2.5562
  3.3797    -0.1001    -1.3938    -1.5253    -0.8910
 -0.6203     0.4252     0.9340    -1.0120     1.2230
 -2.0554     1.1445     0.1203     2.0420    -0.5531

>> a_pinv(a)

```

```
ans =

 1.0000   -0.0000    0.0000   -0.0000   -0.0000
 0.0000    1.0000   -0.0000   -0.0000    0.0000
-0.0000    0.0000    1.0000    0.0000    0.0000
 0.0000   -0.0000   -0.0000    1.0000   -0.0000
-0.0000    0.0000    0.0000    0.0000    1.0000
```

I is called an identity matrix because all its diagonal elements are 1 and all its non-diagonal elements are zero, which makes its determinant 1. The determinant of a matrix a is calculated using the command det(a). Automatic generation of an identity matrix is done using the command eye(a,b), where a and b are values of the numbers of rows and columns.

```
>> eye(2,2)
ans =

1    0
0    1
>> det(eye(2,2))
ans =

1
>> eye(4,5)
ans =

1    0    0    0    0
0    1    0    0    0
0    0    1    0    0
0    0    0    1    0
```

rank()

The rank of a matrix, i.e., the number of linearly independent rows or columns, can be determined by the built-in `rank()` function.

```
a = ones(5,3)

a =

1    1    1
1    1    1
1    1    1
1    1    1
1    1    1

>> rank(a)

ans =

1

>> a = rand(3,2)

a =

0.4456    0.7547
0.6463    0.2760
0.7094    0.6797

>> rank(a)

ans =

2
```

2.6.3 trace()

The sum of the diagonal elements of a matrix is called the *trace* of the matrix. This is given by the built-in `trace()` function, as follows:

```
>> a = ones(4,4)

a =

1    1    1    1
1    1    1    1
1    1    1    1
1    1    1    1

>> trace(a)

ans =

4
```

norm()

The `norm()` function calculates the 2-norm of a matrix, which is equal to the Euclidean length of the vector.

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

A =

1     2
3     4
5     6

```

```
>> norm(A)

ans =

9.5255

>> A = [1,2,3]

A =

1     2     3

>> norm(A)

ans =

3.7417
```

Logical Operations

Two matrices can be compared to each other element-wise.

```
>> a = rand(2,3)

a =

0.6787    0.7431    0.6555
0.7577    0.3922    0.1712

>> b = rand(2,3)

```

```
b =

0.7060    0.2769    0.0971
0.0318    0.0462    0.8235

>> c = (a<b)

c =

2x3 logical array

1   0   0
0   0   1
>> whos
Name      Size            Bytes    Class     Attributes

a         2x3                48    double
b         2x3                48    double
c         2x3                 6    logical
>> a+c

ans =

1.6787   0.7431   0.6555
0.7577   0.3922   1.1712
```

The matrix c has elements, either 1 or 0, which are assigned by determining whether the corresponding elements of a are smaller than b. Note that using whos command, we can probe the variables a, b, and c. The matrix c contains logical data types, i.e., 1 and 0 represent the boolean quantities True and False. But performing a+c treats them as numerals.

This artifact leads to erroneous computations, hence some programming languages like Python explicitly use `True` and `False` representations for boolean values rather than 1 and 0.

2.6.4 Polynomials and Arrays

Every matrix has a characteristic polynomial associated with it. It can be found using the `poly()` function. Let's look at an example:

```
>> A1 = [-3 2 0 4]

A1 =

-3     2     0     4

>> B1 = poly(A1)

B1 =

1    -3    -10    24    0

>> A2 = [1,2;3,4]

A2 =

1    2
3    4

>> B2 = poly(A2)

B =

1.0000    -5.0000    -2.0000
```

In the first case, the resultant polynomial (given by `B1`) is $x^4 - 3x^3 - 10x^2 + 24x$, whereas in the second case (given by `B2`), it's $x^2 - 5x - 2$. The resultant matrix presents the coefficients of the characteristic polynomial.

find()

The built-in function `find()` returns the row and column indices of non-zero entries in a matrix. For example, in the 2×2 matrix defined by `A = [1,0;0,2]`, the non-zero elements exist at `A(1,1)` and `A(2,2)`. The information about rows and columns as a vector is demonstrated here:

```
>> A = [1,0;0,2]

A =

1    0
0    2

>> [row,col,v]=find(A)

row =

1
2


col =

1
2
```

```
v =

 1
 2
```

sort()

