We note the following: At the beginning of the program the values for the constants are `a = 4`, `b = 2`, and `c = 0.47`. These values are passed to the function `Test13` and they are displayed first after the text line `Variables in Test13 are:`. Then, they are changed and displayed from within the function `Test13` as:

`a = 8`, `b = 20`, and `c = 47`

Then we exit the function `Test13` and return to the main program where the variable values are displayed. We see that the values now are the same that were changed in the function `Test13`.

This way to pass variables is called **call by value**. We may pass them to a function but let them unchanged in the main program.

Global Variables

In the previous section we saw how we can pass variables between a main program and a function. We learned that values do not pass just by giving the same name to variables. They need to be passed by value in order to be used in another function or in a main program. Variables defined in this way are called local variables. This may be convenient sometimes when we wish to use the same name for different variables, and we do not wish to change the variable values. When we define a variable, MATLAB assigns it as a local variable. Unless the programmer wishes to use in another function with the same value, we have to remove the local variable restriction and declare it as a global variable. In this way, every time we refer to a given variable previously declared as a global one, it is going to have the same value, and if this value is changed, it is going to change in any other function and main program where the variable is declared as global. This means that if a function does not declare it as a global variable, it will not have the global variable value in that function. The instruction to declare a variable as global is

```
global a b
```

This means that variables `a` and `b` are global variables.

Example Program with a global variable
To see how global variables work, let us consider the variable `a = 3`. We wish to have this variable in a function and in a main program. The following file `six_14.m`, which includes the script and the functions `change1` and `change2`, shows this:

```
% File six_14.m
% Program to check local and global variables.
```

```
global a
a = 3;
fprintf('Value of "a" before function change1 a = %g ', a)
change1
fprintf('Value of "a" after function change1 a = %g \n', a)
change2()
fprintf('Value of "a" after exiting change2 a = %g \n', a)

function x = change1()
  global a
  fprintf('Value of "a" after entering change1 a = %g.\n', a)
  a = 7;
  fprintf('Value of "a" modified in change1 a = %g.\n', a)
end

function x = change2()
  fprintf('Value of "a" after entering change2 a = %g.\n', a)
  a = 12;
  fprintf('Value of "a" modified in change2 = %g.\n', a)
end
```

Now we run the file `six_14`,

```
>> six_14
   Value of "a" before entering function change1 a = 3.
   Value of "a" before after entering function change1 a = 3.
   Value of "a" modified in change1 a = 7.
   Value of "a" after returning from function change1 a = 7.
   Value of "a" entering change2 a = 7.
   Value of "a" before exiting change2 a = 12.
   Value of "a" after returning from change2 a = 7.
```

As we see in the main program and in the functions, the variable `a` is global in the main program and in function `change1` and it is only local in `change2`. When we run the program, the value of `a` is passed to `a` in the instruction `global a`. There it keeps the same value until we change it to `a = 7`. It passes this value to the main program using again the instruction `global a`. The value of `a` is passed as an argument in the function `change2`. There we change its value to `a = 12`. When we exit this function, the value of `a` is not changed in the main program because it is a local variable in `change2`.

6.5.2 The Instruction `return`

In general, a function ends with the last instruction, but sometimes we need to end the function before the last instruction and `return` to the main program or function that called the function. We can do this with the instruction `return`. When the sequence of instructions finds a `return`, it ends the function and it exits it, returning to the function or program that called it. We present an example to show how the instruction `return` works.

Example Use of `return` in a function
Let us consider the script `six_15.m` which uses the function `greater_smaller` to find out if a number is greater than or smaller than 0.

```
% File six_15.m
x = input ('Enter the value of x: \n');
greater_smaller (x);
fprintf (' The run ends. \n')

function greater_smaller (x)
  if x > 0
     fprintf (' x is less than 0. \n')
  elseif x > 0
     fprintf (' x is greater than 0. \n')
     return
  else
     fprintf (' x is equal to 0. \n')
end
```

Now we have several runs to show how the instruction `return` works.

```
>> six_15
   Enter the value for x:
   0
   x is equal to 0.
   The run ends.
>> six_15
   Enter the value of x:
   3
   x is greater than 0.
   The run ends.
>> six_15
   Enter the value of x:
   -3
   x is less than 0.
   The run ends.
```

We observe the following: When `x` is greater than or smaller than 0, the function finds a `return` instruction and interrupts it before executing the remaining instructions. The control is passed to the main program which prints `The run ends`. If `x = 0` the function ends the `if` statement and then goes to the main program. In this last case the function is executed to the last instruction.

