

Programming in Matlab

8 Scripts

So far, we have not been using any automated processes. What we have done is to issue one or a few commands from the Command Window to perform tasks step-by-step. This is one way to use Matlab. If you have a small-scale problem, it can often be solved by issuing just a few commands. Furthermore, if you are unsure of how to go about solving a certain problem, trying a few different approaches, by issuing commands from the Command Window, is often a good first step.

However, there are many situations in which writing a program simplifies the task. Many problems involve repeating the same operations many times. This could be so in different ways. It could be that you want to make new estimates after new data has become available. Then you solve the same problem again, including the new data points. Alternatively, it could be that the problem requires that you perform a simulation, where you repeat the same process many times with random input data. Or it could be that you have a large data set and want to perform the same calculations, again and again, on each data point. In cases like these, writing a program is a good idea.

Matlab differentiates between two types of programs: *scripts* and *user defined functions*. Scripts are similar to automatically executing the same type of command lines as you, up until now, have been issuing manually. All variables defined within the script appear in the Workspace and you can use them after the script has finished.

User defined functions, on the other hand, are similar to the functions you have used, for example `zeros(1,3)` and `sortrows(testMatrix,2)` that we used in sections 4.1.1 and 4.4, respectively. When a user defined function is executed, the operations of the program are hidden from the user, and variables used within the program do not show up in the Workspace and are not possible to use afterwards. You might not even have thought about the possibility that, for example, the function `sortrows()` uses any internal variables. (It does.) The distinction between scripts and user defined functions will probably become clearer as we describe the latter in Section 9.

8.1 The Editor

A program is simply a list of commands. When the program is run, the commands are executed in order from beginning to end. Since a program is just a list of commands, you can use any text editor to write a program, for instance Notepad. However, it is much more convenient to write the program in the Matlab Editor. The Editor is integrated in Matlab and has many features that make programming easier, such as color-coding of selected keywords, automatic row numbers, and a powerful debugger that provides tips on how to correct or improve programs.

You open the Editor by entering the command `edit` from the Command Window, or by choosing `Desktop > Editor`. If you use the second option, you also have to open a new script document in a second step, for instance by choosing `File > New > Script`. If you want to dock/undock the Editor, you click the arrows in the upper-right corner, as described in Section 2.

As mentioned, there is a debugger feature in the Editor. This debugger will highlight parts of the text you write, underline some parts, and, if you hover with the mouse over the underlined parts, give balloon-style tooltips. This is sometimes very helpful, but it can also be quite annoying. In Section 12.1, we will describe how to use this feature. For now, if you want to turn it off, choose `File > Preferences...`, then first click "Code Analyzer" in the menu to the left in the new window, and then uncheck the tick box labeled "Enable integrated warning and error messages". Click "OK" to close the window and return to the Editor.

8.2 Writing a script

Let us use the commands from Section 7.2 that produce a plot of a random series, as an example of how to write a simple script. In the first version of the script, we just want it to reproduce what we did before. Let us collect the lines from before and type them into the Editor.

```
obs = cumprod(1+randn(600,1)/100);
dates = 1950+1/24: 1/12: 2000;
plot(dates, obs, 'g')
title('Levels during 1950 to 2000')
xlabel('Year'), ylabel('Level')
grid
legend('First observations')
```

If you entered the command lines earlier, it is possible to use the Command History window as a shortcut to typing the whole script from the keyboard. Just select the appropriate lines in the window by clicking on them. More than one line can be selected by holding down `<Ctrl>` while clicking new lines. When you have selected the lines you want, you can drag-and-drop them in the Editor or you can copy them and then paste them in the Editor. Alternatively, you can right click one of the selected (and highlighted) lines in the Command History window, and then select `Create Script` from the context menu. In the latter case, Matlab opens a new window within the Editor and copies the lines to that window.