The built-in function `sort()` can be used to sort the elements of each column in a particular order. The order can be specified as a second argument to the function as a string (`ascend` or `descend`).

```
>> A = [1,-2,3;4,5,-2;0,-2,3]

A =

 1    -2     3
 4     5    -2
 0    -2     3

>> sort(A)

ans =

 0    -2    -2
 1    -2     3
 4     5     3

>> sort(A,'ascend')

ans =

```

```
21  0    -2    -2
22  1    -2     3
23  4     5     3
24
25  >> sort(A,'descend')
26
27  ans =
28
29  4    5    3
30  1   -2    3
31  0   -2   -2
```

2.7 Random Matrix

Using random number generators, a random matrix can be created. Use the `rand(a,b)` command:

```
 1  >> rand(4,5)
 2  ans =
 3
 4  Columns 1 through 4
 5
 6  0.8147    0.6324    0.9575    0.9572
 7  0.9058    0.0975    0.9649    0.4854
 8  0.1270    0.2785    0.1576    0.8003
 9  0.9134    0.5469    0.9706    0.1419
10
11  Column 5
12
```

```
13  0.4218
14  0.9157
15  0.7922
16  0.9595
```

Note that the numbers generated here will be different each time even on the same machine, since they are supposed to be random in nature. By default, they are uniformly distributed over the interval (0, 1). A vector is simply a row vector, so it can be generated randomly using the `rand(a)` command. `help rand` provides a detailed description of various other features and arguments of the random number generator.

To create random integers, you can use `randi()` function. You can also specify a range for these random integers. For example, `randi([1,10],1,5)` will create five random integers (an array of 1× 5) within 1 to 10. On the other hand, `randi([1,10],5)` will create an array of random integers (an array of 5×5) within 1 to 10.

```
 1  >> randi([1,10],5)
 2
 3  ans =
 4
 5  5    3    5    8    10
 6  5    7   10    3     6
 7  7    7    4    6     2
 8  8    2    6    7     2
 9  8    2    3    9     3
10
11  >> randi([1,10],1,5)
12
13  ans =
14
15  9    3    9    3    10
```

A random complex number can be generated using the rand command, as follows:

```
>> rand + i* rand

ans =

0.3500 + 0.1966i

>> rand + i* rand

ans =

0.2511 + 0.6160i
```

Sometimes, you might want to generate the same set of random numbers each time the program executes. This can be done by setting the state of the random number function using the rng command, as follows:

```
>> state 1 = rng;
>> r1 = rand(2,3)

r1 =

0.4733    0.8308    0.5497
0.3517    0.5853    0.9172

>> r12= rand(2,3)

r12=

0.2858    0.7537    0.5678
0.7572    0.3804    0.0759

```

```
>> rng(s);
Undefined function or variable 's'.

>> rng(state 1);
>> r3= rand(2,3)

r3=

0.4733    0.8308    0.5497
0.3517    0.5853    0.9172
```

The state is saved in the `state1` variable and then `r1` and `r2` creates two arrays of 2×3 size. They have different elements. But when the state is reset using `rng(state1)`, the new array of the same size stored in `r3` is exactly the same as `r1`, which was created when the state of the machine was saved in the `state1` variable.

A normally distributed random number generator is given by the function `randn()`. The random numbers, thus generated, are normally distributed around 0. Figure 3-7 in Chapter 3 confirms this fact.

A 3D array of random numbers can be generated by inputting an array for each dimension. For example, if an array `A = [3,2,4]` is fed into the `rand()` function, an 3D array of random numbers is created, as shown here:

```
>> A = [3,2,4];
>> B = rand(A)

B(:,:,1) =

0.7482    0.2290
0.4505    0.9133
0.0838    0.1524


```

```
B(:,:,2) =

0.8258    0.0782
0.5383    0.4427
0.9961    0.1067


B(:,:,3) =

0.9619    0.8173
0.0046    0.8687
0.7749    0.0844


B(:,:,4) =

0.3998    0.4314
0.2599    0.9106
0.8001    0.1818

>> size(B)

ans =

3    2    4
```

2.7.1 Matrix Manipulations

Some common matrix manipulations have been written in function form, which makes it easier for developers to use them right away, rather than invest time in writing optimum code.

