Chapter 4
Start Programming

This chapter outlines the Basic programming concepts in MATLAB such as loop generation and program flux control. Programming *style and debugging techniques are also presented.*

M-Scripts and Functions

When we manage long and complex lists of operations, it is inconvenient to type them directly into the MATLAB prompt. It is preferable in such cases to write scripts, which are text files containing lists of commands. Because scripts are text files, they can be written in any text editor. However, MATLAB provides a built-in text editor that offers many advantages over conventional text editors. For example, the MATLAB editor highlights the commands, automatically indents the script, shows where loops start and end, identifies rows with numbers to aid in locating errors, etc. This editor can be opened by typing `edit` at the MATLAB prompt or by selecting File->New -> M-file.

The Fig. 4.1 highlights some of the key icons of the MATLAB editor [1] (e.g., open a file, create a new script, save a script, run). Type into the editor the script shown in the figure (it is also presented in the text) and save it with the name MyFirstProgram.m (do not type the numbers, they refer to the script lines).

Listing 4.1

```
1 % MyFirstProgram
2 %
3 % This is my first program. These lines are used as comment
4
5 x=5;
6 y=cos(x);
7 STR=sprintf('The cosine of %2.2f is %2.2f',x,y);
8 disp(STR);
```

[1] The appearance of the window can differ depending on Matlab version you are using.

M. Borgo et al., *MATLAB for Psychologists*,
DOI 10.1007/978-1-4614-2197-9_4, © Springer Science+Business Media, LLC 2012

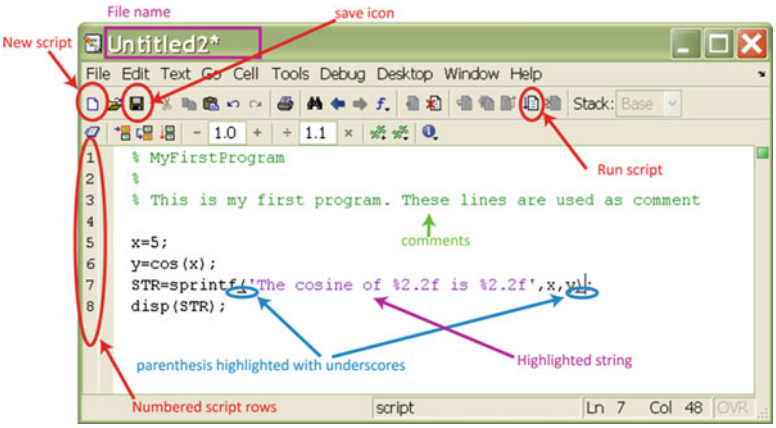


Fig. 4.1 MATLAB editor window

The semicolon after each command prevents MATLAB from echoing the command output in the command window. This echoing can be annoying, especially when the echoed content is large. To insert comments in your script, just type `%` at the beginning of the line you want to comment; this line is not executed by MATLAB. Note that comments are colored green in the MATLAB editor. Comments can also be written in the same row of a MATLAB command, as in the following case:

```
>> a=1; % the comment can be written at the right of a MATLAB command
```

Once you have written a script, you can save it with any name you like. However, we recommend saving scripts with meaningful filenames. Moreover, we recommend starting the script with some comments explaining what the script does. These comments are very useful especially when you open the script after a period of time. MATLAB scripts (M-scripts) are saved with the *.m* extension in the directory where you are working. Once the script is saved, the script name appears in the editor window title.

To run/test the M-script, press the run icon or the F5 key of the keyboard. The example above should output in the command window the following text:

```
The cosine of 5.00 is 0.28
```

It is possible to run a script directly from the command window by typing the filename of the script. In our case we should type:

```
>> MyFirstProgram
The cosine of 5.00 is 0.28
```

An M-script can be saved in any directory of your computer; however, to run the script from the MATLAB prompt you need either to be in the directory where you saved the script or to tell MATLAB where to find your M-script. MATLAB looks for scripts within the directory where you are working and within any directories

defined in the search path. There are two possible ways to add the directory where you are working to the MATLAB default search path:

1. Click on the FILE menu, and then select the SET PATH option.
2. By means of the `path` command from the MATLAB command window or within an M-script file. The `path` command uses the following syntax: `path(path,'newpath')` (for example `path(path,'C:\myFiles')`).

Let's see how to write an M-script file that draws a graph so that every time we want to plot the same type of graph we do not need to retype all the commands. We do this in the following example. Save the script with the name plotLC in the current directory.

Listing 4.2

```
1  % plotLC
2  % Plot the learning curve.
3
4  close all;
5  clear all;
6  x=[1:10];
7  y=10./exp(x);
8  figure;
9  plot(x,y,'rd-');
10 xlabel('trials');
11 ylabel('ErrorRate');
12 grid;
13 MyFirstPrograms;
```

At the beginning of the script we have inserted the commands `close all` and `clear all` to close all figures and then to be sure that no variable can affect the script behavior. After this script is run, a figure appears, and the sentence `The cosine of 5.00 is 0.28` appears in the MATLAB command window.

MATLAB is provided with an excellent debugger to warns us if we have made any errors. Let's suppose, for example, that at line 7 of the script we have forgotten to add the dot after the number 10, in this way.

```
7  y=10/exp(x);
```

In this case, MATLAB returns the following error:

```
??? Error using ==> mrdivide
Matrix dimensions must agree.
Error in ==> PlotLR at 7
y=10/exp(x);
```

The debugger explains what type of error terminated the script prematurely, the filename of the corrupted script, and the line where the error is located (unfortunately, errors are not always so easy to pinpoint).

Control Flow Statements

In the previous scripts (i.e., Listing 4.1 or 4.2), MATLAB executed the instructions as they were typed directly in the command window. This programming style, however, is not always efficient. For example, it does not solve such problems as the execution of repetitive tasks and the control of operation flow. MATLAB, like other programming languages, has flow control statements, which we now explain.

Cycles and Conditionals: If

The `if` command is used to control the execution of a script or a function (see later in this chapter). Frequently, the `if` command is used together with the `else` command in the following form:

```
if condition
     DoSomething
else
     DoSomethingElse
end
```

The `if` command checks whether `condition` is satisfied (i.e., whether it is true or false). If the condition is true, MATLAB executes `DoSomething`, while if the condition is false, MATLAB executes `DoSomethingElse`, which represents the alternative statements. Note the indentation of the text. The indentation helps in visualizing the scope of the *if – else – end* structure. Let's see an example with the following script:

Listing 4.3

```
% Test the if statement

a = 0;
if a > 1
    disp('true')
else
    disp('false')
end
```

Save the script with the name TestIf1 and run it (by typing the name in the command window or by clicking the Run button in the editor window); the word "false" appears in the command window:

```
>> TestIf1
false
```

Because 0 is smaller than 1, the comparison between the variable `a` and the number `1` returns the value false. The condition of an `if` command very often contains a combination of relational and logical operators (see Chap. 2), which returns a logical result: true or false.

Now, try to replace line 4 with the following statement and run the script again:

```
4  if 1
```

the result is also "true." Why this counterintuitive result? We have to keep in mind that the condition after the `if` statement is a logical value (true or false). In MATLAB, every numerical value different from zero is interpreted as true (see Chap. 2). Therefore, the condition "if 1" is always true. By the same token, the condition becomes false if you replace "1" with "0".

In many circumstances you can have more than two conditions to test. In such cases you can use `elseif`:

```
if condition1
      Statements1
elseif condition2
      Statements2
elseif condition3
      Statements3
else
      Statements4
end
```

if *condition1* is false, MATLAB tests *condition2*. If it is false, MATLAB tests *condition3*, etc.

As we wrote in the previous chapter, be careful when you want to verify whether a value is within a certain range. For example, if you want to display the string 'mean score' when the `MemoryScore` variable is between 50 and 100, you cannot write the if statement as follows:

```
1  MemoryScore = 10;
2  if 50 < MemoryScore<100
3      disp('mean score');
4  end
```

If you save and run the above script (use the TestIf2 name), the result is

```
>> TestIf2
mean score
```

which is wrong. As we have seen in the previous chapters, the statement in line 2 is always true (the output of `50<MemoryScore` can be either `0` or `1`, which are both less than `100`). The appropriate way to write the statement is the following:

```
2 if (50 < MemoryScore) & (MemoryScore < 100)
3     disp('mean score')
4 end
```

Or alternatively, you can write the script in a *nested* way:

```
2 if 50 > MemoryScore
3     if MemoryScore < 100
4         disp('mean score');
5     end
6 end
```

In this nested structure, when the first `if` statement is true, everything between `if` and `end` is executed. Then at line 3, there is another `if`. In case of a true condition, everything between the `if` (on line 3) and the associated `end` (on line 5) is executed, and so on. On the other hand, in the case of a false condition, MATLAB skips to the line after the `end` on line 5, which is the `end` on line 6.