The Instructions `nargin` and `nargout`

The instructions `nargin` and `nargout` are used to find out the number of input and output arguments in a function, respectively. For example, for the function in Example 6.15, we have `nargin = 1`, `nargout = 1`. `nargin` is the acronym for Number of ARGuments in the INput and `nargout` for Number of ARGuments in the OUTput. `nargin` and `nargout` are variables.

These two instructions are useful when we have a function that allows the branching to different parts of it depending on the variables when calling a function. For example, a function to solve a quadratic equation, but where the user only gives two coefficients might branch to a section of the function to solve a first order equation.

Recursive Functions

A recursive function is a function that calls itself. This property is available for MATLAB functions. We show an example to show the recursivity in MATLAB functions.

Example Recursive evaluation of the factorial function
The simplest example of a recursive function is the factorial function. As we already know, the factorial of a non-negative integer n is defined as

$$n! = 1 * 2 * ... * n$$

This can be rewritten as

$$n! = n \times (n - 1)!$$

In Example 6.15 we saw that the factorial is evaluated by the function `factorial1`. Using recursion the function can be written as:

```
function x = fact_rec(n)
if n >= 1
   x = n*fact_rec(n-1);
else
   x = 1;
end
```

When we run this program we get

```
>> fact_rec(6)
   ans =
   720
```

We see that the function is calling itself and producing the expected result.

File Management

Up to this point, input data has been entered through the keyboard. The keyboard is thus an input device. The results are usually displayed on the `Command Window` or in a `Figure` window. Thus, the computer screen is the output device. Another way to give input data is by using a file where we have somehow stored the input data. This file can be created by MATLAB or by any other computer program. This last option allows data exchange between MATLAB and any other software package. For example, a spreadsheet such as Excel might exchange data with MATLAB, so they can be processed and visualized. In the same way, data generated in MATLAB can be used by other programs. An obvious advantage of saving data in a file is that we can use it in a later MATLAB session or we can use it in another computer or send it to another user.

TABLE Permit codes to open files

Permission	Action
‘r’	Opens the file to read. The file must already exist. If it does not exist, it sends an error message.
‘r+’	Opens the file to read and write. The file must already exist. If it does not exist, it sends an error message.
‘w’	It opens the file to write. If the file already exists, it deletes its contents. If it does not exist, it creates it.
‘w+’	It opens the file to read and write. If the file already exists, it deletes its contents. If it does not exist, it creates it.
‘a’	It opens the file to write. If the file already exists, it appends the new contents. If it does not exist, it creates it.
‘a+’	It opens the file to read and write.

File Opening and Closing

To read or write data to a file we have first to open it. MATLAB can open a file using the instruction `fopen` that has the format:

```
fid = fopen (file name, permissions)
```

TABLE File handles

Handle fid	Meaning
-1	Error when opening the file. Usually, when we want to read from a non-existent file.
0	Standard input. Usually, the keyboard is always open to write (‘`r`’).
1	Standard output. Usually, the Command Window. Always open with permission to add data at the end of the existing one.
2	Standard error. Always open with permission to add data at the end of the existing one.

Here, `file name` is the file name and it must exist in order to open it. If we want to write to a non-existing file, the file is created and then the data is written in. `Permissions` is a variable that specifies how the data is written to the file. The permission codes available are given in Tables 6.3 and 6.4. We can open as many files as we wish. `fid` is the handle that is used to identify the file. The handle values start with 3. Handle values from -1 to 2 have a special use in MATLAB and they are given in Table 6.4. After using the data from a file, we must close it. The instruction to close files is `fclose`. The format is:

```
status = fclose(fid)
```

The variable `fid` has the handle's value for the file we wish to close. The variable `status` is another handle that indicates if the file was successfully closed. The result `status = 0` indicates that it was closed, while `status = -1` indicates that the file was not closed.