2.7.2 Flipping a Matrix

`flipud(A)` returns a copy of matrix A with the order of the rows reversed along the horizontal axis. `flipud` stands for *flip-up-down*. `fliplr(A)` returns a copy of matrix A with the order of the rows reversed from left to right. `fliplr` stands for *flip left right*.

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

1    2
3    4
5    6

>> fliplr(a)
ans =

2    1
4    3
6    5

>> flipud(a)
ans =

5    6
3    4
1    2
```

2.7.3 Rotating a Matrix

Using the command `rot90(a,n)`, you can rotate a matrix `a n` times by 90 degrees.

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

1    2
3    4
5    6

>> rot90(a,1)
ans =

2    4    6
1    3    5

>> rot90(a,2)
ans =

6    5
4    3
2    1

>> rot90(a,4)
ans =

1    2
3    4
5    6
```

2.7.4 Reshaping a Matrix

The number of rows and columns in a matrix can be changed provided the total number of elements remains the same.

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

1    2
3    4
5    6

>> reshape(a,6,1)
ans =

1
3
5
2
4
6
>> reshape(a,4,1)
Error using reshape
To RESHAPE the number of elements must not change.
```

2.7.5 Sorting

Numbers can be sorted in increasing order using the `sort` function:

```
>> a = rand(1,5)
a =

0.7431    0.3922    0.6555    0.1712    0.7060

```

```
>> sort(a)
ans =

0.1712    0.3922    0.6555    0.7060    0.7431
```

2.7.6 Upper and Lower Triangular Matrix

The upper triangular matrix is such that only diagonal and elements above diagonal are non-zero. Similarly, the lower triangular matrix is such that diagonal and elements below diagonal are non-zero.

```
>> a = rand(3,3)
a =

0.0318    0.0971    0.3171
0.2769    0.8235    0.9502
0.0462    0.6948    0.0344

>> tril(a)
ans =

0.0318    0         0
0.2769    0.8235    0
0.0462    0.6948    0.0344

>> triu(a)
ans =

0.0318    0.0971    0.3171
0         0.8235    0.9502
0         0         0.0344
```

2.7.7 Ones and Zeros Matrix

A matrix having all its numbers as 1 or 0 make up a ones and zeros matrix, respectively:

```
>> ones(3,3)
ans =

1    1    1
1    1    1
1    1    1

>> zeros(3,3)
ans =

0    0    0
0    0    0
0    0    0
```

2.8 Indexing

Each element of the matrix is characterized by two numbers, the row number and the column number. This is used to pinpoint an element and operate on that.

```
>> a = rand(2,3)
a =

0.6557    0.8491    0.6787
0.0357    0.9340    0.7577

```

```
>> a(2,3)=1
a =

0.6557    0.8491    0.6787
0.0357    0.9340    1.0000

>> a(1,1)=0
a =

0         0.8491    0.6787
0.0357    0.9340    1.0000
```

Note that `a(2,3)=1` sets the element at the second row and third column, i.e., number 0.3041072 to 1, and `a(1,1)=0` sets the element at the first row and first column, i.e., number 0.5248873 to 0. To index numbers in a vector, you need a single number.

```
>> a = [1,2,3,4,5,6,7,8,9]
a =

Columns 1 through 7

1    2    3    4    5    6    7

Columns 8 through 9

8    9

>> a(1)
ans =

1
```

```
>> a(-1)
Subscript indices must either be real
positive integers or logicals.
>> a(5)
ans =

5
>> a(10)
Index exceeds matrix dimensions.
```

It is important to note that, unlike some programming languages where indices start at 0, MATLAB starts indices at 1 and does not take negative numbers as indices.

2.8.1 Using Indices to Create a New Vector

```
>> a = [10 20 30 40 50 60]
a =

10   20   30   40   50   60

>> b = a([1 3 6 1])
b =

10   30   60   10
```

In the previous example, b is a new vector formed from vector a, where successive elements are made up of elements taken from an index vector [1 3 6 1].

```
>> a = [11,12,13;40,50,60;17,18,19]
a =

```

```
 4  11    12    13
 5  40    50    60
 6  17    18    19
 7
 8  >> a([1,2], [2,3])
 9  ans =
10
11  12    13
12  50    60
```

Note that since the use of the comma operator is optional, we will define vectors and matrices by simply using whitespace.

2.9 Slicing

Matrices can be sliced to desired portions by using indices and the colon : operator.