Switch Case

An alternative to the *if – else* form is the *switch – case* form. The advantage of the *switch – case* form is that in some situations, it yields a code that is more readable. The *switch – case* form has the following syntax:

```
switch condition
    case fact1
         Statements1
    case fact2
         Statements2
    case fact3
         Statements3
    otherwise
         StatementsOtherwise
end
```

the condition after the switch command is evaluated and compared successively with *fact1*, *fact2*, etc. When a comparison is true, the corresponding statement is executed. Then MATLAB skips to the first line after the end. If no match is found among the `case` statements, then MATLAB skips to the `otherwise` statement, if present, and executes `StatementsOtherwise` or else to the `end` statement. Note that the `otherwise` statement is optional.

Let's see an example. Write the following M-script and save it with the name `ArrOrder`.

Listing 4.4

```
% The M script displays a different string on
% the command window.
% according to the integer value of i (to be set before
% the script execution.)
%
% Author: Borgo, Soranzo, Grassi 2012

switch k
    case 1
        disp('First! Good job!');
    case 2
        disp('Second!');
    case 3
        disp('Third, nice result!');
    otherwise
        disp('Next time is going to be better!');
end
```

Now let's test the how the script works by typing the following:

```
>> clear all
>> k = 27
>> ArrOrder
Next time is going to be better!
```

Note that if you do not define the variable `k` (in the command window or inside the M-script) MATLAB returns an error:

```
>> clear all
>> ArrOrder
??? Input argument "k" is undefined.
Error in ==> ArrOrder at 7
switch k
```

Multiple expressions can be handled in a single *case* by enclosing the fact to compare within a cell array. For example substitute the lines 10, 11, 12 and 13 with the following lines:

```
case {2,3}
    disp('You are on the podium, but... not in first position!');
otherwise
    ...
```

If you test the function with `k=2` or `k=3`, the text displayed at the MATLAB prompt is always the same.

```
>> k=2;
>> ArrOrder
You are on the podium, but… not in first position!
>> k=3;
>> ArrOrder
You are on the podium, but… not in first position!
```

For Loops

Often, there is the need to repeat a block of statements a fixed number of times. For example, let us suppose we want to display the first ten responses to a questionnaire, which are contained within an array of numbers. A possible (but inefficient) solution is to repeatedly write the statement we need to run to see the scores. Another (more elegant and efficient) solution is to use a loop structure. The *for* structure is a loop structure that makes it possible to repeat a block of statements a *fixed* number of times. The *for* format is:

```
for counter = list_of_values;
    statements
end
```

The `counter` variable consecutively assumes the value of each column of the list from the first value to the last. The counter variable is often used within the loop as an index to address the content of a vector or as a variable for other calculations. Let's see how the for loop works by writing and executing the following M-script.

Listing 4.5

```
1  % First For Loops example
2  % Author: Borgo, Soranzo, Grassi 2012
3
4  ListOfValues=[1,2,3,4];
5  disp('First Test');
6  for counter = ListOfValues
7      disp(counter);
8  end
9
10 disp('Second Test');
11 for counter = 1:4
12     disp(counter);
13 end
```

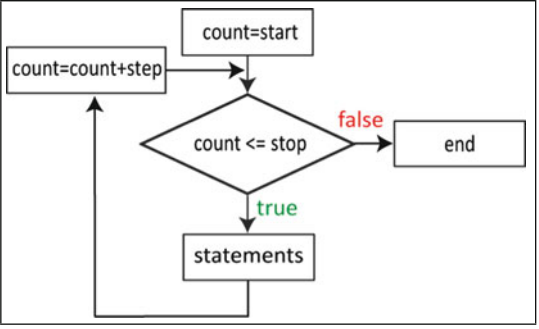
Once you have saved the file (we named it "FirstForTest") and run it, the result is:

```
First Test
     1
     2
     3
     4
Second Test
     1
     2
     3
     4
```

As you can see, the result of lines from 6 to 8 and from 11 to 13 are identical, i.e., the display of the first four integers. This is because the statement `1:4` at line 11 generates a vector equivalent to the `ListOfvalues` vector. Note that the `disp` command is repeated four times, without the necessity of repeatedly writing the statement.

The most common format of the *for* loop is shown below. It may be simpler to understand the *for* loop when it is represented in a flow chart graphical format.

Format	Graphical interpretation

`for count = start:step:stop;`
`  statements`
`end;`
	

The variable `count` is the counter. At the first cycle it is set to the *start* value (the first element of the vector). MATLAB verifies whether the counter `count` is still less than or equal to the *stop* value, thus that the variable count has not yet reached the last element of the vector. If this is true, MATLAB adds to the counter `count` the *step* value (count assumes the value of the next element in the vector) and executes the statements. At the next cycle, MATLAB checks whether the counter `count` is still less than or equal to the *stop* value. If this is false, the next command to be executed is the command after the `end` command. Please note that the counter should not be changed inside the loop.

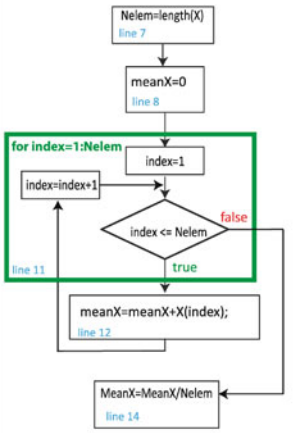
Now suppose you need to obtain the average of the values stored in the vector X. You can write an M-script using a *for* loop as follows:

Listing 4.6

M-script "LoopMean"

```
1   % M-script to calculate the mean
2   % of a vector X. X must exist before
3   % running this M-script
4   %
5   % Author: Borgo, Soranzo, Grassi 2012
6
7   Nelem=length(X);
8   meanX = 0;
9
10  %calculate the X mean
11  for index=1:Nelem
12     meanX=meanX+X(index);
13  end
14  meanX=meanX/Nelem;
15
16  disp(sprintf('the X mean is %2.2f',meanX));
```

Graphical Interpretation



Analysis

Line 7: The number of elements in `X` is obtained.

Line 8: The vector `meanX` is *initialized* to zero. This initialization is necessary because the variable is used as an accumulator to store the partial sum of the elements.

Line 11: Here is the `for` command. First, the variable `index` is set to 1, then it is compared to `Nelem`. Next, on line 12, the partial sum is obtained by summing the value of `X` indexed by `index` and the previous partial sum value contained in the `meanX` variable. Line 12 is repeated till `index` is less than or equal to `Nelem`. The flow chart represented in the figure illustrates what we have just described. Let's test the M-script (named LoopMean) by typing the following:

```
>> X= [3 4 6 2 7 1 9 11 4 7];
>> LoopMean
the X mean is 5.40
```

The *for* structure can be nested into another *for* structure. For example, here we nest two *for* structures to calculate the mean of a 2-dimensional matrix.

Listing 4.7

```
% The M-script to calculate the mean of a 2-dimension X2 matrix.
% X2 must exist before running this M-script
%
% Author: Borgo, Soranzo, Grassi 2012

[Nrows,Ncols]=size(X2)
meanX = 0;

for indr=1:Nrows
   for indc=1:Ncols
       meanX=meanX+X2(indr,indc);
    end
end
Nelem=Nrows*Ncols;
meanX=meanX/Nelem;

disp(sprintf('the X mean is %2.2f',meanX));
```

Note here that the counter `indr` is used to address the matrix rows, while `indc` is used to address the matrix columns. The counter `indc` changes in the inner loop, which means that before increasing the `indr` value, all the columns (= `indc` values) have to be considered.

If we save the script (name it NestedFor) and run it, we get:

```
>> X2=[4 5 6; 4 7 8];
>> NestedFor
Nrows =
      2
Ncols =
      3
the X mean is 5.67
```

You can write the *for* loop in a single command line. Here we show the general form:

```
for index = j:m:k, statements, end
```

Remember: do not forget the commas (or semicolons, which are preferable to commas because they prevent the echoing of output in the command window). If you forget, them you will receive an error message. The word *statements* represents one or more statements, separated by commas or semicolons. Remember to "close" the for loop with the `end` statement.