Example **Use of** `fopen`.
Let us suppose that we want to open a file called `Example_six_17.txt`. Then we use:

```
handle_file = fopen('Example_six_17.txt', 'r');
handle_file
handle_file =
-1
```

The value of `handle_file` is `-1` because the file does not exist. If we use ‘`w`’ instead of ‘`r`’, we get:

```
handle_file = fopen('Example_six_17.txt', 'w');
handle_file
handle_file =
3
```

The new handle value is 3 indicating that before writing to file `Example_six_17.txt`, the file had to be created because it did not exist before. We now create another file `data.txt` with:

```
handle_file2 = fopen('data.txt', 'w')
handle_file2 =
4
```

The handle for this file is 4 because handle values are sequentially assigned. In the `Current Folder` window we see that these files were created. We can use any word processor, such as Office Word, the Notepad or the WordPad, and even we can open a text file with the MATLAB editor, to see their contents.

To create a file in another directory different from the one we are working on we only have to specify the path. For example,

```
File_name = fopen('C:\new\file.txt', 'w')
File_name = 5
```

To close a file we use the instruction `fclose` whose format is

```
status = fclose(fid)
```

`fid` is the handle corresponding to the file we wish to close. The value of the variable `status` indicates if the file was closed successfully (`status = 0`). If this was not the case and the file was not closed, MATLAB sends an error message to let us know that the file could not be closed. Before closing MATLAB, it is convenient to close all files open during the session. This avoids losing the data stored in them during the session.

Writing Information to a File

The simplest way to write information to a file is with the instruction `fprintf`. We have used this instruction before to display output data in the computer screen through the MATLAB `Command Window` as:

```
fprintf ('Display this text to the screen')
Display this text to the screen
```

To write to a file we can use this same instruction. Let us create two new files with,

```
handle1 = fopen('file1.txt', 'w');
handle2 = fopen('file2.txt', 'w');
```

To write to a file we need to use its handle as in:

```
fprintf (handle1, 'We write here to file1.txt \n');
fprintf (handle2, 'We now write here to file2.txt \n');
```

To see what is written in them we can open them either with the Notepad or Worpad or any text editor, but first we close them with:

```
fclose(handle1);
fclose(handle2);
```

Reading and Writing Formatted Data

MATLAB allows users to read and write formatted data to a file. We show the procedure with an example.

Example Writing formatted data
Suppose we want to write data about the planets in the solar system. The data we wish to write is:

1. Name 7-character string.
2. Position 2-digit integer number.
3. No. of moons 2-digit integer number.
4. Diameter 10-digit floating number.

If data in a planet name is less than the seven characters indicated, the remaining characters are filled with blank spaces. The elements of each item are stored in an array. The data is:

```
Planet name = ['Mercury'; 'Venus  '; 'Earth  '; 'Mars   '; ...
   'Jupiter'; 'Saturn '; 'Uranus  '; 'Neptune']
Position = [ 1; 2; 3; 4; 5; 6; 7; 8];
No_of_moons = [ 0; 0; 1; 2; 63; 34; 21; 13];
Diameter_in_km = [ 4880; 12103.6; 12756.3;...
   6794; 142984; 120536; 51118; 49532];
```

Note that we are writing the data as column vectors. To see the data for the second planet we write:

```
>> fprintf ('%s\n%g\n%g\n%g\n', Name(2,:), Position(2,:),...
     No_of_moons(2,:), Diameter_in_km(2,:));
```

To obtain:

```
Venus
2
0
12103.6
```

To write this information to a file we use the following script:

```
% File Example_6_18.m
%
handle_planets = fopen('Planets.txt', 'w');
for i = 1 : length(Position);
   fprintf(handle_planets,'%7s ,%5d ,%2d ,%2d \n', ...
     Planet_name(i, :), Position(i, :), ...
     No_of_moons(i, :), Diameter_in_km(i, :));
end
fclose(handle_planets);
```

We can now read the data in a file with a script or a function written in m-language. We have to check for several things when reading this data:

1. If we have gotten to an `end_of_file` which has the variable name `feof`.
2. Read each string with `fscanf` and assign it to its corresponding field.
3. Close the file after we find the `feof`.