```
 1  >> a = [1 2 3 4 1 3 2 4 6 4 5]
 2  a =
 3
 4  Columns 1 through 7
 5
 6  1    2    3    4    1    3    2
 7
 8  Columns 8 through 11
 9
10  4     6     4     5
11
```

```
>> b =a(1:5)
b =

1   2   3   4   1

>> c = a(5:7)
c =

1    3    2
```

This is an important feature, as most of experimental calculations would demand filtering the data. Here, a slice of data will be stored separately in a variable and then various mathematical operations can be performed on it.

Now let's try to access slices of a multidimensional array. A matrix a is defined to be a 5×5 matrix.

```
>> a = rand(5,5)

a =

0.6948    0.3816    0.4456    0.6797    0.9597
0.3171    0.7655    0.6463    0.6551    0.3404
0.9502    0.7952    0.7094    0.1626    0.5853
0.0344    0.1869    0.7547    0.1190    0.2238
0.4387    0.4898    0.2760    0.4984    0.7513

>> b = a(1,1)

b =

0.6948
```

```
>> c = a(1,:)

c =

0.6948    0.3816    0.4456    0.6797    0.9597

>> d = a(:,1)

d =

0.6948
0.3171
0.9502
0.0344
0.4387
>> e = a(:)

e =

0.6948
0.3171
0.9502
0.0344
0.4387
0.3816
0.7655
0.7952
0.1869
0.4898
0.4456
0.6463
```

```
47  0.7094
48  0.7547
49  0.2760
50  0.6797
51  0.6551
52  0.1626
53  0.1190
54  0.4984
55  0.9597
56  0.3404
57  0.5853
58  0.2238
59  0.7513
60  >> f = a(:,[1,3])
61
62  f =
63
64  0.6948    0.4456
65  0.3171    0.6463
66  0.9502    0.7094
67  0.0344    0.7547
68  0.4387    0.2760
69  >> g= a([1,3],:)
70
71  g =
72
73  0.6948    0.3816    0.4456    0.6797    0.9597
74  0.9502    0.7952    0.7094    0.1626    0.5853
```

- To access a single element, we use the index value of the row and column, For example, `b = a(1,1)` accesses the element within the first row and first column.

- To access all elements of a row or column, you can use the `:` operator. Hence, `c = a(1,:)` accesses all elements of the first row. Similarly, `>> d = a(:,1)` accesses all elements of the first column. A simple way to remember in words is to read the colon (`:`) as *all elements for* and then the words *n^{th} row/column*, where *n* is a given value.
- Using `a(:)`, you can create a new column matrix that has all the elements.
- A sub-matrix can be accessed by defining *all elements for* column/row and then defining indices in square brackets. For example, `f = a(:,[1,3])` defines a new matrix where elements are composed of *all elements of the first and third columns*. Similarly, `a([1,3],:)` uses all elements of first and third rows.

You can compose complex sub-matrices using this powerful way of defining your choice of elements.

```
>> a = rand(5,6)

a =

0.3510    0.1233    0.9027    0.9001    0.2417    0.9561
0.5132    0.1839    0.9448    0.3692    0.4039    0.5752
0.4018    0.2400    0.4909    0.1112    0.0965    0.0598
0.0760    0.4173    0.4893    0.7803    0.1320    0.2348
0.2399    0.0497    0.3377    0.3897    0.9421    0.3532

>> b = a([2,5],1:3)

```

```
b =

0.5132    0.1839    0.9448
0.2399    0.0497    0.3377
>> c = a(2:5,[1,3])

c =

0.5132    0.9448
0.4018    0.4909
0.0760    0.4893
0.2399    0.3377
>> d = a([2,5],[1,3])

d =

0.5132    0.9448
0.2399    0.3377
>>e = a(2:5,1:3)

e=

0.5132    0.1839    0.9448
0.4018    0.2400    0.4909
0.0760    0.4173    0.4893
0.2399    0.0497    0.3377
```

We define a new 5×5 matrix a and then define a subset of this matrix using `a([2,5],1:3)`, which says that *from the second and third row, take elements from the first column to the third column.* Similarly, `c = a(2:5,[1,3])` creates a matrix using this logic: *from* the first and fifth *column, take elements from the second row to the third row*. Now you can easily guess what `a([2,5],[1,3])`

and `a(2:5,1:3)` should do. It's a good idea to practice slicing of arrays rigorously, as this is one of the most sought-after skills in data cleaning and data analysis in general.

2.10 Automatic Generation of Arrays

MATLAB presents a variety of ways to generate arrays of numbers automatically according to a specified rule. Three methods are discussed in the following sections.

2.10.1 The : Operator

One of the most useful operators in MATLAB, the : operator can be mastered easily. You have already seen its usage in selecting a sub-matrix in Chapter 2.