Before we can run the script, we need to save it. It is often convenient to save programs to the Current Folder. Select `File > Save` and the Current Folder is the preselected destination folder. In the dialogue window that appears, choose a name for the script, for example `randomPlot.m`. The script needs the extension `.m` to work, but if you save it from the Editor, the extension is automatically added to the name. If you try to run the program before it is saved, you will be asked to save it first.

After you have saved it, it is possible to run the script. There are a few different ways to do that, and you can choose the one that is most convenient for you. You can click the icon  at the top of the Editor window, you can press F5 while in the Editor Window, or you can issue the name of the script as a command in the Command Window.

```
>> randomPlot
```

In either case, a random plot with years from 1950 to 2000 on the X-axis, axis-labels, and a title appears, just as it did in Section 7.2. If you run the script repeatedly, new plots appear with new sets of random data. As is typical for a script, the variables defined and used within the script, `dates` and `obs`, appear in the Workspace, and you can access them after the script is run. It is also possible to use variables defined outside the script from within the script, if you would like to do that. In short, running a script is just like issuing the commands manually, one-by-one, only faster.

Suppose we want to define the title of the graph manually, outside the script. To do this, we can change the line `title('Levels during 1950 to 2000')` to `title(titleVar)`. After having changed the line, you *must save the script again* to implement the change. After having done this, we go to the Command Window and define the title variable. Then, issue the script name to rerun the program.

```
>> titleVar = 'Level evolution';
```

```
>> randomPlot
```

The plot now has the new title.

It is very useful to add comments to programs. In the present case, we should at least add a comment with the name of the script at the top, and a brief description of what the script does.

```
% RANDOMPLOT
% This script creates a random series of changes and turns them
% into an index-like series by multiplying cumulatively.
% A series of dates of the same length is also created.
% Finally, the series is plotted against the dates.
```

As you see, comments are, by default, color-coded green.

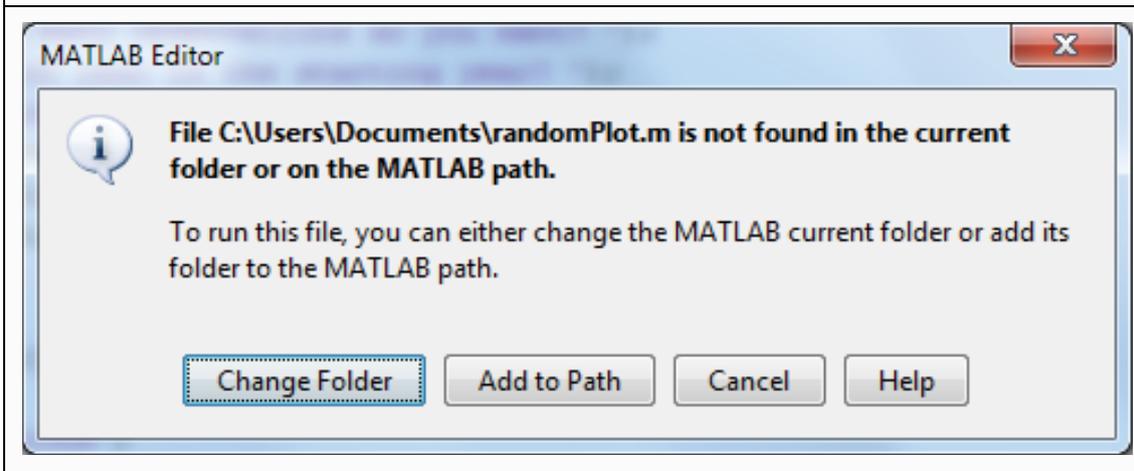
8.3 The search path

Often, it is convenient to save programs to the Current Folder. This is at least true while developing a new program and if you do not have many programs saved there already. However, with time it will become necessary to sort programs in different subfolders. You may also want to save programs in a completely different location in the folder hierarchy. If you just move a program to another location, Matlab will usually not be able to find it, and you will get an error message when you try to run it.