The m-file is:

```
% This is file read_data.m
%
handle_data = fopen('Planets.txt', 'r');
% We define the names of the column vectors.
Names = [ ];
Positions = [ ];
Moons = [ ];
Diameters = [ ];
while ~feof(handle_data)
   % Read the planet name
   stringg = fscanf(handle_data, '%7c', 1);
   Names = [Names; stringg];
   comma = fscanf(handle_data, '%1c', 1);
```

```
    % Read the position
    number = fscanf(handle_data, '%5d', 1);
    Positions = [Positions; number];
    comma = fscanf(handle_data, '%1c', 1);
    % Read the number of moons
    number = fscanf(handle_data, '%5d', 1);
    Moons = [ Moons; number];
    comma = fscanf(handle_data, '%1c', 1);
    % Read the diameter
    number = fscanf(handle_data, '%12e', 1);
    Diameters = [ Diameters; number];
    end_of_line = fscanf(handle_data, '%1c', 1);
end
fclose(handle_data);
```

The instruction `while` checks for the `end_of_file`. The instruction `fscanf` searches for the planet name characters using the format `%7c`. It reads the first seven characters including the blank spaces. If we have used instead the format `%7s`, only the characters are read and the blank spaces are ignored. The 1 after `%7c` means that only an element is read. In this way, `stringg = fscanf(handle_data, '%7c', 1)` does the following: reads an element of the open file which has the handle `handle_data` and places it in the variable `stringg`. The line `comma = fscanf(handle_data, '%1c', 1)` indicates that after reading the first variable, a comma is read. We do not do anything with the comma but we need to read it. Otherwise it is read by the next instruction `fscanf`.

For the variables `Positions`, `Moons`, and `Diameters`, we need to read a numerical value. For this we use `number = fscanf(handle_data, '%5d', 1)`. After reading the data, we arrive at the end of the line. This is read with a special character `end_of_line = fscanf(handle_data, '%1c', 1)`. Again, we do not need this character but we need to read it, for the same reason we did with the comma. To see how this file works, we run the file and see the variables:

```
>> read data
>> who
```

We now see the data:

```
>> Names
Names =
Mercury
Venus
Earth
Mars
```

```
Jupiter
Saturn
Uranus
Neptune

>> Positions
Positions =
1
2
3
4
5
6
7
8

>> Moons
Moons =
0
0
1
2
63
34
21
13

>> Diameters
Diameters =
1.0 e+005 *
0.0488
0.1210
0.1276
0.0679
1.4298
1.2054
0.5112
0.4953
```

As we can see, these are the values entered before.

TABLE Options for variable precision

Option	Meaning
'char'	8-bit characters. Used for text.
'short'	16-bit integers. Integer numbers in the range from -215 to 215-1.
'long'	32-bit integer (2's complement). Integers in the range from -231 to 231-1.
'ushort'	Unsigned integer (16 bits).
'uint'	Unsigned integer (32 bits).
'float'	Single precision floating point real numbers (32 bits).
'double'	Double precision floating point real numbers (64 bits).

Reading and Writing Binary Files

So far, we have used alphanumeric data that can be read by any text processor besides the MATLAB editor. This type of data is called ASCII data. A disadvantage of this type of data is the size of the files when we handle a large amount of data. An alternative way is to store data in a binary format. This is a more efficient way to store information. Unfortunately, information stored in a binary format cannot be read by a text processor. To write and read in a binary format we use the instructions `fwrite` and `fread`, respectively. The format for `fwrite` is:

```
count = fwrite(handle, A, 'precision')
```

The variable `A` contains the data to be written. `count` is the number of elements that were successfully written. `handle` is the handle for the opened file. `precision` gives information about how we want to store the information. Some of the options for this variable are given in Table 6.5.