```
>> help:
:  Colon.
J:K is the same as [J,J+1,...,J+m], where m = fix(K-J). In the
case where both J and K are integers, this is simply
[J,J+1,...,K].
This syntax returns an empty matrix if J>K.

J:I:K is the same as [J,J+I,...,J+m_I], where
m = fix((K-J)/I).
This syntax returns an empty matrix when I == 0,
I>0 and J>K, or
I<0 and J<K.

colon (J,K) is the same as J:K and colon (J,I,K) is the
same as J:I:K.

```

```
13 The colon notation can be used to pick out selected rows,
   columns
14 and elements of vectors, matrices, and arrays. A(:) is all the
15 elements of A, regarded as a single column. On the left
   side of an
16 assignment statement, A(:) fills A, preserving its shape
   from before.
17 A(:,J) is the J-th column of A. A(J:K) is
   [A(J),A(J+1),...,A(K)].
18 A(:,J:K) is [A(:,J),A(:,J+1),...,A(:,K)] and so on.
19
20 The colon notation can be used with acellar ray to produce
   a comma-
21 separated list. C{:} is the same as Cf1g,Cf2g,...,Cfendg.
22 The comma separated list syntax is valid inside () for
   function calls, [] for
23 concatenation and function return arguments, and inside fg
   to produce
24 a cell array. Expressions such as S(:). name produce the
   comma
25 separated list S(1).name,S(2).name,...,S(end). name for the
   structure S.
26
27 For the use of the colon in the FOR statement, See FOR.
28 For the use of the colon in a comma separated list, See
   VARARGIN.
29
30 Reference page for colon
31 Other functions named colon
```

You can generate a series of numbers and store them as arrays by using the `start:step:stop` command.

```
>> a=1:1:10
a =

Columns 1 through 7

1    2    3    4    5    6    7

Columns 8 through 10

8    9    10

>> a =[1:1:10]
a =

Columns 1 through 7

1    2    3    4    5    6    7

Columns 8 through 10

8    9    10
```

Note that brackets ([ ]) are optional here. If a step is not defined, then it is taken as 1.

```
>> a=1:10
a =

Columns 1 through 7

```

```
 6  1    2    3    4    5    6    7
 7
 8  Columns 8 through 10
 9
10  8    9    10
11
12  >> a=1:2:10
13
14  a =
15
16  1    3    5    7    9
```

2.10.2 Linearly Spaced Vectors

The `linspace(start, stop, n)` command produces an array starting at the first number and stopping at the second one with a total of *n* numbers. Hence, they are linearly spaced.

```
 1  >> a = linspace(1,2,5)
 2  a =
 3
 4  Columns 1 through 4
 5
 6  1.0000    1.2500    1.5000    1.7500
 7
 8  Column 5
 9
10  2.0000
11
12  >> a = linspace(1,2,10)
13  a =
14
```

```
15  Columns 1 through 4
16
17  1.0000    1.1111    1.2222    1.3333
18
19  Columns 5 through 8
20
21  1.4444    1.5556    1.6667    1.7778
22
23  Columns 9 through 10
24
25  1.8889    2.0000
```

2.10.3 logspace

Similar to the `linspace` command, `logspace(start, stop, n)` produces *n* numbers from start to stop, which are linearly spaced in logarithmic nature.

