2 Matlab Programming, IO, and strings

Programming is the basic skill for implementing numerical methods. In this chapter we describe the fundamental programming constructs used in MATLAB and present examples of their applications to some elementary numerical methods. The second part of this chapter is dedicated at exploring input/output functions provided by MATLAB including operations with files. Finally, manipulation of strings in MATLAB is presented.

MATLAB programming constructs

MATLAB provides the user with a number of programming constructs very similar to those available in FORTRAN, Visual Basic, Java, and other high-level languages. We present some of the constructs below:

Comparison and Logical Operators

MATLAB comparison operators are

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

MATLAB logical operators are

&	and
|	or
~	not

As an example, try the following commands in MATLAB:

3 ~= 2 <enter>
3 == 3 <enter>
(2>1)&(3>1) <enter>
(2>1)&(3>5) <enter>
(2<1)&(3>1) <enter>
(2<1)&(3>5) <enter>
(2>1) | (3>1) <enter>
(2>1) | (3>5) <enter>
(2<1) | (3>1) <enter>
(2<1) | (3>5) <enter>
~(2<1) <enter>
~(2>1) <enter>
~(2>1) | (3>5) <enter>

Loops in MATLAB

MATLAB includes *For* and *While* loops. The *For* loop is similar to the DO loop in FORTRAN or the FOR..NEXT loop in Visual Basic. The basic construct for the *For* loop is:

```
for index = starting_value : increment : end_value, …statements…, end

for index = starting_value : end_value, …statements…, end
```

If no increment is included it is supposed to be equal to 1. For example, enter the following For loops in MATLAB:

```
r = 1; for k = 1:0.5:4, r = r+k, end <enter>
xs = -1.0; dx = 0.25; n = 10; for j = 1:n, x = xs + (j-1)*dx; end; x <enter>
for m = 1:10, a(m) = m^2; end; a <enter>
```

The basic construct for the While loop is:

```
while condition, …statements…, end
```

For example, try the following *while* loop:

```
s = 100; while s>50, disp(s^2), s = s - 5, end <enter>
```

For and *while* loops can be terminated with the command *break*, for example, try the following:

```
for j = 1:10, disp(j), if j>5 break, end, end <enter>
```

Conditional constructs in MATLAB

In the example above we used an *if…end* construct. There are two type of conditional constructs in MATLAB, one is the *if-(then)-else-end* construct (as in the example above) and the second one is *the select-case* conditional construct. Different forms of the *if-then-else* construct are:

if *condition statement*, end
if *condition statement1*, else *statement2*, end
if *condition1 statement1*, elseif *condition2 statement2*, else *statement3*, end

Try the following examples:

x = 10; y = 5; if x> 5 disp(y), end <enter>
x = 3 ; y = 5; if x>5 disp(y), else disp(x), end <enter>
x = 3; y = 5; z = 4; if x>5 disp(x), elseif x>6 disp(y), else disp(z), end <enter>

The general form of the *switch-case* construct is:

switch *variable*, case *n1*, *statement*, case *n2*, *statement*, …, end

Try the following examples:

x = -1; switch x, case 1, y = x+5, case -1, y = sqrt(x), end <enter>
r = 7; switch r, case 1, disp(r), case 2, disp(r^2), case 7, disp(r^3), end <enter>

M-files in Matlab

All the constructs shown above can be programmed in files following a structure similar to FORTRAN or Visual Basic programs, and then executed from within MATLAB. Such files are referred, in general, as *m* files, because they all use the *.m* subscript. *M*-files can be further

classified as *scripts* or *functions*. For example, type the following MATLAB script into a file called *program1.m*:

```
clear %erase all variables
x = [10. -1. 3. 5. -7. 4. 2.]
suma = 0;
[n,m] = size(x);
for j = 1:m
    suma = suma + x(j);
end
xbar = suma/m
```

Save it into the *work* sub-directory. Within MATLAB type:

```
program1.txt <enter>
```

Note that since x is a row vector (actually a matrix with *n = 1* row and *m = 7* columns), the *size* function provides you with an array of two values in the statement `[n,m] = size(x)`. Then, m is used in the *for* loop and in the calculation of *xbar*.