Be careful when you use the counter to address the cells of a matrix. For example, do not set the counter so that it starts from 0. If you do so, you will encounter the following problem:

```
>> for i=0:10; a(i)=i; end;
??? Subscript indices must either be real positive integers or logicals.
```

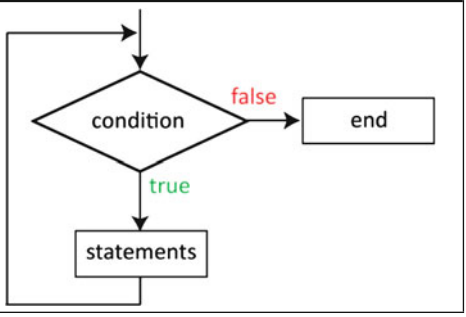
Clearly, the index of a vector cannot be zero. You can, however, run the loop with no error if you modify it slightly, as follows:

```
>> for i=0:10; a(i+1)=i; end;
>> a
a =
    0   1   2   3   4   5   6   7   8   9   10
```

While

Like the for structure, the *while* structure can be used to repeat a number of statements. The difference between the *while* and the *for* structures is that `while` allows one to repeat the statements an *indefinite* number of times until a certain condition is true. The `while` format is as follows:

Format	Graphical interpretation

`while condition`
`  statements`
`end`
	

The `while` construct repeats the `statements` *while* its condition remains true. The condition is tested each time before the *statements* within the loop are repeated. Note that the condition should refer to the statements within the loop otherwise the loop would never end. While loops are complex to deal with, especially when you start programming, because often MATLAB enters in an infinite loop. If this happens, you can press the buttons CTRL+C and force-quit the loop.

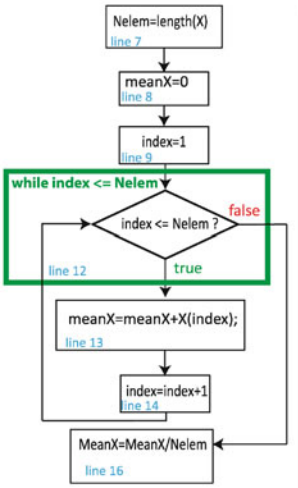
If we use a counter, we can copy the functionality of *for* loop with a *while* loop. Here, for example, we translate the for loop of the `LoopMean` function example into a while loop.

Listing 4.8

M-Script "WhileMean"

```
1   % M-script to calculate the mean
2   % of a vector X using while.
3   % X must exist before running this M-script
4   %
5   % AUTHORS:   MB-AS-MG - 2012
6
7   Nelem=length(X);
8   meanX = 0;
9   index=1;
10
11  %calculate the X mean
12  while index <= Nelem;      % while loop.
13     meanX = meanX + X(index);
14     index = index + 1;  % counter updating
15  end
16  meanX = meanX / Nelem;
17
18  disp(sprintf('the X mean is %2.2f',meanX));
```

Graphical Interpretation



Analysis

Line 9: We define the variable `index` used as a counter and initialize it to 1.

Line 12: Here is the `while` command; the condition `index <= Nelem` is evaluated. If it is true, then (line13) the value of `X` indexed by `index` is added to the previous partial sum in the `meanX` variable.

Line 14: the counter is increased. Note here the difference between the *while* and *for* loops. In the *for* loop, the counter is "automatically" increased and "compared" at each cycle. In contrast, in the while loop, you have to increase the counter inside the loop. Then MATLAB goes to line 13 and evaluates whether `index<= Nelem`, etc. The flow chart gives you a better picture of what is happening here.

At first glance, it may seem that the *while* command is not as useful as the *for* structure. For example, the majority of experiments in psychology use a fixed number of trials. In other words, before the subject begins the experiment, we know exactly how many trials the subject is going to run. In the MATLAB language this suggests that one implement a for loop (see also the brick section for an extensive example): *for* the desired number of trials we present the stimulus to the subject and collect the subject's answer. The counter, in this case, is the trial number, which is increased of one unit every time the subject completes one trial. When the current trial number exceeds the total number of trials, we exit the *for* loop.

However, there are experiments in which we do not know in advance how many trials the subject is going to run. The adaptive procedures used in psychophysics are a typical example of experiments that use a variable number of trials (Gescheider 2003). The simplest example is the method of limits (Fechner 1889, but see also Boring 1961). Let's suppose we use this method to estimate the subject's absolute threshold for the intensity of a 1-kHz pure tone. We may proceed as follows. We start the experiment by presenting one tone of a given intensity to the subject. The subject has to respond whether s/he can hear the stimulus (by pressing 1) or not (by pressing any other number, e.g., 2). Every time the subject's answer is positive, we halve the intensity of the tone. At a certain trial *n*, the subject's answer will be negative because s/he is not able to hear the tone. At this point we have just reached the subject's absolute threshold for the tone's intensity.

We can implement the example just described with a while loop. In detail, while the answer of the subject is 1 (equivalently, until the answer is not 1), we play the tone to the subject and collect the answer. The answer is evaluated in the logical statement that keeps the while loop running. While the answer is TRUE (equivalently, until the answer if FALSE) we repeat the while loop. At the first negative answer we quit the loop.

Listing 4.9

```
1  sf = 44100; % Sampling frequency
2  d = .25;    % Duration in seconds
3  f = 1000;   % frequency in Hertz
4  I = 1;      % starting intensity
5  % tone synthesis
6  t = linspace(0, d, sf*d);
7  tone = sin(2*pi*f*t);
8  % play the sound the first time and get the answer
9  sound(tone, sf);
10 answer = input('Can you hear the tone? ');
11 % repeat the stimulus until the answer changes from "yes" to "no"
12 while answer == 1
13     I = I/2;
14     sound(tone * I, sf);
15     answer = input('Can you hear the tone? ');
16 end
17 % return the threshold
18 fprintf('The threshold is equal to %2.1f dB Fs\n', 20*log10(I));
```

Analysis

The sound functions used in the script are described in detail in the chapter dedicated to sound. Let's see the rest of the script:

Lines 1–4: The variables needed to synthesize the sound are created and set.

Lines 5–7: The tone is synthesized.

Line 9–10: We play the tone for the first time to the subject and store the answer in the variable "answer".

Line 12–16: The while loop. We evaluate the answer of the subject. While the answer of the subject is positive (i.e., equal to 1) we halve the intensity of the tone (line 14) and play the tone at the halved intensity (line 15). Then we repeat the question to the subject.

Lines 17–18: the subject has pressed a button different from "1" and the last intensity value is used to calculate the subject's threshold.

Break

The `break` command is used to terminate in advance the execution of a `while` loop or that of a `for` loop. In nested loops, `break` terminates the innermost loop only. If you want to program with a good programming style you should avoid the use of the break command. In fact, it is important to know that the break command can be always substituted by for or else statements.

Here we present a variation of the `WhileMean` M-script without the comments at the beginning.

Listing 4.10

```
1  Nelem=length(X);
2  meanX = 0;
3  index=1;
4
5  %calculate the X mean
6  while 1     % Never ending while loop.
7     meanX = meanX + X(index);
8     index = index + 1;  % counter updating
9     if index <= Nelem
10        break
11    end
12 end
13 meanX = meanX / Nelem;
14
15 disp(sprintf('the X mean is %2.2f',meanX));
```

Note that the condition of the while loop (i.e., 1) is always true, and therefore the loop above is a never-ending loop. This happens because MATLAB interprets every value different from zero as a true logical value. The rows 9, 10, and 11 are there exactly with the aim of terminating the loop. You can try to insert the break command in Listing 4.10 to obtain the same result.

Try–Catch

The try–catch statement is not used to control elements; rather it tries to execute a statement or statements and catch any errors that may occur. The try–catch statement takes the following form:

```
try
     statements
catch
     statements
end
```

The try–catch statement is useful when you are running scripts that can return anomalous responses, such as importing data from a file or using the psychtoolbox commands (see later chapters). When you use the psychtoolbox, it can sometimes happen that your computer seems to crash. In such cases, the CTRL+C command is insufficient to restore the command window. If you want to avoid such behavior, you should write the psychtoolbox commands between `try` and `catch` commands.

Here is an example of how the try–catch statement works. Save the following M-script in the file named trycatchExample and run it:

Listing 4.11

```
1 try
2    for i=0:10; a(i)=i; end;
3 catch
4    disp('Something strange happened!')
5 end
```

```
>> trycatchExample
Something strange happened!
```

This example shows clearly the use of the try/catch statement. The `for` loop generates an error because the `i` counter is used as index for the vector a. But `i` starts from 0, and therefore MATLAB cannot read the first datum of a. Because the error is generated within the try portion of the script, the program continues and goes to the catch part, where it returns the sentence "Something strange happened!." If you want to see the error returned by MATLAB, you have to type the command `lasterror` at the prompt.