```
1  >>> help logspace
2  logspace Logarithmically spaced vector.
3  logspace(X1,X2) generates a row vector of 50
   logarithmically
4  equally spaced points between decades 10^X1 and 10^X2. If X2
5  is pi, then the points are between 10^X1 and pi.
6
7  logspace(X1,X2,N) generates N points.
8  For N = 1, logspace returns 10^X2.
9
10 Class support for inputs X1,X2:
11 float:double, single
12
```

```
13  See also linspace, colon.
14
15  Reference page for logspace
16  >>>logspace (1,5,10)
17
18  ans =
19
20  1.0e+05 *
21
22  Columns 1 through 4
23
24  0.0001    0.0003    0.0008    0.0022
25
26  Columns 5 through 8
27
28  0.0060    0.0167    0.0464    0.1292
29
30  Columns 9 through 10
31
32  0.3594    1.0000
```

2.11 Solving a System of Equations

Solving a system of equations in one line simply involves the \ operator. Suppose the following system of equations needs to be solved:

$$2x - 2y = 4 \qquad \text{(Equation 2-1)}$$

$$-3x + 4y = 9 \qquad \text{(Equation 2-2)}$$

You can define this problem in a matrix, as follows:

$$\begin{bmatrix} 2 & -2 \\ -3 & 4 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix} \quad \text{(Equation 2-3)}$$

Suppose:

$$A = \begin{bmatrix} 2 & -2 \\ -3 & 4 \end{bmatrix} \quad \text{(Equation 2-4)}$$

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{(Equation 2-5)}$$

$$B = \begin{bmatrix} 4 \\ 9 \end{bmatrix} \quad \text{(Equation 2-6)}$$

In this way, you can write the following:

$$A \times X = B \quad \text{(Equation 2-7)}$$

The solution is given by $X = A^{-1}B$. You can find the inverse of `A` (using the `inv()` or `pinv()`) function) and then multiply the resultant matrix with the matrix given by `B` to find a solution. Alternatively, you can accomplish this task in just one command, as `A\B`:

```
>> A = [2,-2;-3,4]

A =

2   -2
-3    4

>> B = [4;9]

```

```
B =

4   9

>> C = A/B

ans =

17.0000
15.0000
```

Hence, the solution is $x = 17$ and $y = 15$. Since the elements of the C matrix are solutions, this is often called a solution matrix.

2.12 Eigen Values and Eigen Vectors

The eigenvalue problem is to determine the solution to the equation $Av = \lambda v$, where A is an $n \times n$ matrix, v is a column vector of length n, and λ is a scalar. The values of λ that satisfy the equation are the eigenvalues. The corresponding values of v that satisfy the equation are the right eigenvectors. The left eigenvectors, w, satisfy the equation $w'A = \lambda w'$. The MATLAB function `eig()` returns the eigenvalues and eigenvectors. It also gives the matrix D (diagonal matrix D of eigenvalues), which is related to W and A as $W'A = DW'$:

```
>> A = rand(3,3)

A =

0.6551    0.4984    0.5853
0.1626    0.9597    0.2238
0.1190    0.3404    0.7513
```

```
>> [V,D,W] = eig(A)

V =

-0.7284   -0.9532   0.8945
-0.5300   0.2997    -0.4178
-0.4341   0.0411    0.1590


D =

1.3665        0             0
0             0.4732        0
0             0             0.5264


W =

-0.2724    -0.3066    -0.1266
-0.7915    -0.3145    -0.5186
-0.5471    0.8984     0.8456
```

2.13 Structure Arrays

Arrays stores elements of the same data types, whereas *structure arrays* can store data of different data types. Structures are collections of data organized by named fields. For example, one field may contain textual data, another a number, and a third may be an array, etc. A single structure is a `1-by-1` structure array. Let's understand how to create them by using an example. Let's create a structure array for this book and name this array `book`. Now, various fields can be added using the dot operator, such

as name, author, pages, and chapter. The book array is a 1-by-1 structure with four fields. This is demonstrated here.

```
>> book.name = 'Introducing MATLAB'

book =

struct with fields:

name:'Introducing MATLAB'

>> book.author = 'Sandeep Nagar'

book =

struct with fields:

name:'Introducing MATLAB'
author:'Sandeep Nagar'

>> book.pages = '175'

book =

struct with fields:

name:'Introducing MATLAB'
author:'Sandeep Nagar'
pages:'175'

>> book.chapters = [1 2 3 4 5 6 7]

```

```
30 book =
31
32 struct with fields:
33
34 name:'Introducing MATLAB'
35 author:'Sandeep Nagar'
36 pages:'175'
37 chapters:[1 2 3 4 5 6 7]
```

2.13.1 Defining a New Structure Element Within a Structure Array

A new structure element can be defined within an existing structure array (book, in this example) using index values in the following manner.