As illustrated in the example above, a Matlab script is an *m*-file containing a series of Matlab commands that will be executed from the file as they would have been typed in the Matlab interface. Every command ending in a semi-colon will be executed but no output from that command will show up in the Matlab interface. Only commands without semi-colons will show a result in the interface. Thus, in the example above, the only commands that produce outputs are `x = [10. -1. 3. 5. -7. 4. 2.]` and `xbar = suma/m`.

A script can be produced out of a *diary* file (see Chapter 1) after cleaning out prompts and results.

Function files in MATLAB

Functions are procedures that may take input arguments and return zero, one or more values. Functions are defined either *as inline* functions, using the *inline* command, or as a *separate m-file* that is saved under the same name as the function. Functions are invoked by typing the name of the functions with the appropriate arguments. Next, some examples of *inline* functions are presented:

Euler = inline('r*exp(i*theta)','r','theta') <enter>

Matlab replies with this result:

```
Euler =

     Inline function:
     Euler(r,theta) = r*exp(i*theta)
```

Evaluate this function by using:

Euler(1.0,-pi/2) <enter>

A couple of functions of two variables are given next. These functions are used to calculate the magnitude and angle of the vector connecting the origin (0,0) with point (x,y):

r = inline('sqrt(x^2+y^2)','x','y')<enter>

theta = inline('atan(y/x)','x','y')<enter>

The following is an evaluation of these functions:

r(3.,4.), theta(3.,4.) <enter>

NOTE: To learn more about the *inline* command, type: `help inline` <return>

Functions defined in files must start with the command

`Function [y1,…,yn] = fname(x1,…,xm)`

Where *fname* is the function name, [*y1*,…,*yn*] is an array of output values, and *x1*,…,*xm* are the input values. Type in the following function into a file called *sphecart.m* using the text editor (*File>New>M file*):

```
function [x,y,z] = sphecart(r,theta,rho)
%conversion from spherical to Cartesian coordinates
x = r*cos(rho)*cos(theta);
y = r*cos(rho)*sin(theta);
z = r*sin(rho);
```

In MATLAB evaluate the function using:

[x1,y1,z1]=sphecart(10.0, pi/3, pi/6) <enter>

Notice that MATLAB on-line functions are similar to FORTRAN function declarations, while MATLAB functions defined in files are similar to FORTRAN or Visual Basic function sub-programs or subroutines. The main difference is that FORTRAN and Visual Basic functions can only return one value, while MATLAB functions can return zero, one or more values.

An example of a function - Frobenius norm of a matrix.

This function is to be stored in file *AbsM.m* within subdirectory *work* in the MATLAB directory. The Frobenius norm of a matrix **A** = [a_{ij}] with n rows and m columns is defined as the square root of the sum of the squares of each of the elements of the matrix, i.e.,

$$\| A \|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}^2}$$

The function AbsM(A), listed below, calculates the Frobenius norm of a matrix:

```
function [v]=AbsM(A)
% This function calculates the Frobenius norm of a matrix
% First obtain the matrix size
[n m] = size(A);
% Then initialize suma and add terms a(i,j)^2
suma = 0;
for i = 1:n
  for j = 1:m
    suma = suma + A(i,j)^2;
  end
end;
% take square root and show result
v = sqrt(suma);
% end of the function
```

Within MATLAB try the following commands to load and run the function for a particular case:

```
clear <enter>
getf('AbsM.txt') <enter>
R = [1. 3. 4. 2. <enter>
3. -2. 5. -7. <enter>
1. 3. 4. 5. ] <enter>
AbsM(R) <enter>
```

Functions are defined throughout this book in relation to different mathematical subjects, i.e., vectors, matrices, integrals, differential equations, etc. The following sections of this chapter deal with the subjects of input/output and string manipulation in MATLAB.

Input/Output in MATLAB

In this section we present examples of input and output operations in Matlab.

Saving and loading variables.