What happens internally when you issue a command is that Matlab starts looking for a match for the command that you have issued. (Compare with the discussion in Section 3.7.) First, it searches the Workspace, and then it searches the Current Folder. If no match is found, it starts searching other folders. However, it does not search the whole computer; it only searches folders that have been added to its “search path”, in the order that they are listed. Therefore, if you want to use a folder that is not already on the search path, you have to explicitly add it.

To open the dialogue for this, choose `File > Set Path...`, alternatively issue the command `pathtool`. To add a new folder to the path, click "Add folder..." and find the folder in the hierarchy that appears in a new window. Select it and click "OK" to add it. If you want the addition to be permanent, so that Matlab will remember the path the next time you open it, you also have to click "Save" before clicking "Close" to exit the dialogue. If you do not save the information, the added folder will only be accessible until the next time you close Matlab.

Figure 8-1: Change folder/add path dialogue



If you try to run a script that is not saved on the path or in the Current Folder, using either F5 or by clicking the run-button, a dialogue is opened. It will ask whether you want to change the Current Folder to the location of the script file, or if you want to add the folder that contains the file to the path (see Figure 8-1). Note that, this does not happen if you issue the script name from the Command Window, as Matlab cannot find the file in that case.

8.4 User interaction with the script

As the program is written, you now have to enter the graph title from the Command Window and you cannot change any other features without changing the script itself. However, suppose you want to be able to produce graphs that are slightly more dynamic. You may want to be able to produce graphs of different lengths as well as with different starting years. It is then convenient if the script lets you enter this information interactively (i.e., that it stops to ask you for input). We add three lines at the beginning of the script that ask for input.

```
nOfObs = input('How many observations do you want? ');
startYear = input('Which year is the starting year? ');
titleVar = input('Which title do you want for the graph? ');
```

When we rerun the script, it now first stops to ask for the number of observations and waits until we have entered a number and pressed <Enter>. When we do that, the variable `nOfObs` is assigned the value that we enter. Then, similarly, the script asks for the starting year and a title. Note that you have to enter the title as a string (i.e., enclose it within single quotes). Now, for the program to actually produce what we have asked it to do, we have to change how the variables `obs` and `dates` are created.

The number of observations is easily changed by changing the line `obs = cumprod(1 + randn(600,1)/100);` to `obs = cumprod(1 + randn(nOfObs,1)/100);`. However, for the dates variable, we also have to calculate the value with which to end the matrix. Earlier, we did this manually. The starting value is easy. It will be `startYear + 1/24`. The increment, the value we increase each consecutive observation with, is `1/12`, as before. The last observation must then be `startYear + 1/24 + 1/12*(nOfObs-1)`.

As a new feature, let us also add explicit output from the program, where it specifies the starting year, the ending year, and the number of observations. One way to do that is to delete the semicolons at the end of the lines where the corresponding variables are defined. That, however, produces very ugly output. It is more appealing to use the `disp()` command. One problem here, though, is that we need to mix character data with numerical data. To circumvent that, the numerical data has to be translated to character data using the `num2str()` function. Since the information we want to display in each case is a concatenation of two matrices, first a character string and then a translated numerical variable, we have to enclose the two parts within square brackets.

The updated script will then look like this, where the creation of dates is done in two steps.

```
% RANDOMPLOT
% This script creates a random series of changes and turns them
% into an index-like series by multiplying cumulatively.
% A series of dates of the same length is also created.
% Finally, the series is plotted against the dates.
nOfObs    = input('How many observations do you want? ');
startYear = input('Which year is the starting year? ');
titleVar  = input('Which title do you want for the graph? ');
obs       = cumprod(1+randn(nOfObs,1)/100);
endYear   = startYear+1/24+1/12*(nOfObs-1);
dates     = startYear+1/24: 1/12: endYear;
plot(dates, obs, 'g')
title(titleVar)
xlabel('Year'), ylabel('Level')
grid
legend('First observations')
disp(['Start year: ' num2str(startYear)])
disp(['End year: ' num2str(round(endYear))])
disp(['Number of observations: ' num2str(nOfObs)])
```

Save the script and run it a few times with different input values to see that it works as expected.