Loops Versus Matrices and If Versus Logicals

Once people have learned how to use the *for*, the *while* and the *if* structures, they tend to forget the power of using matrices in MATLAB. Let's take a look at the

following two statements. These statements do the same thing: they look into a matrix `A` and return the number of elements whose value is included between 0.2 and 0.3.

Matrices & Logicals

```
1  NumEl=length(a(a>0.2 & a<0.3));
```

Loops & If

```
1  NumEl=0;
2  for i=1:size(A,1);
3      for j=1:size(A,2)
4          if A(i,j)>0.2 & A(i,j)<0.3;
5              NumEl = NumEl +1;
6          end
7      end
8  end
```

As you can see, if you "think in the matrix way," the program you have to write is much shorter. Moreover, the example on the left runs faster than the example on the right: MATLAB is optimized to work with matrices. Therefore, we recommend the use of matrices and logicals whenever possible.

Functions

Scripts work with variables that are defined within the same script or in the command window. However, sometimes we want scripts to receive data as input and return results as output. Such scripts are called *functions*. We have already seen several built-in MATLAB functions, such as sin, sum, and length. However, MATLAB makes it possible to create your own functions.

If we want to write a function, the script must start with the reserved word `function`, and the M-file name has to match the function name. The difference between M-scripts and function scripts is that **functions communicate with MATLAB through *input* and *output* arguments**. We will stress this concept further throughout the chapter.

Before highlighting the *function* properties, type the following example as a new script. Remember to save the script with the same name you use to call the function. In this case, the name must be "statistic.m"

Listing 4.12

```
1  function [mea, vari] = statistic( x )
2
3  % [mea, vari] = statistic( x )
4  %
5  % The function returns the mean and the standard deviation
6  % of the input vector.
7  %
8  % INPUT:    x vector of numbers. Do not use with matrices.
9  % OUTPUT:   mea = is the mean of the input vector x
10 %           vari = is the variance of vector x
11 % Author:   Borgo, Soranzo, Grassi 2012
12
13 Nelem = length(x);
14 mea=sum(x)/Nelem;
15 vari=sum(x.^2)/Nelem;
```

The aim of our statistic function is to calculate the mean and the variance of a vector passed as argument to the function. Note that we also benefited from the MATLAB built-in function `sum(x)`, which calculates the sum of the elements of vector `x`, and the MATLAB built-in function `length` (`x`), which returns the number of elements of the vector. When we save this script, MATLAB automatically suggests a filename that is identical to the function name; in our case the filename is `statistic`. Let us test the function in the command window by typing the following:

```
>> randnum=rand(100,1);
>> [m,v]= statistic(randnum)
m =
   0.4753
v =
   0.3103
```

Finally, type `help statistic` at the MATLAB prompt. If you do so, you will see the comments we added at the beginning of the script:

```
>> help statistic
  [mea, vari] = statistic( x )

  The function returns the mean and the standard deviation
  of the input vector.

  INPUT:     x vector of numbers. Do not use with matrices.
  OUTPUT:    mea = is the mean of the input vector x
             var = is the variance of vector x
  Author:    Borgo, Soranzo, Grassi 2012
```

Let's see the general form of a function:

When you write a function, the first line must start with the keyword function. The general form of a function is the following:

```
function [ out1, out2, … ] = fun_name( inp1, inp2, … )
% comments to be displayed go here
…
out1 = … ;
…
out2= …;
```

- *Keyword* `function`: The function must start with the keyword `function`. As you have seen, the `statistic` function starts with the keyword `function`.
- *Output argument*: For more than one output argument, the output arguments must be separated by commas and enclosed in square brackets. However, if the function returns only one output variable, this output can be written without square brackets.
- *The function name*: usually the function line name should be identical to the filename (in the previous example function line name `statistic` was saved in

the file named "statistic.m"). If the filename and the function definition line name are different, the latter name is ignored.
- *Input argument*: The input variables are written within parentheses. For more than one input argument, commas must separate the input arguments.

Input and output variables can be of any kind: numerical, logical, text, structures, cells. However, **the input and output variables are "dummies" and serve to define a way for the function to communicate with the workspace**. What happens is that the workspace input arguments are copied into the dummy input arguments within the function when the function is invoked. For example, the variables `mea`, `vari`, and `x` exist only within the function and not outside of it (i.e., in the workspace). This can be seen simply by typing the command `who` in the command window:

```
>> who
Your variables are:
m         randnum v
```

This does not happen for input and output variables only, but for all the variables that are defined within the function, such as, for example, `Nelem`. In fact, if we type `Nelem` at the MATLAB prompt, we get the following error:

```
>> Nelem
??? Undefined function or variable 'Nelem'.
```

In the lines after the first, there are some comments. These lines will be displayed if you type `help fun_name` at the MATLAB prompt (e.g., `help statistic` in our case). It is useful to comment your function so that you know exactly what the function does. Consider also writing comments about the input and the output variables, about who wrote the script and when it was made. After the comments, there are some command lines calculating the mean and the variance of the input vector.

All the output variables have to assume a value. If this does not happen, MATLAB informs you that you have not assigned a value to (at least) one variable. Let's test this by commenting the line `vari=sum(x.^2)/Nelem;` i.e., insert the `%` symbol at the beginning of the line. Save the function script and call the function from the command window:

```
>> [m,v]= statistic(randnum)
Error in ==> statistic at 13
Nelem = length(x);

??? Output argument "vari" (and maybe others) not assigned during call to
"/Users/Script/statistic.m (statistic)".
```

MATLAB informs you that you have forgotten to assign a value to the output argument. Note that MATLAB returns another error at line 13. However, this error does not exist. It is an artifact of the fact that MATLAB stopped the execution of the function; it gives you the reference of the first useful script line.

Scope of Variables

As we have written above, when a function is called from the workspace or from another function, the variables defined in the function are created and live only within the function. Usually in computer programming, the term *scope* is used (but not limited) to define the visibility or accessibility of variables from different parts of the program. The variables within a function are called *local variables*. The *scope* of Local variables is that they exist only within the function where they are defined. However, there are other types of variables, for example global variables and persistent variables, and these work in a different way from local variables:

1. *Global variables*: They are ubiquitous, or better, they exist everywhere. When you define them in the workspace, they exist not only in the workspace but also inside functions. The *scope* of the variable is to be globally accessible. It is a good practice to use capital letters for global variable names so to identify them easily. Global variables are generated through the `global` command. For example:

   ```
   >> global DEUSVAR
   ```

2. *Persistent variables*: Once they are created, they "live in the space" where they have been created. While local variables normally stop existing when a function returns its value, persistent variables remain in existence between function calls, keeping the value they had after the last manipulation. The following example should clarify this.

Listing 4.13

```
1 function TestPersistent
2 % Function to test persistent variables
3
4 persistent RemCount
5 if isempty(RemCount)
6    RemCount = 0
7 end
8 RemCount = RemCount + 1
```

Analysis

Line 4: we define the variable `RemCount` as a persistent variable

Line 5: The `isempty()` function returns true if the variable within parentheses is empty. For example, test the function with empty text and with text that is not empty:

```
>> txt1 = '';
>> txt2 = 'hallo';
>> isempty(txt1)
```

```
ans =
     1
>> isempty(txt2)
ans =
     0
```

Once you save the function as TestPersistence.m, run it repeatedly from the command window; you should see something like this:

```
>> TestPersistent
RemCount =
     1
>> TestPersistent
RemCount =
     2
>> TestPersistent
RemCount =
     3
```

The first time you run the `TestPersistence` function, the variable `RemCount` does not exist. The command persistence RemCount creates a new empty variable. The result of `isempty` is true, so line 6 is executed, `RemCount=0`. Next line 7 is executed, and `RemCount` becomes `1`. By calling again `TestPersistence`, 1 is added to the last value, so `RemCount` becomes 2; and so on. This happens because in contrast to local variables, `RemCount` exists when the function `TestPersistent` is called, and it also remembers its previously assigned values.

Generally, the most-used variables are those with local and global scope.

Change the Number of Inputs and Outputs

Sometimes we need to write functions that can take a different number of input arguments, such as the `plot` function, which we have described in the previous chapter. It is possible to pass as inputs a variable number of arguments. If we need to write a function with such characteristics, we need to use the `varargin` and `nargin` commands, which stand for, respectively, **var**iable **arg**uments **in** and **n**umber of **arg**uments **in**. But if necessary, we can also write a function that returns a variable number of outputs. In this case we have to use the `varargout` and `nargout` commands.