To save all current active variables in a file use the command *save*. Let's try an example:

clear <enter>
A = [1. 2. 3.; -3. 4. 5.; 2. 4. 5.; 1. 3. 2.]; b = 1:10; <enter>
A <enter>
b <enter>
save 'DataAb.mat' <enter>

Notice that the file uses the suffix .mat. Next, using the text editor (*File>Open*, then select file) open the file *DataAB.mat* in sub-directory *work* of MATLAB. You will notice that you cannot see the numbers in the file. That is because they have been saved in a binary format. Let's clear the variables in MATLAB and re-load the values of A and b using the command *load*:

clear <enter>
who <enter>
load 'DataAb.mat' <enter>
A <enter>
b <enter>

Recall that the command *who* lists all active variables in Matlab. Function *save* will save all current active variables in Matlab with the name you have given them. Function *load* will load the variables in a *mat* file with the appropriate names.

Unformatted output to the screen

To print strings and single variables to the screen, without a format, you can use the *disp* function. The general form of the function is: *disp (sting* or *variable_name)*. Try the following examples:

x = 5; y = sin(pi*x/10); r = 1:2:25; A = rand(5,3); <enter>
disp('x = '), disp(x) <enter>
disp('y : '), disp(y) <enter>
disp('vector r is'), disp(r)
disp('matrix A is'), disp(A)

Reading from the keyboard

Reading from the keyboard can be accomplished by using the *input* function. The general form of the function call is:

A = input(prompt_string)

Where *A* is the variable to be loaded, and *prompt_string* is a message to be included to inform the user about the variable being loaded. Here is an example:

x = input('Enter the value of x:') <return>

Matlab responds with the prompt

```
Enter the value of x:
```

While leaving the cursor in front of the prompt, ready for the user to enter a value for x and press <return>. Once this operation is completed, the value entered by the user is assigned to variable ***x***.

It is possible to enter strings as input by adding a second argument 's' to the function call, e.g.,

name = input('Enter filename:','s') <return>

At the prompt:

```
Enter filename:
```

Type a string of characters, say, *c:\file1.txt*, and press <return>.

Working with files

You can write data to a file or read data from a file by using functions *fprintf* and *fscanf*, respectively. These functions are similar to their counterparts of the same name in *C*, *C++*, or *C#*. In order to access a file, however, you must first open it with function *fopen*. To close a file after reading or writing data, use function *fclose*. Examples of the use of these functions are shown next.

Function *fopen*

The general form of function *fopen* is:

file_id = fopen(filename, permission)

where *file_id* is a variable name to be used as reference to the file, *filename* is an appropriate file name reference, e.g., '*c:\file1.txt*', and *permission* is a string representing one of the following actions:

'r'	read
'w'	write (create if necessary)
'a'	append (create if necessary)
'r+'	read and write (do not create)
'w+'	truncate or create for read and write
'a+'	read and append (create if necessary)
'W'	write without automatic flushing
'A'	append without automatic flushing

There are other forms of calling function *fopen*. For more details, use the command:

```
help fopen
```

Suppose that we want to create a new file called *c:\results1.txt* so that we can write data into it. We could use to produce this file the command:

myFile = fopen('c:\results1.txt','w') <return>

Notice that Matlab responds with the output:

```
myFile =

     3
```

indicating that variable *myFile* has been assigned the value 3. If you were to open a second file, its *file_id* will be assigned the number 4, and so on.

Printing to the Matlab interface with *fprintf*

File id numbers 0, 1, and 2 correspond to pre-defined devices. For example, file id numbers 1 and 2, or no file id, produce output in the Matlab interface. Try the following examples:

fprintf(1,'hello world! \n') <return>

x = 3.2; fprintf('The value of x is %f \n',x) <return>

In these two examples we have used function *fprintf* which prints a string, i.e., the contents between quotes. Parts of the printed string are specifications such as *\n* or *%*f known as C conversion specifications. For example, the conversion specification \n indicates to start a new line, while the conversion specification %f indicates that a floating-point field should be included to take the value of variable(s) listed at the end of the string in the *fprintf* function. More details and examples of using function *fprintf* are shown below.

