1 Introduction to MATLAB

1.1 Quick Overview

This chapter is not intended to be a comprehensive manual of MATLAB®. Our sole aim is to provide sufficient information to give you a good start. If you are familiar with another computer language, and we assume that you are, it is not difficult to pick up the rest as you go.

MATLAB is a high-level computer language for scientific computing and data visualization built around an interactive programming environment. It is becoming the premiere platform for scientific computing at educational institutions and research establishments. The great advantage of an interactive system is that programs can be tested and debugged quickly, allowing the user to concentrate more on the principles behind the program and less on programming itself. Because there is no need to compile, link, and execute after each correction, MATLAB programs can be developed in a much shorter time than equivalent FORTRAN or C programs. On the negative side, MATLAB does not produce stand-alone applications – the programs can be run only on computers that have MATLAB installed.

MATLAB has other advantages over mainstream languages that contribute to rapid program development:

- MATLAB contains a large number of functions that access proven numerical libraries, such as LINPACK and EISPACK. This means that many common tasks (e.g., solution of simultaneous equations) can be accomplished with a single function call.
- Extensive graphics support allows the results of computations to be plotted with a few statements.
- All numerical objects are treated as double-precision arrays. Thus there is no need to declare data types and carry out type conversions.
- MATLAB programs are clean and easy to read; they lack the syntactic clutter of some mainstream languages (e.g., C).

The syntax of MATLAB resembles that of FORTRAN. To get an idea of the similarities between these programming languages, let us compare the codes written in the two languages for solution of simultaneous equations $\mathbf{Ax} = \mathbf{b}$ by Gauss elimination (do not

worry about understanding the inner workings of the programs). Here is the subroutine in FORTRAN 90:

```
subroutine gauss(A,b,n)
   use prec_mod
   implicit none
   real(DP), dimension(:,:), intent(in out) :: A
   real(DP), dimension(:),   intent(in out) :: b
   integer, intent(in)                      :: n
   real(DP) :: lambda
   integer  :: i,k
 ! --------------Elimination phase--------------
   do k = 1,n-1
      do i = k+1,n
         if(A(i,k) /= 0) then
            lambda = A(i,k)/A(k,k)
            A(i,k+1:n) = A(i,k+1:n) - lambda*A(k,k+1:n)
            b(i) = b(i) - lambda*b(k)
         end if
      end do
   end do
 ! ------------Back substitution phase----------
   do k = n,1,-1
      b(k) = (b(k) - sum(A(k,k+1:n)*b(k+1:n)))/A(k,k)
   end do
   return
end subroutine gauss
```

The statement `use prec_mod` tells the compiler to load the module `prec_mod` (not shown here), which defines the word length `DP` for floating-point numbers. Also note the use of array sections, such as `a(k,k+1:n)`, a very useful feature that was not available in previous versions of FORTRAN.

The equivalent MATLAB function is (MATLAB does not have subroutines):

```
function b = gauss(A,b)
n = length(b);
%-----------------Elimination phase-------------
for k = 1:n-1
    for i = k+1:n
        if A(i,k) ~= 0
            lambda = A(i,k)/A(k,k);
            A(i,k+1:n) = A(i,k+1:n) - lambda*A(k,k+1:n);
            b(i)= b(i) - lambda*b(k);
         end
     end
```

```
end
%--------------Back substitution phase-----------
for k = n:-1:1
    b(k) = (b(k) - A(k,k+1:n)*b(k+1:n))/A(k,k);
end
```

Simultaneous equations can also be solved in MATLAB with the simple command `A\b` (see later).

MATLAB can be operated in the interactive mode through its command window, where each command is executed immediately upon its entry. In this mode, MATLAB acts like an electronic calculator. Following is an example of an interactive session for the solution of simultaneous equations:

```
>> A = [2 1 0; -1 2 2; 0 1 4]; % Input 3 x 3 matrix.
>> b = [1; 2; 3];              % Input column vector
>> soln = A\b                  % Solve A*x = b by 'left division'
soln =
    0.2500
    0.5000
    0.6250
```