Let's see the use of `nargin` and `varargin` and `nargout` and `varargout` by extending the `statistic` function outlined in Listing 4.12. That function returned the mean and the variance of an input vector. Now we want to add the optional possibility to exclude the outliers from the computation and to return the variance as an option. Therefore, we implement the `statTwo` function, which takes an "optional" argument, a logical vector, indicating the values that have to be excluded from the computation because they are outliers. The standard output is the mean of the

function. However, the variance can be returned optionally. This function can be written as follows:

Listing 4.14

```
function [mea, varargout] = statTwo( x,varargin )

% The function returns the mean and the standard deviation
% on the element of x eventually selected by i.
%
% INPUT:    x vector of numbers. Do not use with matrices.
%           i vector of logicals.
% OUTPUT:   mea = is the mean
%           if required by the number of outputs, the
%           second element is a cell variable containing the
%           variance.
% Author:   Borgo, Soranzo, Grassi 2012

disp(sprintf('Number of input: %d',nargin));
disp(sprintf('Number of output: %d',nargout));
i=logical(ones(size(x)));
if nargin == 2
   i = logical(varargin{1});
end
Nelem = sum(i);
mea=sum(x(i))/Nelem;
if nargout == 2
    varargout{1}=sum(x(i).^2)/Nelem;
end
```

Save the function `statTwo` and run it from the command window in this way:

```
>> aa=[2 5 4 7 6];
>> bb=[1 0 0 0 1];
>> cc=[1 2 5 1 1];
>> m = StatTwo(aa)
Number of input: 1
Number of output: 1
m =
   4.8000
```

As you can see, MATLAB returns the number of input arguments (in this case there is only the vector `aa`) and the number of output arguments (in this case only one). The result of the function is stored in the variable `m`.

```
>> m = StatTwo(aa,bb)
Number of input: 2
Number of output: 1
m =
   4
```

In this second case, there are two inputs (the vectors `aa` and `bb`) and again one output, now the mean is calculated considering only the values 2 and 6. Let's see what happens if we change again the number of inputs and outputs:

```
>> [m,v] = StatTwo(aa,bb)
Number of input: 2
Number of output: 2
m =
   4
v =
   20
>>
>> [m,v] = StatTwo(aa)
Number of input: 1
Number of output: 2
m =
   4.8000
v =
   26
```

Let's see what happens if we pass either multiple inputs or multiple outputs:

```
>> m=StatTwo(aa,bb,cc)
Number of input: 3
Number of output: 1
m =
   4.8000
>>
>> [m,v,boh]=StatTwo(aa,bb)
Number of input: 2
Number of output: 3
??? Error using ==> statTwo
Too many output arguments.

Error in ==> statTwo
```

If we pass multiple inputs, the function works because the vector `c` is ignored. In contrast, if we pass multiple outputs, the function assigns only two variables at the output. MATLAB does not know what values to store in the vector `boh`, so it returns an error. We now describe in detail the `statTwo` function.

Analysis

Line 1: In the first line there are two input arguments; one is the variable `x`, and the other is `varargin`. `varargin` is a cell matrix, which can contain many values,

and therefore it can also contain any number of input variables. `varargin` is a **cell matrix variable** that collects all the inputs' indices with the same input order. In the previous example, when we have called the function `m=statTwo(aa,bb,cc)`, a copy of `a` was put in the variable `x`, a copy of `b` was put in the cell `varargin{1}`, and a copy of `c` was put in the cell `varargin{2}`.

`varargin` must appear at the end of the argument lists. In the same way, the outputs are the variable `mea`, and `varargout`. `varargout` is a cell matrix that collects all the outputs.

Lines 13–14: The number of input and output arguments is displayed. Here, we use the commands `nargin` and `nargout`, which return, respectively, the numbers of inputs and outputs of the called function.

Line 15: the `i` vector is initialized in case there is only one input argument. The vector `i` is used as a logical, so that all the values in the vector `x` are considered in evaluating the mean and the variance.

Line 16: Here is used again the command `nargin`. In brief, in this line we evaluate how many input arguments there are. If and only if the number of input arguments is 2, line 17 is executed. In fact, when the number of input arguments differs from 2, line 17 is not executed. If you want always to use the second input argument of the function (when present), you should change line 16 as follows:

```
if nargin >= 2
```

Line 17: Use the second input argument as a logical vector.

If you need for a function to return an indefinite number of output variables, you can use `nargout` and `varargout`. This is the case in the example.

Line 19: verify how many outputs are required. If there are two output values, the variance is computed on line 20 and the result is stored in the first element of the cell matrix `varargout`.

You can use the functions `nargchk` and `nargoutchk` inside an M-file function to check that the desired number of input and output arguments is specified in the call of that function.

For further information, please refer to MATLAB help.

More on Data Import/Export: Script Examples

The current section shows how to write a script (M-script or function) that retrieves data from an unknown file format (e.g., data saved in ASCII form by an old EEG machine, or collections of data, organized in different files that you want to rearrange in a unique dataset). Moreover, the current section teaches you how to save a file in a format that is readable by some other software.

Before we begin to write some examples, we introduce a few new commands especially dedicated to accessing the content of a file. These commands are presented in the following table. Usage examples are given within the next example scripts.

MATLAB function	Description
`fid=fopen(filename)` `fid=fopen(filename,perm)`	`fopen(filename)` open a file specified by the string `filename`. `perm` specifies how the file is opened, according to the following characters: • 'r' read • 'w' write (create the file if necessary) • 'a' append (create the file if necessary) The function returns a value called *file identifier*. Usually the variable `fid` is used to collect it. This variable must be used by other input/output commands If the returned value is equal to −1, the file cannot be opened
`fclose(fid)`	It closes the file associated by the file identifier `fid`. The function `returns 0 if successful or -1 if not`
`feof`	Return 1 when the end-of-file of the file identified by `fid` is reached, and 0 otherwise
`fprintf(fid, format, A,...)`	Write formatted data into a file identified by `fid`. The `format` string is created in the same way as the `sprintf` format. `A` is the variable we write into the file according to the format specified in `format` The function returns the count of the number of bytes written
`fscanf(fid, format)`	This function reads data from the file specified by `fid` and converts it according to the specified `format` string. Moreover, the file content is returned in a matrix. The `format` string has the same property of the format string for functions `fprintf` and `sprintf`
`fgetl(fid)`	This function returns a string with the content of the next line of the file associated to the file identifier `fid` When the function encounters the end-of-file indicator it returns `-1`
`fgets`	Same as `fgetl`, but it contains the line terminators
`textscan(fid, 'format')`	This function reads data from an open text file identified by the file identifier `fid` and returns the data into a cell. The format input is a string enclosed in single quotes These conversion specifiers determine the type of each cell in the output cell array. The number of specifiers determines the number of cells in the cell array. Some of the specifiers are equal to the `sprintf` formats Note that `testscan` stops when the format is different from the specified one

Here we write an example that displays the content of a file in the command window.

Listing 4.15

```
function DisplayFile(filename)

% function DisplayFile(filename)
%
% It reads a text file and displays it on the command window
%
% INPUT:    filename = a string containing the file name
% Author: Borgo, Soranzo, Grassi 2012

try
    fid=fopen(filename);        % open file returning the file id.
    if fid == -1                % file doesn't open correctly
        disp(sprintf('Unable to open file ''%s'' ',filename));
    else                        % in this case file opens correctly
        while ~feof(fid)        % Start loop. End if eof is reached
            tline = fgetl(fid); % read next line of text from file
            disp(tline);        % display text line on command window
        end
        fclose(fid);            % close the file
    end
catch
    fclose(fid);
    disp('Something wrong happened');
end
```

Note in the example the use of the try–catch statement: the catch is useful for closing the file in case something goes wrong. Let's suppose an error occurs before the script closes the file. In this case, you can have no more access to the file because the operating system acts as though someone were still using it.

We test the function by reading the file howmany2.m and then by reading a nonexistent file.

```
>> DisplayFile('howmany2.m')
function[b]= howmany2(A)
b=0;
for i=1:size(A,1);
    for j=1:size(A,2)
        if A(i,j)>0.2 & A(i,j)<0.3;
           b=b+1;
        end
    end
end

>> DisplayFile('LotsOfMoney')
Unable to open file 'LotsOfMoney'
```

The previous function explains how to obtain a string from each line. With such a string it is then possible to do all the processing you need. However, let's suppose you want to obtain the data in a matrix, ready for computation. This can be done, but it is important to know how the content of the file is formatted. For this reason we show here a function that saves the data in a specific format, and then we show a function that reads these data.