Function *fclose*

The general form of function *fclose* is:

status = fclose(filename)

where *status* is a numerical variable that can take the value 0 if the file closing operation is successful, or -1 if it is not successful (e.g., the file wasn't open in the first place, or the file doesn't exist). For example, having open file *c:\results1.txt*, as shown earlier, we can close this file by using either

fclose(myFile) <return>

or

fclose(3)

In both cases, Matlab responds with the result:

```
ans =

     0
```

Function *fprintf*
The general form of function *fprintf* is

fprintf(file_id, format_string, variables)

where *file_id* is a variable or number referring to a file open with function *fopen*, or to the Matlab interface (i.e., *file_id = 1* or *2*), *format_string* is a string containing characters or conversion specifications, and *variables* is an optional list of variables. The following examples show uses of function *fprintf* to print to the Matlab interface, although the same examples can be used to write to a text file.

Printing a string with function *fprintf*
Function *fprintf* can be used to print simply a string, e.g., a title for a text file:

fprintf('\n ---- Output File for Program "Open_Channel" ---- \n\n') <return>

Recall that the conversion specification *\n* produces a new line (linefeed), thus, the result of this command will be shown in Matlab as follows:

```
» fprintf('\n ---- Output File for Program "Open_Channel" ---- \n\n') [return]

 ---- Output File for Program "Open_Channel" ----

»
```

Format specifications for printing control
The "slash" format specifications are used to control the printed output. Here is the description of these specifications taken from the Matlab *help fprintf* command:

> "The special formats \n,\r,\t,\b,\f can be used to produce linefeed, carriage return, tab, backspace, and form-feed characters respectively."

Also,

> "Use \\ to produce a backslash character and %% to produce the percent character."

Here is an example using some of these format specifications:

```
» fprintf('\nUse \\n to produce a line feed \t\t and these are two tabs.\n\n')
[return]

Use \n to produce a line feed            and these are two tabs.

»
```

Format specifications for integer values
The format specification *%i* allows to replace the value of an integer variable. For example:

```
» i2 = 5; jj = -102;
» fprintf('\nInteger indices: i2 = %i and jj = %i\n\n',i2,jj)

Integer indices: i2 = 5 and jj = -102

»
```

If the variables actually contain floating-point values (i.e., values with decimal parts), the integer format specifications will be replaced by scientific notation formats, e.g.,

```
» i2 = 6.23; jj = -102.45;
» fprintf('\nInteger indices: i2 = %i and jj = %i\n\n',i2,jj)

Integer indices: i2 = 6.230000e+000 and jj = -1.024500e+002
```

NOTE: If you are not familiar with *scientific notation format*, the results above represent i2 = 6.23×10^{0} and jj = -1.0245×10^{2}. Thus, the notation *a.aa…ebbb*, in general, represents the number *a.aa…* $\times10^{yyy}$.

One can specify the width of the integer field to assign to an integer value (i.e., the number of spaces allowed), for example:

```
» i2 = 5; jj = -102;
» fprintf('\nInteger indices: i2 = %5i and jj = %10i\n\n',i2,jj)

Integer indices: i2 =     5 and jj =       -102

»
```

In this case, variable *i2* was assigned 5 positions, but only one is needed, thus 4 positions are left blank leaving a larger space between the characters = and *5* in the output. Similarly, variable *jj* requires only 4 positions for output, but 10 positions are assigned (*%5i*), thus, leaving 6 leading blanks before the value *-102*.

If you don't provide enough positions for an integer output, *fprintf* will force a minimum number of spaces to print the correct output. For example, in the following specification for variable *jj* only two positions are provided (*%2i*), but the value requires 4 positions, thus, *fprintf* forces the field to have a minimum of 4 positions.

```
» i2 = 5; jj = -102;
» fprintf('\nInteger indices: i2 = %2i and jj = %2i\n\n',i2,jj)

Integer indices: i2 =  5 and jj = -102
```

<u>*%f* format specifications for floating point values</u>

The format specification *%f* allows to replace the value of floating-point variables. For example:

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %f and y = %f\n\n',x,y)

Real variables: x = 123.450000 and y = -0.000560

»
```

Notice that the specification *%f* shows floating-point values with six decimal places. You can control the number of integer and decimal places by using the specification *%w.df*, where *w* is the total number of positions in the output (i.e., the width of the field) and *d* is the number of decimal positions. To ensure plenty of positions available for your output, make sure that $w \geq d+3$, three being the minimum number of spaces required beyond the decimal part (i.e., one position for a sign, one for an integer digit, and one for the decimal point). Here is an example using *f* formats with specified number of decimals and field width:

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %7.2f and y = %9.5f\n\n',x,y)

Real variables: x =  123.45 and y =  -0.00056

»
```