9 User defined functions

We have already used functions many times. For example, we used `cumsum()` and `randn()` in the previous section, and we have listed many useful functions throughout the book.

User defined functions are similar to these functions. One similarity is that you call them in the same way. You type the function name and enclose one or several arguments within parentheses after the name, for example `randn(5,2)`, which uses two input arguments and calls a function that creates a 5x2 matrix of random numbers. Another similarity is that variables used within the function, do not show up in the Workspace after the function has been executed. You could think of this as that the inner workings of the function are hidden from the user. This is often an advantage, as it prevents the Workspace to become cluttered with variables that you do not need.

Unlike scripts, user defined functions must begin with a declaration that it is a function, and you must specify input and output variables. The general form of this declaration is

```
function [output] = functionName(inputArguments)
```

Here, the word `function` states that the program is a user defined function, `output` is one or a list of several variables that the function returns, `functionName` is a name that the user gives to the function and that is later used to call it. `inputArguments` is one or a list of several variables that the function uses as input. If there are several input arguments they must be separated by commas, but output variables can be separated by either commas or blank space. If there are more than one output variable, they must be enclosed within square brackets, but if there is only one, no brackets are needed. (Note that all these requirements adhere to the syntax of functions described in Section 3.3.) The word “function” is color-coded blue, to show that it is a keyword in Matlab. The help text at the beginning of the function also has special uses, which are described later in sections 13.1.1 and 13.1.2.

Let us write a simple function to show how this works. In the previous section, we created a matrix of the cumulative product of random numbers, where the desired number of observations was supplied by the user (i.e., the line `obs = cumsum(randn(nOfObs,1))`). This could be done with a simple function. The input argument is just one number, the desired number of observations, and the output is a matrix, similar to the one from the previous section. So, to write the function, open a new window in the Editor, enter the following code, and then save it.¹³

```
function obs = randomIndex(nOfObs);  
% RANDOMINDEX  
% Produces an index with random changes and nOfObs elements.  
obs = cumprod(1 + randn(nOfObs,1)/100);
```

When saving a new function, Matlab always suggest that you save it under the name you have entered as function name. This is good practice. If you use different names, then the function can only be called using the name you saved it with, so the name in the text of the function will be meaningless.

Now, in the Command Window, issue the function as a command. (A function cannot be executed using F5 or clicking the run-button.)

```
>> randomIndex(5)  
ans =  
    -1.3489  
    -0.045377  
    -1.665  
    -1.9938  
    -3.6051
```

There are a few things to note here. As already mentioned, the variables within the function are local (i.e., they only exist within the function). When you call the function using `randomIndex(5)`, the value 5 is supplied to the function, and there the variable `nOfObs` is assigned the value 5. Then, inside the function, a new variable, `obs`, is defined and it is set equal to the cumulative product of a set of random variables. Lastly, the value of `obs` is returned from the function to the Workspace. Since we have not entered a variable name, Matlab sets the variable `ans` equal to the result. None of the variables `nOfObs` and `obs` is accessible outside the function. To make this obvious, we clear the Workspace from all variables and reissue the command.

```
>> clear
>> numbers = randomIndex(5);
```

The only variable listed in the Workspace now is the new variable `numbers`. Note, also, that variables in the Workspace are not accessible from within the function either. Therefore, all information that the function needs must be entered in the form of input arguments, or be imported into the function by some other method, for instance from an Excel file as in Section 6.6.2.

Note that, if the function has any output, the output variable(s) must be explicitly defined within the function. Not all functions have input or output, though. For instance, a function might use random data that is defined within the function, and the output could be a graph instead of variables.

We can now change the script from Section 8.4 to use our new user defined function. Just change the line `obs = cumsum(randn(nOfObs,1));` to `obs = randomIndex(nOfObs);`.

9.1 About the differences between scripts and user defined functions

When should you write a script and when should you write a function? Consider first the main differences between the two.