Let's suppose you run an experiment that investigates iconic memory (Sperling 1960). In the experiment we present to the subject an n by m matrix on the screen. During the trials, certain cells of the matrix are filled with an arbitrary symbol (e.g., a cross). The matrix stays on the screen for just a short time, and the subject has to tell, by means of a mouse click, the locations of the crosses presented on the screen. At each mouse click we save the mouse position and the time interval (in milliseconds) between the moment the matrix disappeared and the mouse click. Let's suppose we store our data in a structure named `STIM`, with the following fields: `nStim`, `time`, and `Mpos`. We can save the data in a readable manner using the following function.

Listing 4.16

```
function SaveStrangeFormat(filename, STIM)

% function SaveStrangeFormat(filename, STIM)
%
% reads text file and display it to the console
%
% INPUT :   filename = a string containing the file name
%       :   STIM = a vector of struct cointaining the following fields
%       :    nStim = the stimulus number
%       :    MPos = a Matrix 2 x Npos of mouse clicks
%       :    time = a vector 1 x Npos of time for each click
% Author: Borgo, Soranzo, Grassi 2012

try
    if ~exist(filename)     % control if the file doesn't exist
        fid=fopen(filename,'w');    % open file to write
        if fid == -1                % it can be not possible
            disp(sprintf('Unable to write file ''%s'' ',filename));
        else                        % in this case file opens correctly
            for i=1:length(STIM)    % write data loop
                fprintf(fid,'Stimolous number: %d \n',STIM(i).nStim);
                fprintf(fid,'\t Time (ms) \t Mouse Position\n');
                for j=1:length(STIM(i).time)
                    fprintf(fid,'\t %4d \t \t x=%4.0f y=%4.0f \n',...
                   STIM(i).time(j),STIM(i).MPos(j,1),STIM(i).MPos(j,2));
                end
            end
            fclose(fid);            % close the file
            disp('File saved correctly!');
        end
    else
        disp('The file already exists, please change file name');
    end
catch
    fclose(fid);
    disp('Something wrong happened');
end
```

The function has the same structure of the previous one. The difference is in the use of "open" at line 16, where it is specified that the file is open to write something into it (we added the "w" *perm* character). Pay attention as well to the multiple use of the `fprintf` function to write the data into a file in a formatted way.

We test this function in two steps: we first create the data, then we write the data into a file. We then repeatedly look into the file using the function `DisplayFile` that we created earlier. We use a custom file extension.

```
>> STIM.nStim=1; STIM.time=[123 576 1034];
>> STIM.MPos=[23, 45; 345,15; 256,176];
>> SaveStrangeFormat('TestSave.myf', STIM);
File saved correctly!
>> DisplayFile('TestSave.myf')
Stimolous number: 1
       Time (ms)      Mouse Position
        123           x=  23 y=  45
        576           x= 345 y=  15
       1034           x= 256 y= 176
```

If you copied the example with no errors, everything should work properly. Here, we want to go back to the function and outline the `fprintf` format at line 24. Type:

```
>> STIM
STIM =
     nStim: 1
      time: [123 576 1034]
      MPos: [3x2 double]
```

Note that MPos is a *double* matrix, which is why in the `fprintf` format we uses the `%f` identifier.

Now let's make a further step and import the data from an odd file format, like the previous one. We want to highlight that it is necessary to know exactly how data are formatted and their meaning to import them correctly. Let's take a look at the following function.

Listing 4.17

```
1  function [STIM]=ImportStrangeFormat(filename)
2
3  % function ImportStrangeFormat(filename)
4  %
5  % it reads a text file and displays it on screen
6  %
7  % INPUT :   filename = a string containing the file name
8  % OUTPUT:   STIM = is a vector of struct cointaining the following fields
9  %       :    nStim = stimulus number
10 %       :    MPos = Matrix 2 x Npos of mouse clicks
11 %       :    time = vector 1 x Npos of time for each click
12 % Author: Borgo, Soranzo, Grassi 2012
13
14 try
15     fid=fopen(filename);          % open file
16     if fid == -1                  % it can be not possible
17         disp(sprintf('Unable to write file ''%s'' ',filename));
18     else                          % in this case file opens correctly
19         NStmp=0;
20         while ~feof(fid)          % Start loop. End if eof is reached
21             NStmp=NStmp+1;        % Update the number of NStmp
22             sn=textscan(fid,'Stimulus number: %d'); % read the line
23             STIM(NStmp).nStim=sn{1};     % Save the stimulous number
24             fgetl(fid);           % jump a line
25             STIMtemp=textscan(fid,'\t %4d \t \t x=%4.0f y=%4.0f');
26             STIM(NStmp).time=STIMtemp{1};       % Create the struct vect
27             STIM(NStmp).MPos=[STIMtemp{2}, STIMtemp{3}];
28         end
29         fclose(fid);              % close the file
30     end
31 catch
32     fclose(fid);
33     disp('Something wrong happened');
34 end
```

Analysis

Line 22: The `textscan` function is used. `textscan` skips the string 'Stimulus number:' and reads the number. It converts the number as described by the specifier (`%d`) and then puts the number in the cell `sn`. On line 23, the value is stored in the structure.

Line 24: We skip the file line containing the string ' `Time (ms) Mouse Position`'.

Line 25: We read the following numbers according to the specified format. Note that since `textscan` finds the same format, it reads multiple lines. The result is put in a cell.

Line 26 and 27: The cell values are put into the structure.

Line 28: The counter `NStmp` is updated.

Now test the function by typing the following lines:

```
>> STIM1= ImportStrangeFormat('TestSave.myf');
STIM1 =
    nStim: 1
     time: [3x1 int32]
     MPos: [3x2 double]
>> STIM.MPos-STIM1.MPos
ans =
     0      0
     0      0
     0      0
```

As you can see, the data are loaded correctly (remember that `STIM` was defined to test the `SaveStrangeFormat` function).

Guidelines for a Good Programming Style

Here we report a few guidelines for a good programming style. When you begin to write a new program, first you have to address the problem and find out a way to solve it, i.e., the algorithm. This means that you have to decide on the basic tools you need to solve the problem. Then you have to follow a design process that decomposes the problem into subordinate problems. This helps you to spot the recursive tasks of your problem and the functions that you need to create. Finally, you have to translate or convert the algorithm into a MATLAB script and test it. This type of approach is called *top-down*. Your ability to use all of MATLAB's potential is limited by your experience. The more you increase your knowledge, the more you will be creative and efficient in solving problems.

In any case, it is important to follow at least some general guidelines that can be learned from expert programmers. Sometimes you can be impatient to get on with your job. However, just a little attention to your programming style can help you later in your work. The goal of these guidelines is to help you in producing code that is clearer and easier to understand and update. If the code can be easily read and understood by yourself and by other users, you can probably quickly modify it and control it to spot where any errors are. These recommendations apply to any programming language. Here, they are simply adapted to the development and writing of MATLAB scripts.

Writing Code

As we wrote previously, the best way to write a long and complex program is to assemble it from well-designed small programs (usually functions). The idea is an IKEA-like programming style: a lot of small parts, each specialized for a specific task, that can be assembled modularly.

So the basic points for modularity are:

- Small and well-designed functions are more likely to be reused by other applications and programs.
- Make the interaction clear: define well the input and output arguments and their format. Structures can be used to avoid a long list of input or output arguments.
- Use a defensive programming approach. This means that the input variables should be controlled. Check whether the input variable is of a type that the function is expecting. For example, if you are writing a function that works with numbers, check whether the input variable is a number, a logical, or a string. This can be done using the conditional structures (i.e., if–else, switch–case) at the beginning of the script. For example:

```
function [outvar]=ControlInput(x,filename)

if ~isnumeric(x)
   return;  % if x is not a number exit from the function
   disp('No numeric input was given') % communicate the error to the user
end
if isempty(filename)
    filename = 'Test' % If the filename is forgotten give a default
                      % provide with a default name
end
```

- Insert a default condition if necessary.
- Communicate the errors to the user.

A good visual appearance of your code also helps you to focus on the structure of your script. Therefore:

- Use indentation. Indentation helps you to find where the loops and the conditionals begin and end. MATLAB has a built-in smart indenting tool. Just select the code you are writing, right-click the mouse button, and select *smart indent*.
- Comment your script. Write a comment at the beginning of the script that tells you what the script does. In addition, comment the crucial points of the program. In any case, pay attention to the following:
  - Comment while you are writing your code. Comments that are added later are often confusing. Commenting while you are programming helps you to organize your algorithm/code.
  - Avoid useless or unnecessary comments like:

```
num = 2; % set num equal to 2
```

- Variables often have a meaning. Therefore:
  - Use meaningful variable names. For example, use the names in use in your research field.
  - Use long variable names if necessary. The variable names can be as long as you desire. In this book we use the capital letters to separate parts of a compound variable name: `responseTime`. The same result can be obtained with the underscore (e.g., `response_time`).