If you don't specify enough decimals for a field, the information printed may be truncated. For example, variable *y* in the following example, shows only zeros in the decimal part because only 2 decimal positions were specified:

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %7.2f and y = %7.2f\n\n',x,y)

Real variables: x =  123.45 and y =   -0.00

»
```

In the following example, even though the field for variable *x*, in principle, does not have enough width to show all the integer digits, *fprintf* expands the field width to show all of them:

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %5.2f and y = %7.5f\n\n',x,y)

Real variables: x = 123.45 and y = -0.00056

»
```

Using *d = 0* will truncate the values to be printed to their integer parts, e.g.,

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %5.0f and y = %7.0f\n\n',x,y)

Real variables: x =   123 and y =      -0

»
```

<u>*%e* format specifications for floating point values</u>

The format specification *%e* allows to replace the value of floating-point variables using scientific notation. For example:

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %e and y = %e\n\n',x,y)

Real variables: x = 1.234500e+002 and y = -5.600000e-004

»
```

The default field for scientific notation format shows one integer digit, six decimal digits, and three integer digits for the exponent. Also, positions are allowed within the field for signs on the integer part and on the exponent, and one position for the decimal point.

One can specify the width of the field (*w*) and the number of decimals *(d*) by using the specification *%w.de*, for example:

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %e and y = %e\n\n',x,y)

Real variables: x = 1.234500e+002 and y = -5.600000e-004

»
```

Since we need to allow for three positions for the exponent, one for the exponent's sign, one for the integer-part sign, one for one integer position, one for the decimal point, and one for the sign of the number, we need to have at least $w \geq d+8$.

Using *d = 0* truncates the *mantissa* (i.e., the part that is not the exponent) to an integer, e.g.,

```
» x = 123.45; y = -0.00056;
» fprintf('\nReal variables: x = %10.0e and y = %13.0e\n\n',x,y)

Real variables: x =     1e+002 and y =       -6e-004

»
```

<u>*%g* format specifications for floating point values</u>

The format specification *%g* allows to replace the value of variables according to the most appropriate format, be it integer, floating-point, or scientific notation. For example:

```
» m = 2; r = -12.232123; s = 0.00000000000023;
» fprintf('\nSee these values: m = %g, r = %g, and s = %g\n\n',m,r,s)

See these values: m = 2, r = -12.2321, and s = 2.3e-013

»
```

Notice that the *%g* specification chose an integer format for *m*, a regular floating-point format for *r*, and a scientific-notation format for *s*.

An attempt to provide for field width and number of decimals for the *%g* format, by using *%10.5g*, shows that only the width specification has an effect (i.e., keeping the fields at a width of 10 characters):

```
» m = 2; r = -12.232123; s = 0.00000000000023;
» fprintf('\nSee these values: m = %10.5g, r = %10.5g, and s = %10.5g\n\n',m,r,s)

See these values: m =         2, r =    -12.232, and s =   2.3e-013

»
```

Thus, we may as well have just used *%10g* in the specification:

```
» m = 2; r = -12.232123; s = 0.00000000000023;
» fprintf('\nSee these values: m = %10g, r = %10g, and s = %10g\n\n',m,r,s)

See these values: m =          2, r =   -12.2321, and s =   2.3e-013

»
```

_%s_ specification for strings

The *%s* format specification can be used to replace the value of a string variable. A string variable contains text. The following Matlab commands show how to load string variables and how to use the *%s* format specification to print out the values of those string variables:

```
» sname = 'Bart Simpson'; prof = 'daredevil';
» fprintf('\nHis name is %s and he is a %s.\n\n', sname, prof)

His name is Bart Simpson and he is a daredevil.