```
 1 >> book(2).name = 'Introducing SCILAB'
 2
 3 book =
 4
 5 1x2 struct array with fields:
 6
 7 name
 8 author
 9 pages
10 chapters
11
12 >> book(2).author = 'Sandeep Nagar'
13
14 book =
15
16 1x2 struct array with fields:
17
```

```
18  name
19  author
20  pages
21  chapters
22
23  >> book(2).pages = 175
24
25  book =
26
27  1x2 struct array with fields:
28
29  name
30  author
31  pages
32  chapters
33
34  >> book(2).chapters = [1 2 3 4 5 6 7 8 9]
35
36  book =
37
38  1x2 struct array with fields:
39
40  name
41  author
42  pages
43  chapters
```

In this way, the book is now a 1×2 structure array. All structures in a structure array have the same number of fields and all fields have the same number of field names. When the name of the structure array is entered at the command prompt, the summary of information and fields is displayed.

The `fieldnames()` function can be used to get a *cell array* having information about the fields. This is demonstrated in the following code.

```
>> book

book =

1x2 struct array with fields:

name
author
pages
chapters

>> fieldnames(book)

ans =

4x1 cell array

'name'
'author'
'pages'
'chapters'
```

While expanding a structure array, it is not mandatory to fill in all the fields. Fields that are not associated with values are left empty.

2.13.2 Adding and Removing Fields

A new field can be added at any point to a single structure. For example, let's add the field `publisher` to the structure `book`, as demonstrated here.

```
>> book(2).publisher = 'Apress'

book =

1x2 struct array with fields:

name
author
pages
chapters
publisher

>> book

book =

1x2 struct array with fields:

name
author
pages
chapters
publisher

>> book = rmfield(book,'publisher')

book =

1x2 struct array with fields:

```

```
name
author
pages
chapters
```

To remove a field, say `publisher`, from the structure `book`, you can use the `rmfield()` function, as demonstrated.

2.13.3 struct()

The function `struct()` can also be used to define a structured array with the syntax shown in the following code:

```
>> book1 = struct('name','Introducing MATLAB','author',
'Sandeep Nagar','pages',175,'chapters',[1, 2, 3, 4, 5, 6, 7])

book1 =

struct with fields:

name:'Introducing MATLAB'
author:'Sandeep Nagar'
pages:175
chapters:[1 2 3 4 5 6 7]

>> book1(2) = struct('name','Introducing python','author',
'Sandeep Nagar','pages',175,'chapters',[1, 2, 3, 4, 5, 6,
7, 8, 9])

book1 =

```

```
16  1x2 struct array with fields:
17
18  name
19  author
20  pages
21  chapters
```

A new structure named `book1` is created where *field names* and *values* are filled in successively. It can be expanded using the index number in a similar fashion, making it a `1-by-2` structure array.

A structure array may contain another structure or even a structure array as its fields. These are called nested array. This is demonstrated here, where `book1` (a structure array defined previously) is added as a new field to the structure array `book`.

```
 1  >> book(3).linked book = book1
 2
 3  book =
 4
 5  1x3 struct array with fields:
 6
 7  name
 8  author
 9  pages
10  chapters
11  linked_book
```

2.14 Getting Data from a Structure Array

Data *values* can be assigned from a structure array using index numbers, as demonstrated next. Here, `info1` stores the value of the field `name` for the second structure (signified by the syntax `book(2)`). In a similar fashion,

info2 stores the value of the field name for the first structure (signified by the syntax book(1)). The variable info3 extracts the third element of the field chapter from the second structure of the structure array book.

```
>> info1 = book(2).name()

info1 =

'Introducing SCILAB'

>> info2 = book(1).name()

info1 =

'Introducing MATLAB'

info3 = book(2).chapters(3)

info3 =

3
```

2.15 Cell Arrays

Cell arrays are arrays of cells where each cell stores an array. Within a cell, elements must be the same type (because cells store arrays), but two cells may have different types. For example, suppose you have three arrays—array1 (stores numerical values), array2 (stores textual values), and array2 (stores numerical values). You can then construct a cell array using these three arrays. The elements of this cell array store different types of arrays, but each element stores just one type of data.

2.15.1 Creating Cell Arrays

The `cell(m,n)` function makes an *empty* cell array of the size $m - by - n$. By assigning data values to this empty cell array, it can then be constructed as desired, *one cell at a time*. Let's first create an empty cell array, referenced by a variable, say `a`. There are two ways to assign the data:

- *Cell indexing*: Cell indices are mentioned within parentheses `()` and cell contents are mentioned within brackets `{}` on either side of assignment operator, like so:

```
>> a = cell(3,3)

a =

3x3 cell array

[]    []    []
[]    []    []
[]    []    []

>> a(1,1) = ([1,2,3]);
>> a(1,2) = (['a','b']);
>> a(1,3) = ("Sandeep");
>> a(2,3) = ([1.5,-2]);
>> a(2,2) = ([-200]);
>> a(2,1) = (["Nagar"]);
>> a(3,1) = ([-10,-15.5,5.3]);
>> a(3,2) = (["Hello"]);
>> a(3,3) = (["World"])

a =

```

```
23  3x3 cell array
24
25  [1x3 double]    'ab'           ["Sandeep"]
26  ["Nagar"]       [-200]         [1x2 double]
27  [1x3 double]    ["Hello"]      ["World"]
28
```

- *Content indexing*: Here, brackets/parentheses are used in reverse fashion, i.e., () for content and [ ] for indices.

```
 1  >> a = cell(3,3)
 2
 3  a =
 4
 5  3x3 cell array
 6
 7  []    []    []
 8  []    []    []
 9  []    []    []
10
11  >> a {1,1} = ([1,2,3]);
12  >> a {1,2} = (['a','b']);
13  >> a {1,3} = ("Sandeep");
14  >> a {2,3} = ([1.5,-2]);
15  >> a {2,2} = ([-200]);
16  >> a {2,1} = (["Nagar"]);
17  >> a {3,1} = ([-10,-15.5,5.3]);
18  >> a {3,2} = (["Hello"]);
19  >> a {3,3} = (["World"])
20
```

```
21  a =
22
23  3x3 cell array
24
25  [1x3 double]    'ab'          ["Sandeep"]
26  ["Nagar"]       [-200]        [1x2 double]
27  [1x3 double]    ["Hello"]     ["World"]
28
```

2.15.2 The celldisp() and cellplot() Functions

The constructed cell arrays can be displayed by using two functions called `celldisp()` and `cellplot()`. The `celldisp()` command displays the full cell contents, whereas `cellplot()` displays a graphical display of the cell architecture. See Figure 2-1.

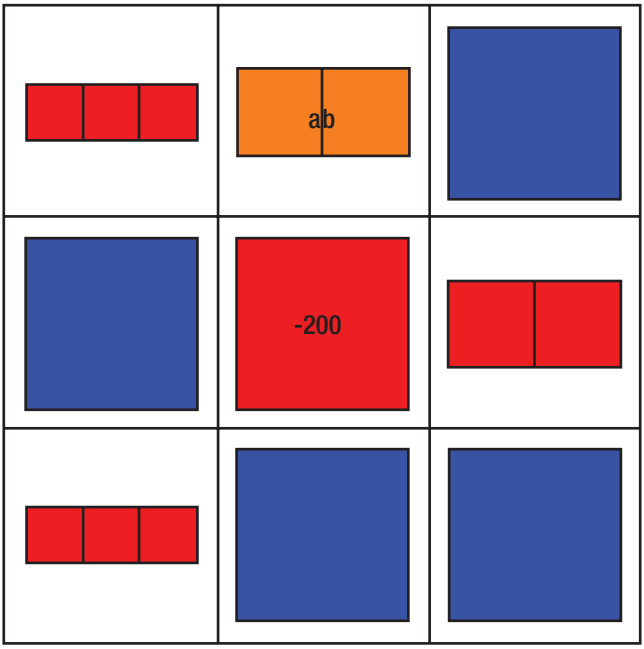

Figure 2-1. *Output of cellplot (a)*

2.15.3 The cell2struct(), num2cell(), and struct2cell() Functions

The `cell2struct()` command can be used to convert a cell array to a structure. Similarly, `num2cell()` can be used to convert a numeric array into a cell array and `struct2cell()` can be used to convert a structure into a cell array.

2.16 Summary

Array based computing lies at the very heart of modern computational techniques. MATLAB presents a very suitable platform to perform this technique with ease. A variety of predefined functions enable users to save time while prototyping a problem. Having flexible methods to define multidimensional arrays and perform fast computation is the necessity of our times. Most of the time spent on a simulation is either in loops or in array operations. Predefined array operations have been optimized with algorithms for reliability, time savings, and efficient memory management.