Debug

Errors, like death and taxes, are certain: you will surely produce them. The process of detecting and correcting errors is called *debugging*. MATLAB has an efficient tool for debugging your script. Before explaining this tool, let's have a look at the most frequent errors in MATLAB. They are mainly of three types:

- Typos (e.g., sim(t) rather than sin(t));
- Syntax errors in function calls (e.g., wrong number of parameters)
- Algorithmic errors.

The most difficult errors to spot and fix are the algorithmic ones. Basically, if you make an algorithmic error, you obtain unexpected results when you run your script. Such a type of error is also known as a "run-time error." They are difficult to detect because the function's local workspace is lost when the error forces the return to the MATLAB workspace. However, the MATLAB debugging tool overcomes this problem.

The MATLAB debugger can be activated in the MATLAB editor by putting one (or more) breakpoints in the script. When you run your script, MATLAB stops temporarily at the line where the breakpoint was positioned. Once MATLAB stops, the prompt `k>>` appears in the workspace. Now you can see the values of all the variables of your program, type commands, etc. Then you can resume the execution of your script until the next breakpoint.

Setting breakpoints is simple. Just go to the left of the target line and click on – (just to the right of the number line). Here we show how the debugger works using the function `statTwo`.

Action and description	Graphical visualization

Set one breakpoint at line 20 of the script statTwo. A red circle appears to the right of the line in the MATLAB editor	In the editor:
	`19 –  Nelem = sum(i);`
	`20 ●  mea=sum(x(i))/Nelem;`
	`21 –  vari=sum(x(i).^2)/Nelem;`

Test the function	In the command window type: `>> statTwo(rand(1,20))`

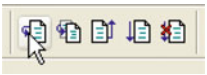
MATLAB runs the function and stops at line 20
At line 20 a green arrow appears, indicating that MATLAB has temporary stopped at line 20. In the command window appears the number of the line followed by the commands on that line
The k>> prompt waits for your (possible) input
	In the editor:
	`19 –  Nelem = sum(i);`
	`20 ●➪ mea=sum(x(i))/Nelem;`
	`21 –  vari=sum(x(i).^2)/Nelem;`
	In the command window:
	`>> statTwo(rand(1,20))`
	`20 mea=sum(x(i))/Nelem;`
	`K>>`

(continued)

(continued)

Action and description	Graphical visualization

You can show the values of the variables used in the function. Line 19 has already been computed by MATLAB, so the variable `Nelem` exists, while the variable `mea` does not exist yet	In the command window:
	`K>> Nelem`
	`Nelem =`
	`20`
	`K>> mea`
	`??? Undefined function or variable 'mea'.`

Now, you can run your function line by line. Just click on the step by step button	In the Editor: 

The debugger goes to the next command line. The mea variable is created and calculated	In the Editor:
	`19 – Nelem = sum(i);`
	`20 ● mea=sum(x(i))/Nelem;`
	`21 – ➪vari=sum(x(i).^2)/Nelem;`
	In the command window
	`K>> mea`
	`mea =`
	`0.4972`

You can continue in step-by-step mode, or run the function or script till the next breakpoint. If there are no more breakpoints, the script is executed till the end	In the editor: 
To clear the breakpoints click on the red circles or on the clear breakpoints button	In the editor: 

MATLAB also provides other tools to help you in writing and managing your M-files. These tools are *M-Lint Code Check* and *Profiler* Reports. They are briefly described in the following paragraphs:

- The M-Lint Code Check Report displays a message for each line of an M-file and determines whether the program can be improved. For example, a frequent M-Lint message is that a variable is defined but never used in the M-file. To activate this tool just select TOOLS and then M-LINT report in the MATLAB toolbar. For further information refer to the MATLAB online help.
- The Profiler helps you to improve the performance of your M-files. When you run a MATLAB statement or an M-file, the Profiler produces a report about the time spent by each function and step of your code. To activate the Profiler just type `profile on` at the prompt. Then you have to type `profile viewer` to stop the Profiler and display the results in the Profiler window. Try to type the following statements:

```
>> A = rand(10);
>> profile on;
>> howmany2(A);
>> profile viewer;
```

A window opens and displays the time used to execute the function howmany2. If the M-script calls more than one function, a list of all these functions also appears. You can look into each function by clicking on the function name. For further information refer to MATLAB help.

Summary

- A script is a list of commands. Use the MATLAB editor to write, edit, and save your scripts. The extension of a MATLAB script is *.m*
- Comments are useful for understanding what a script does. Comments are preceded by the `%` character.
- An M-script can be run by typing its filename at the MATLAB prompt.
- A script starting with the keyword `function` is a function. A function can have input and output arguments. Input and output arguments are the way the function communicates with the variables in the workspace.
- Comments after the first line of a function are displayed in the command window when the *help* of the function is called for.
- The variables used within a function are *local*: they exist only within the function. In contrast, a global variable can be used everywhere.
- The command `nargin` indicates how many input arguments are used in a particular function call, while `varargin` is a cell matrix variable that collects all the inputs.
- `if-else` executes different groups of statements according to whether a condition is true or false.
- `switch` allows the script to make choices between different cases.
- `try-catch` attempts to execute a block of statements and catch errors.
- The for loop repeats a set of commands a fixed number of times. The for form is:

```
for start:step:stop
      Statements
end
```

- The while loop repeats a set of commands an undefined number of times as long as the specified condition is satisfied. The while form is:

```
while condition
      Statements
end
```

- Loops can be nested. The `break` command can be used to quit the (innermost) loop.
- Use logicals and matrix operations whenever possible.

- It is possible to import/export data in specific formats using commands dedicated to file management: `fopen, fclose, fprintf, textread.`
- Use your own programming style, but remember to write your code as clearly as possible. Clear scripts are more maintainable. Prefer many short functions to a few large ones (i.e., use modular programming).
- Use the MATLAB debugger, profiler, and M-Lint tools to support your programming and to find errors.

Exercises

1. Write an M-script that executes the following for loops:

Question	Solution

1.1. Write a loop that generates a column vector A with ten random numbers. Then create a 10×4 matrix having A as its first column, with the remaining columns the product of the first column and the column number	`for ind = 1:10`
	`  A(ind)=rand;`
	`end`
	`for ind2= 1:4`
	`  B(1:10,ind2) = A*ind2;`
	`end`

1.2. Write the vector
`color=[2,1,3,0,1,3,1,0,2]`
Write a loop to display in the command window the corresponding color name as follows: 0 = yellow; 1 = red; 2 = green; 3 = blue
	`color =[2,1,3,0,1,3,1,0,2];`
	`cName ={'yellow','red','green',blue'};`
	`for ind=1:length(color)`
	`  disp(cName{color(ind)+1});`
	`end`
	Note: The values of the color vector are used as index for the cName cell. However, the vector also contains zeros, and for this reason a +1 is used

1.3 Generate a cell with ten element. Each element is a vector of random length (between 1 and 10) containing ones if the length is odd and zeros if the length is even	`NumOfVector = 10`
	`for i = 1:NumOfVector`
	`  Vlen=floor(rand*10)+1;`
	`  V{i} = ones(1,Vlen)*rem(Vlen,2);`
	`end`
	Note: To obtain a random number between 1 and 10, we create a random number between 0 and 1 (using rand), then we multiply it by 10, and take the floor to obtain an integer

2. Repeat Exercise 1.2 using a while loop.
3. Repeat Exercise 1.2 by substituting line 4 with the `switch-case` command.
4. Exit from the for loop of Exercise 1.3 if the vector length is equal to 7 (use `if` and `break`).
5. Rewrite Exercise 4 using `while`.
6. Write a function that displays the bar and the error bar graphics in the same figure having as input the x value, the y value, the color of the bar and the length of the error. Inside your function use the MATLAB function bar and errorbar.

A Brick for an Experiment

Now we have sufficient knowledge to write the brick script. As we wrote previously in this chapter, when you plan to program an experiment, there are two choices: you can write a single long program or you can write several subprograms and one main program that calls each of the subprograms. For the brick we use an approach similar to the second.