»
```

Additional information about string variables is provided below.

Combining format specifications

Control characters (i.e., *\n, \t*, etc.) and format specifications (*%i*, *%f*, *%e*, *%g*, *%s*, etc.) can be combined in a *fprintf* output string to produce output to the Matlab interface or to a file. Here is an example that combines several specifications:

```
» fprintf('\nIn iteration no. %2i, x = %g, while y = %f8.2, ...
for case: %s.\n\n', jj, x, y, title)

In iteration no.  5, x = 5.6e-010, while y = -123.4500008.2, for case: Logan
Canal.

»
```

Scripts for writing to a file

We present two scripts as examples of opening, writing to, and closing a file. In the first script we only write numerical data.

```
%Script to write numerical data to a text file
x = 0:1:10; y = sin(x*pi/8); z = exp(x);
myFile = fopen('c:\Results1.txt','w');
n = length(x);
for j = 1:n
   fprintf(myFile,'%5i %10.6f %16.8e \n',x(j),y(j),z(j));
end;
fclose(myFile);
```

Type the script into the Matlab editor and save it as *Write1.m*. Then run the script, and open file *c:\Results1.txt* using the editor. The result is the following file:

```
    0   0.000000  1.00000000e+000
    1   0.382683  2.71828183e+000
    2   0.707107  7.38905610e+000
    3   0.923880  2.00855369e+001
    4   1.000000  5.45981500e+001
    5   0.923880  1.48413159e+002
    6   0.707107  4.03428793e+002
    7   0.382683  1.09663316e+003
    8   0.000000  2.98095799e+003
    9  -0.382683  8.10308393e+003
   10  -0.707107  2.20264658e+004
```

An output like this may be useful for future data analysis. On the other hand, you may want to produce a fancier output to include in a report. The following example shows such an output. Type the following script and save it as *Write2.m*:

```
%Script to write numerical data to a text file
x = 0:1:5; y = sin(x*pi/8); z = exp(x);
myFile = fopen('c:\Results2.txt','w');
n = length(x);
fprintf(myFile,'There were %2i measurements made, with the following
results:\n\n',n);
for j = 1:n
   fprintf(myFile,'For time %5i sec., the tide elevation was %10.6f m,\n',
x(j),y(j));
   fprintf(myFile,'while the available energy was %16.8e J. \n',z(j));
end;
fclose(myFile);
```

After running this script, open file *c:\Results2.txt* to see the following results:

```
There were  6 measurements made, with the following results:

For time     0 sec., the tide elevation was   0.000000 m,
while the available energy was  1.00000000e+000 J.
For time     1 sec., the tide elevation was   0.382683 m,
while the available energy was  2.71828183e+000 J.
For time     2 sec., the tide elevation was   0.707107 m,
while the available energy was  7.38905610e+000 J.
For time     3 sec., the tide elevation was   0.923880 m,
while the available energy was  2.00855369e+001 J.
For time     4 sec., the tide elevation was   1.000000 m,
while the available energy was  5.45981500e+001 J.
For time     5 sec., the tide elevation was   0.923880 m,
while the available energy was  1.48413159e+002 J.
```

Reading from a data file

The following example shows a simple way to read data from the file *Results1.txt* created above. The file contains a matrix of data consisting of 11 rows and 3 columns. The following script will read the data from the file and separate the columns into three variables, namely, *x*, *y*, *z*. Save the script under the name *Read1.m*:

```
%Script to read data from file "c:\Results1.txt"
%created with script "Write1.m"
fclose('all'); %Ensures all files are closed
myFile = fopen('c:\Results1.txt','r');
R = fscanf(myFile,'%5i %10f %16e',[3,11]);
fclose(myFile);
R = R';%Transpose to make it look like original table
%The next line extract columns
x = R(:,1); y = R(:,2); z = R(:,3);
```

After executing the script, try the following Maple commands to see the data recovered from the file:

```
» R

R =

  1.0e+004 *

         0         0    0.0001
    0.0001    0.0000    0.0003
    0.0002    0.0001    0.0007
    0.0003    0.0001    0.0020
    0.0004    0.0001    0.0055
    0.0005    0.0001    0.0148
    0.0006    0.0001    0.0403
    0.0007    0.0000    0.1097
    0.0008         0    0.2981
    0.0009   -0.0000    0.8103
    0.0010   -0.0001    2.2026
```

Because of the different orders of magnitude in the data read, the matrix *R* is shown as being multiplied by the factor *1.0e+004* (i.e., 1×10^4). To see the numbers in more detail use:

```
EDU» format long
EDU» R

R =

  1.0e+004 *

                  0                   0   0.00010000000000
   0.00010000000000   0.00003826830000   0.00027182818300
   0.00020000000000   0.00007071070000   0.00073890561000
   0.00030000000000   0.00009238800000   0.00200855369000
   0.00040000000000   0.00010000000000   0.00545981500000
   0.00050000000000   0.00009238800000   0.01484131590000
   0.00060000000000   0.00007071070000   0.04034287930000
   0.00070000000000   0.00003826830000   0.10966331600000
   0.00080000000000                   0   0.29809579900000
   0.00090000000000  -0.00003826830000   0.81030839300000
   0.00100000000000  -0.00007071070000   2.20264658000000
```

Use the following commands to see the separated columns:

x <return> y <return> z <return>

The following are things to keep in mind when using the *fscanf* function to read data in the form of tables (i.e., data that can be stored in a matrix):

1 Make sure that you know the format in which the data is stored in the input file.

2 Make sure that the file is open with function *fopen* and that it uses the specification `'r'` for reading.

3 The general form of the call to function *fscanf* is:

A = fscanf(file_id, format, size)

where *file_id* is the reference to the file being read, *format* is a format string describing the format of the data columns, and *size* is a vector of the form *[n,m]* where *n* is the number of columns and *m* is the number of row in the data matrix.

4 The matrix *A* that is loaded will be transposed with respect to the data matrix in the file. Thus, a statement like *A = A'* may be used to convert matrix *A* to the same shape as the data in the file.

5 After matrix *A* has been loaded, the different columns can be separated as in the script shown above.

Manipulating strings in MATLAB

A string is basically text that can be manipulated through MATLAB commands. Strings in MATLAB are written between single quotes. The following are examples of strings:

```
'myFile' 'The result is: ' 'a b c' 'abc' 'a' 'b' 'c'

'Text to be included' 'Please enter the graphic window number' '1' '3' '5'
```

String concatenation

The joining of two or more strings is called *concatenation*. Function *strcat* can concatenate tow or more strings into a single one. In the next example variables s1, s2, and s3 are defined and concatenated:

```
» s1 = 'The result from ';
» s2 = 'multiplication ';
» s3 = 'is given below.';
» sOut = strcat(s1,s2,s3)

sOut =

The result frommultiplicationis given below.
```

Notice that function *strcat* chops off trailing spaces, however, it keeps leading spaces in strings. Thus, a better result is obtained by using;

```
» s1 = 'The result from ';
» s2 = ' multiplication ';
» s3 = ' is given below.';
» sOut =

The result from multiplication is given below.
```

Alternatively, we could use the first version of variables *s1, s2, s3* and concatenate them by using square brackets as shown next:

```
» s1 = 'The result from ';
» s2 = 'multiplication ';
» s3 = 'is given below.';
» sOut = [s1, s2, s3]

sOut =

The result from multiplication is given below.
```

Other string functions and operations

Function *length* determines the number of characters in a given string, for example:

```
>>length(sOut)
ans =
46.
```

The function *findstr*, with a typical call of the form *findstr (string1, string2)* determines the position of the first occurrence of sub-string *string2* within *string1*. For example,

```
>>findstr(sOut,'mult')
ans =
17.
```

Once the position of a sub-string has been determined you can use sub-indices to extract parts of a string. For example, next we extract characters 17 to 24 out of string *sOut*:

```
>>sOut(17:24)
ans =
multipli
```

Function *strrep*, with typical call of the form *strrep(s1,s2,s3)* , replaces sub-string *s2* with sub-string *s3* within string *s1*, for example:

```
» sOut = strrep(sOut,'multiplicat','divis')
sOut =
The result from division is given below.
```

Converting numerical values to strings and vice versa

The function *num2str* is used to convert a numerical result into a string. This operation is useful when showing output from numerical calculations. For example, the next MATLAB input line performs a numerical calculation, whose immediate output is suppressed by the semi-colon, and then produces an output string showing the result. The output string produced consists of the sub-string "`The sum is`" concatenated to the numerical result that has been converted to a string with `string(s)`.