The symbol ≫ is MATLAB's prompt for input. The percent sign (%) marks the beginning of a comment. A semicolon (;) has two functions: it suppresses printout of intermediate results and separates the rows of a matrix. Without a terminating semicolon, the result of a command would be displayed. For example, omission of the last semicolon in the line defining the matrix `A` would result in the following:

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

Functions and programs can be created with the MATLAB editor/debugger and saved with the `.m` extension (MATLAB calls them M-files). The file name of a saved function should be identical to the name of the function. For example, if the function for Gauss elimination is saved as `gauss.m`, it can be called just like any MATLAB function:

```
>> A = [2 1 0; -1 2 2; 0 1 4];
>> b = [1; 2; 3];
>> soln = gauss(A,b)
soln =
    0.2500
    0.5000
    0.6250
```

1.2 Data Types and Variables

Data Types

The most commonly used MATLAB data types, or *classes*, are `double`, `char`, and `logical`, all of which are considered by MATLAB as arrays. Numerical objects belong to the class `double`, which represents double-precision arrays; a scalar is treated as a 1×1 array. The elements of a `char` type array are strings (sequences of characters), whereas a `logical` type array element may contain only 1 (true) or 0 (false).

Another important class is `function_handle`, which is unique to MATLAB. It contains information required to find and execute a function. The name of a function handle consists of the character `@` followed by the name of the function, for example, `@sin`. Function handles are used as input arguments in function calls. For example, suppose that we have a MATLAB function `plot(func,x1,x2)` that plots any user-specified function `func` from `x1` to `x2`. The function call to plot $\sin x$ from 0 to π would be `plot(@sin,0,pi)`.

There are other data types, such as `sparse` (sparse matrices), `inline` (inline objects), and `struct` (structured arrays), but we seldom come across them in this text. Additional classes can be defined by the user. The class of an object can be displayed with the `class` command, for example,

```
>> x = 1 + 3i  % Complex number
>> class(x)
ans =
double
```

Variables

Variable names, which must start with a letter, are *case sensitive*. Hence `xstart` and `XStart` represent two different variables. The length of the name is unlimited, but only the first N characters are significant. To find N for your installation of MATLAB, use the command `namelengthmax`:

```
>> namelengthmax
ans =
    63
```

Variables that are defined within a MATLAB function are local in their scope. They are not available to other parts of the program and do not remain in memory after exiting the function (this applies to most programming languages). However, variables can be shared between a function and the calling program if they are declared `global`. For example, by placing the statement `global X Y` in a function as well as the calling program, the variables `X` and `Y` are shared between the two program units. The recommended practice is to use capital letters for global variables.

MATLAB contains several built-in constants and special variables, most important of which are

ans	Default name for results
eps	Smallest number for which `1 + eps > 1`
inf	Infinity
NaN	Not a number
i or j	$\sqrt{-1}$
pi	π
realmin	Smallest usable positive number
realmax	Largest usable positive number

Following are a few examples:

```
>> warning off  % Suppresses print of warning messages
>> 5/0
ans =
   Inf

>> 0/0
ans =
   NaN

>> 5*NaN        % Most operations with NaN result in NaN
ans =
   NaN

>> NaN == NaN   % Different NaN's are not equal!
ans =
     0

>> eps
ans =
  2.2204e-016
```

Arrays

Arrays can be created in several ways. One way is to type the elements of the array between brackets. The elements in each row must be separated by blanks or commas. Following is an example of generating a 3×3 matrix:

```
>> A = [ 2 -1  0
        -1  2 -1
         0 -1  1]
```

```
A =
     2    -1     0
    -1     2    -1
     0    -1     1
```

The elements can also be typed on a single line, separating the rows with semicolons:

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

Unlike most computer languages, MATLAB differentiates between row and column vectors (this peculiarity is a frequent source of programming and input errors). For example,

```
>> b = [1 2 3]          % Row vector
b =
     1     2     3

>> b = [1; 2; 3]        % Column vector
b =
     1
     2
     3

>> b = [1 2 3]'         % Transpose of row vector
b =
     1
     2
     3
```

The single quote (') is the *transpose operator* in MATLAB; thus `b'` is the transpose of `b`.