Let's begin by writing an M-script with only comments in it and save it as `SexulerExp.m`:

Listing 4.18

```
% M-script to realize a experiment based on the crossmodal perception
% The experiment first performed by Sekuler, Sekuler and Lau (1997)
% Author: Borgo, Soranzo, Grassi 2012
% EXPERIMENT'S SETTINGS
% STIMULI (SELECTION)
% STIMULI (CREATION)
% STIMULI (PRESENTATION)
% COLLECT SUBJECT'S ANSWER
% STORE RESULTS
```

Remember that comments are important. It is important to comment your program so that you know, when you read your code after a certain time, what the various parts of the code were there for. Here, the comments highlight the crucial points of the program.

Now we can write one of the corollary functions and program a function that writes an event table. As we have written in this chapter, the majority of experiments in psychology are fixed-stimuli experiments, i.e., we know in advance the stimuli we are going to present, how many times we are going to present them, and even the random sequence of trials that we will present to the subject. This is an event table: the storyline of the sequence of events that occur during the experiment, in other words, a place where the particular stimulus we have to present in each successive trial is written. In MATLAB, the event table is a matrix organized in rows and columns, where each row represents one trial and each column represents one variable. The event table we design here has the trial number in the leftmost column and in the right columns, one variable that represents the kind of disc motion (continuous or with the stop) and one variable that represents the presence/absence of the sound. Once we have set and written the event table, when we are running the experiment, at the moment we have to present a stimulus to the subject, we read the content of the event table for that specific trial so that we know which stimulus is to be presented.

Let's write the function `GenerateEventTable`. This function will receive as input the conditions of the experiment, the number of repetitions for each condition, and a logical value that informs the function whether the trials are to be written in a random or fixed order.[2] The conditions have to be passed to the function in matrix form. We define a matrix in which each row represents one combination of factors and each column represents one factor. In brief, for the experiment that we implement in the brick, the variable "conditions" could be written as follows:

```
>> cond = [1, 1; 1, 2; 2, 1; 2, 2]
```

Here the 1, 1 content of the first row represents the continuous motion (1, left column) without sound (1, right column), the 1, 2 content of the second row represents the continuous motion (1, left column) with sound (2, right column), and so on.

Listing 4.19

```
function EventTable = GenerateEventTable(conds, repetitions, isfixed)

% function GenerateEventTable(conds, repetitions, isfixed)
%
% reads text file and displays it to the console
%
% INPUT :   conds = vector containing factors/number
%       :   repetitions = number of repetitions
%       :   isfixed = 0: random order  1: fixed order
% OUTPUT:   EventTable = is a vector of struct
%
% Author: Borgo, Soranzo, Grassi 2012

EventTable = [];
for i=1:repetitions
   EventTable = [EventTable; conds];
end

TotalNumberOfTrials = length(EventTable(:, 1));

if isfixed == 1
   FixedTrialSequence = (1:TotalNumberOfTrials)';
   EventTable = [FixedTrialSequence, EventTable];
else
   RandomTrialSequence = randperm(TotalNumberOfTrials)';
   EventTable = [RandomTrialSequence, EventTable];
   EventTable = sortrows(EventTable, 1);
end

```

[2] It is often useful to run an experiment in fixed order, in particular when you are debugging the experiment.

Analysis

Let's analyze the function. As you can see, at lines 14–17, the function `GenerateEventTable` repeatedly copies the condition variable to itself. This is done for the number of times the stimuli will be repeated during the experiment. This copying process returns a variable that is very similar to the final event table (although without the trial number). At line 19 we calculate the number of trials. The number of trials is equal to the number of rows of the `EventTable` variable created at line 16. Lines 21–28 control the `isfixed` variable, and according to its value, write the final event table. If `isfixed` is equal to 1, the function appends to the left of the event table a successive number (i.e., the trial number) but leaves the rest of the EventTable variable unchanged. In contrast, if `isfixed` is equal to 0 (i.e., the event table has to be sorted in random order), the function first generates a random sequence of `TotalNumberOfTrials` by means of the function `randperm` (see row n. 26). Successively, the function appends to the left of the `EventTable` matrix the vector just created (row 26). Finally, the content of the `EventTable` matrix is sorted according its leftmost column (column number 1), i.e., the column that contains the trial number written in random order. This sorting randomizes the rows of the event table, i.e., randomizes the trial sequence of our experiment.

Now let's return to the main script (i.e., `SekulerExp`) and modify it as follows:

Listing 4.20

```
1  % M-script to realize a experiment based on the crossmodal perception
2  % The experiment first performed by Sekuler, Sekuler and Lau (1997)
3  % Author: Borgo, Soranzo, Grassi 2012
4  % EXPERIMENT'S SETTINGS
5
6  % set the experiment details
7  conditions = [1, 1; 1, 2; 2, 1; 2, 2];
8  repetitions = 20;
9  EventTable = GenerateEventTable(conditions, repetitions, isfixed);
10 TotalNumberOfTrials = length(EventTable(:, 1));
11
12 % STIMULI (SELECTION)
13 % STIMULI (CREATION)
14 % STIMULI (PRESENTATION)
15 % COLLECT SUBJECT'S ANSWER
16 % STORE RESULTS
```

We add the settings (rows 7–8), we call for the `GenerateEventTable` function (row 9), and calculate the number of trials by reading the event table (row 10).

Let's take the script a step further. As we have written in this chapter, fixed-stimuli experiments can be programmed with a `for` loop. We can now write the main for loop driving the experiment. In the `for` loop, each time we make one cycle, the subject performs one trial. Therefore, each time we make one cycle we generate

the appropriate audio/video stimulus that we have to present by reading the content of the `EventTable` matrix:

Listing 4.21

```
1  % M-script to realize an experiment based on the crossmodal perception
2  % The experiment first performed by Sekuler, Sekuler and Lau (1997)
3  % Author: Borgo, Soranzo, Grassi 2012
4  % EXPERIMENT'S SETTINGS
5
6  % set the experiment details
7  conditions = [1, 1; 1, 2; 2, 1; 2, 2];
8  repetitions = 20;
9  if nsub == 0
10    repetitions = 1;
11 end
12 EventTable = GenerateEventTable(conditions, repetitions, isfixed);
13 TotalNumberOfTrials = length(EventTable(:, 1));
14 for trial = 1:TotalNumberOfTrials
15
16    % STIMULI (SELECTION)
17    VideoStimulusToPlay = EventTable(trial, 2);
18    SoundStimulusToPlay = EventTable(trial, 3);
19
20    % STIMULI (CREATION)
21    % STIMULI (PRESENTATION)
22    % COLLECT SUBJECT'S ANSWER
23 end
24 % STORE RESULTS
```

At rows 14–23 there is a for loop that keeps the experiment running trial after trial. At rows 17–18, within the for loop, we read the event table to know the stimulus that we have to generate and present to the subject. Note that we use the step variable "trial" as index to read the content of the event table matrix. Moreover, in the script we introduced a feature that may help the experimenter (see rows 9–11). Often, when we run an experiment, it may be useful to have the possibility to run a short version of it. Let's insert this option in our program (lines 9–11). Here we use a rule that is similar to that used in the E-Prime and MEL software (Schneider 1990; Schneider et al. 2002). If the subject number is equal to zero, the subject runs the complete experiment (i.e., s/he observes all stimuli) but each stimulus is repeated only once and no data will be written in the data file (see later bricks).

Note that for the moment, there is no way to tell the program the subject number as well as the `isfixed` value. These will be given later through the graphical interface. Note also that we wrote the conditional if after the number of repetitions has been declared and before the `GenerateEventTable` function. Otherwise, the operation would have no effect.

References

Boring EG (1961) Fechner: inadvertent founder of psychophysics. Psychometrika 26:3–8
Fechner GT (1889) Elemente der Psychophysik, 2nd edn. Breitkopf & Härtel, Leipzig
Gescheider GA (2003) Psychophysics: the fundamentals, 3rd edn. Lawrence Erlbaum Associates, Hillsdale
Schneider W (1990) MEL user's guide: computer techniques for real time psychological experimentation. Psychology Software Tools, Pittsburgh
Schneider W, Eschman A, Zuccolotto A (2002) E-prime user's guide. Psychology Software Tools, Pittsburgh
Sperling G (1960) The information available in brief visual presentations. Psychol Monogr 74:1–29

Suggested Readings

Hahn BD, Valentine DT (2009) Essential MATLAB for engineers and scientists, 4th edn. Elsevier/Academic Press, Amsterdam
Sayood K (2007) Learning programming using MATLAB. Morgan & Claypool, San Rafael, CA
Singh YK, Chaudhuri BB (2008) MATLAB programming. Prentice-Hall of India, New Delhi