```
» s = 5+2; ['The sum is ',num2str(s)]

ans =

The sum is 7
```

If a string contains a numerical value that needs to be used in numerical operations, for example, it is necessary to convert such string into a number with function *str2num*. For example:

```
» r = str2num('123.4')

r =

  123.4000

» sqrt(r)

ans =

   11.1086
```

Each character has a numerical value associated with it. The numerical value of a character is given by its ASCII code, which can be obtained by using function *double*. For example, here is the list of ASCII numerical codes for the characters *0*, *1*, *a*, *b*, and *A*:

```
» [double('0'),double('1'),double('a'),double('b'),double('A')]

ans =

    48    49    97    98    65
```

Function *char*, on the other hand, produces the character corresponding to a given numerical code, for example:

```
EDU» char(102)

ans =

f

EDU» char(125)

ans =

}
```

The variable *ans*

The variable *ans* (*ans*wer) contains MATLAB's current output. You can refer to the last MATLAB output by using the variable name *ans*. For example, the following commands uses the contents of *ans* to operate on the most recent MATLAB output:

```
>>3+2
ans =
5.

>>exp(ans)
ans =
148.41316
```

To verify that the result obtained is correct use:

```
-->exp(5)
ans =
148.41316
```

Exercises

[1]. Write a MATLAB function to calculate the factorial of an integer number:
n! = n .(n-1) .(n-2)…3 .2 .1

[2]. Write a MATLAB function to calculate the mean value ($\bar{x}$) and the standard deviation (*s*) of the data contained in a vector ***x*** = *[x1 x2 … x3]*. The definitions to use are the following:

$$x = \frac{1}{n}\sum_{i=1}^{n} x_i \ , \ s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

Determine the mean and standard deviation of ***x*** = *[2.3, 1.2, 1.8, 2.5, 0.7, 1.6, 1.2, 0.8]*.

[3]. Write a MATLAB function to calculate the function defined by

$$f(x) = \begin{cases} |\, x^2 + 1\,|, -1 < x \le 1 \\ \sqrt{x^2 + 1}, 1 < x \le 2 \ \Sigma \\ (x^2 + 1)^3, 2 < x \le 4 \end{cases}$$

Plot the function for 0 < x < 3.

[4]. Write a MATLAB function that request from the user the values of the bottom width (b) and water depth (*y*) for a rectangular cross-section open channel and prints the area ($A = bh$), wetted perimeter ($P = b+2h$), and hydraulic radius ($R = A/P$) properly labeled. Try the function for values of b = 3.5 and y = 1.2.

[5]. Write a MATLAB function that request from the user the values of the initial position (x_0, y_0) of a projectile, the initial velocity given as a magnitude v_0, and an angle θ_0, and the acceleration of gravity *g*. The function also requests from the user an initial time t_0, a time increment Δt, and an ending time t_f. The function produces a table of values of the velocity components $v_x = v_0 \cos(\theta_0)$, $v_y = v_0 \cos(\theta_0)$, the magnitude of the velocity, $v = (v_x^2+v_y^2)^{1/2}$, the position of the projectile, $x = x_o + v_0 \cos(\theta_0)t$, $y = y_o + v_0 \sin\theta_0)t - gt^2/2$, and the distance of the projectile from the launching point, $r_0 = ((x-x_0)^2 +(y-y_o)^2)^{1/2}$. The function also produces plots of *x - vs. - t*, *y - vs. - t*, r_0 *- vs. - t*, and *y - vs. -x* in different graphic windows. [Note: to generate a new graphics window use the MATLAB command `>>figure(j)` where *j* is the window number.]

[6]. Suppose you want to plot the function $r(\theta) = 3.5(1 - \cos(\theta))$. Write a MATLAB function that generates values of θ from *0* to 2π, calculates the values of *r*, and the Cartesian coordinates $x = r\cos(\theta)$, $y = r \sin(\theta)$, and prints a table showing those values, i.e., θ, *r*, *x*, and *y*. The function also produces a plot of *y-vs.-x*.