The elements of a matrix, such as

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

can be accessed with the statement `A(i,j)`, where `i` and `j` are the row and column numbers, respectively. A section of an array can be extracted by the use of colon notation. Following is an illustration:

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

>> A(2,3)     % Element in row 2, column 3
ans =
     7

>> A(:,2)     % Second column
ans =
     1
     5
     9

>> A(2:3,2:3)  % The 2 x 2 submatrix in lower right corner
ans =
     5     7
     9     2
```

Array elements can also be accessed with a single index. Thus `A(i)` extracts the *i*th element of `A`, counting the elements *down the columns*. For example, `A(7)` and `A(1,3)` would extract the same element from a 3×3 matrix.

Cells

A cell array is a sequence of arbitrary objects. Cell arrays can be created by enclosing their contents between braces `{ }`. For example, a cell array `c` consisting of three cells can be created by

```
>> c = {[1 2 3], 'one two three', 6 + 7i}
c =
    [1x3 double]    'one two three'    [6.0000+ 7.0000i]
```

As seen, the contents of some cells are not printed to save space. If all contents are to be displayed, use the `celldisp` command:

```
>> celldisp(c)
c{1} =
     1     2     3
c{2} =
one two three
c{3} =
   6.0000 + 7.0000i
```

Braces are also used to extract the contents of the cells:

```
>> c{1}              % First cell
ans =
     1     2     3
>> c{1}(2)           % Second element of first cell
ans =
     2
>> c{2}              % Second cell
ans =
one two three
```

Strings

A string is a sequence of characters; it is treated by MATLAB as a character array. Strings are created by enclosing the characters between single quotes. They are concatenated with the function `strcat`, whereas the colon operator (`:`) is used to extract a portion of the string, for example,

```
>> s1 = 'Press return to exit';   % Create a string
>> s2 = ' the program';           % Create another string
>> s3 = strcat(s1,s2)             % Concatenate s1 and s2
s3 =
Press return to exit the program
>> s4 = s1(1:12)                  % Extract chars. 1-12 of s1
s4 =
Press return
```

1.3 Operators

Arithmetic Operators

MATLAB supports the usual arithmetic operators:

+	Addition
−	Subtraction
*	Multiplication
^	Exponentiation

When applied to matrices, they perform the familiar matrix operations:

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

>> A + B               % Matrix addition
ans =
     8    10    12
     4     6     8
```

```
>> A*B'                % Matrix multiplication
ans =
    50     8
   122    17

>> A*B                 % Matrix multiplication fails
??? Error using ==> *  % due to incompatible dimensions
Inner matrix dimensions must agree.
```

There are two division operators in MATLAB:

/	Right division
\	Left division

If `a` and `b` are scalars, the right division `a/b` results in `a` divided by `b`, whereas the left division is equivalent to `b/a`. In the case where `A` and `B` are matrices, `A/B` returns the solution of `X*A = B` and `A\B` yields the solution of `A*X = B`.

Often we need to apply the `*`, `/`, and `^` operations to matrices in an element-by-element fashion. This can be done by preceding the operator with a period (`.`), as follows:

.*	Element-wise multiplication
./	Element-wise division
.^	Element-wise exponentiation

For example, the computation $C_{ij} = A_{ij} B_{ij}$ can be accomplished with

```
>> A = [1 2 3; 4 5 6]; B = [7 8 9; 0 1 2];
>> C = A.*B
C =
     7    16    27
     0     5    12
```

Comparison Operators

The comparison (relational) operators return 1 for true and 0 for false. These operators are as follows:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
~=	Not equal to

The comparison operators always act element-wise on matrices; hence they result in a matrix of `logical` type, for example,

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

Logical Operators

The logical operators in MATLAB are as follows:

&	AND
|	OR
~	NOT

They are used to build compound relational expressions, an example of which follows:

```
>> A = [1 2 3; 4 5 6]; B = [7 8 9; 0 1 2];
>> (A > B) | (B > 5)
ans =
     1     1     1
     1     1     1
```

1.4 Flow Control

Conditionals

if, else, elseif

The `if` construct

```
if condition
    block
end
```

executes the *block* of statements if the *condition* is true. If the condition is false, the block skipped. The `if` conditional can be followed by any number of `elseif` constructs:

```
if condition
    block
elseif condition
    block
⋮
end
```