Example **Reading and writing binary data**

Let us suppose that we want to write the following data in binary format:

$$A = \begin{bmatrix} 57 & 10 \\ 14 & 75 \end{bmatrix}$$

```
B = 27, C = 'MATLAB'
```

This data is entered as:

```
>> A = [57 10; 14 75];
>> B = 27;
```

```
>> C = 'MATLAB';
```

Then we open the file

```
>> fid_binary = fopen('binary.dat', 'w')
   fid_binary = 3
```

To write we use

```
>>  fwrite(fid_binary, A, 'double')
   ans =
   4
```

The answer is 4 indicating that 4 elements were written. These elements belong to matrix A. We now write in B:

```
>> fwrite(fid_binary, B, 'short')
   ans =
   1
```

This time the result is a 1 because B has only one element. We now continue with C:

```
>>  fwrite(fid_binary, C, 'char')
   ans =
   6
```

The answer is 6 because the word MATLAB has 6 elements. We now close the file

```
>> fclose(fid_binary)
```

Now we try to read the file with the WordPad and we get the results shown in Figure 6.2. As we can see, the data is not what we wrote to the file because it is in binary format and not in ASCII format, thus trying to read it with a text processor does not display the information stored in it. To read the data, first we open the file:

```
>> fid_bin = fopen('binary.dat', 'r')
   fid_bin = 3
```

And now we use `fread`, whose format is:

```
>> [A, count] = fread(fid_binary, [2, 2], 'double')
```

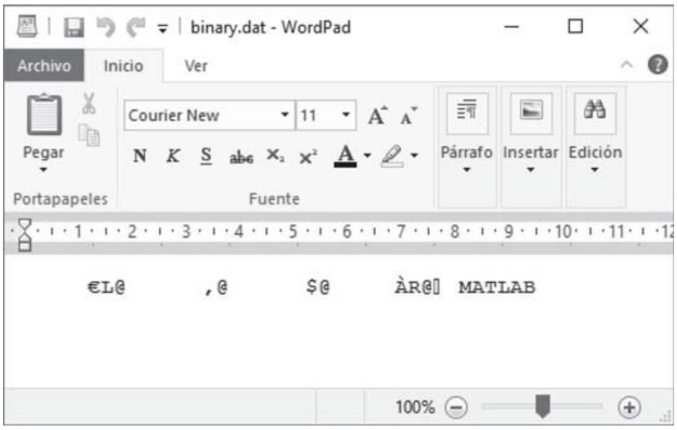

FIGURE 6.2: WordPad window showing the contents of `binary.dat`.

where `A` is the matrix name and `count` is the number of elements (4 in the case of matrix `A`). If `A` were an 8 × 4 matrix, then instead of [2, 2] we should write the size [8, 4]. The results are:

```
>> [A, count] = fread(fid_bin, [2, 2], 'double')
>> A =

   57 10
   14 75
   count =
   4
```

Data must be read in the same order and with the same format that it was written. To read out the remaining variables we use:

```
>>  B = fread(fid_bin, [1], 'short')
    B =
    27

>> C = fread(fid_bin, [6], 'char')
   C =
   77
   65
   84
   76
   65
   66
```

Note that `C` is an ASCII string in a column vector. To change it to a row vector in characters we transpose it and then use `setstr` as in:

```
>> C = setstr(C')
   C =
   MATLAB
```

Finally, we close the file:

```
>>  fclose(fid_bin)
   ans = 0
```

This value indicates that the file was successfully closed.

Passing Data Between MATLAB and Excel

A software package used in engineering, science, and finance is Excel. Excel and MATLAB can read and write data to files. In this section we show how such files can be used by any of the packages. For example, MATLAB can write data separated by commas in files with extension `csv`, for comma separated values, and then read by Excel, and vice versa.

Exporting Data to Excel

To show how we can export data from MATLAB to Excel we have the following example.

Example Exporting data to Excel from MATLAB
Let us consider the following data about the five countries with the largest territories in square miles in the American continent together with their capital cities:

```
Canada, Ottawa, 3849660
United States of America, Washington D.C., 3787319
Brazil, Brasilia, 3300410
Argentina, Buenos Aires, 1073596
Mexico, Mexico D.F., 759589
```

This data is written by MATLAB to file `countries.csv`. The following file opens the file `countries.csv`, writes the data, and closes the file. Each country name must have the same number of characters. Each capital name must have ten characters, including blank spaces.