- Scripts use the existing Workspace whereas functions use their own. Variables used in functions are consequently hidden from the user and are not accessible outside the function.
- Functions usually return explicit output whereas scripts do not.
- Running scripts is similar to rerunning code you have issued from the Command Window. Running a user defined function is similar to executing Matlab functions.

These differences make for the following rule of thumb, which usually, but not always, is correct: If you are solving a general problem, use a function; if you are solving a specific problem, use a script.

9.2 More about functions

- You can open functions directly into the Editor by issuing `edit functionName`. You can also open many Matlab functions this way. Note, however, that many of the basic functions are built-in functions and these are not viewable. Try, for instance, `edit mean` to see how Matlab calculates the mean.
- Matlab also has a concept called “anonymous functions”. This is a way to enter a function quickly without storing it to a file. See `help function_handle` and search “anonymous functions” in the Help Browser. (The Help Browser is described in Section 13.2.1.)

10 Flow control

The programs we developed in sections 8 and 9 were executed in a linear fashion (i.e., execution started with the first line of code and then moved to the second line, and so forth, until the last line was reached, after which it ended). Oftentimes we want to make programs that are more complex than that, though. First, we want to be able to repeat specific tasks many times, possibly with minor changes each time. That is convenient to do in a so-called loop. Second, we want the program to be able to make decisions, so that, depending on different circumstances, it performs different operations.

Before we begin, it is good to learn the key combinations `<Ctrl>-C` or `<Ctrl>-<Break>`. Both of these break the execution of a program. Sometimes it happens that, by a programming mistake, a program goes into an infinite loop, or it might go into a loop that will take more time to execute than you are willing to wait. Then it is essential to know how to get out of that.

10.1 Loops

A loop is a collection of commands that is repeated a certain number of times. There are a few different ways to do this. One of the most frequently used is the for-loop. This is particularly useful when you want to repeat a task a certain number of times, say exactly 10 times.

The simplest form this command takes is

```
for counter = startValue:increment:endValue
    :
    :
end
```

Here, `for` indicates that a for-loop begins and `counter` is the name of the so-called counter variable. The part `startValue:increment:endValue` is a matrix created using the colon operator. Finally, `end` indicates where the loop ends. The part between the `for` and the `end` statements is what is repeated. Note that, the `for` and the `end` statements, when written by themselves on separate rows, do not have to be succeeded by a comma or a semicolon as most other commands do. Note also that, for-loops can be nested, so that there can be a loop within another loop.

To give an example of a simple loop, consider the case of the Fibonacci numbers. This is a series of numbers where the first two are equal to one, and then each consecutive number is equal to the sum of the previous two. The first twelve are then 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and 144. Say we want to write a user defined function that produces a vector of Fibonacci numbers, where we can choose how many elements we want. This can be easily done in a for-loop.

```
function fib = fibonacci (fib_max)
% FIBONACCI
% Produces a column vector of fib_max Fibonacci numbers.
fib = ones (fib_max, 1);
for fib_nr=3:fib_max
    fib (fib_nr, :) = fib (fib_nr-1, :) + fib (fib_nr-2, :);
end
```

This code first defines a column vector of ones to hold the Fibonacci numbers. Then, starting from the third number, it calculates the sum of the preceding two numbers, and then repeats this process until the last requested number is reached. Then it returns the vector of Fibonacci numbers.

Note the line `fib = ones (fib_max, 1);`. What this does, besides setting the first two numbers equal to 1, is to preallocate space in a vector that will hold all the numbers. This ensures that the vector is large enough to hold all numbers already from the beginning. Suppose line was changed to `fib (1, :) = 1;` `fib (2, :) = 1;`. The program would then produce exactly the same output vector. However, at each new step, the vector would grow by one element, forcing Matlab to redefine it every time. And this takes much more time. Therefore, preallocating in this type of situation is a standard procedure in Matlab.

As you see, the line within the loop is indented. This useful convention makes the code easier to read. However, it does not change the way the code is executed. If you write the code in one single line (in which case you have to end the `for`- and `end`-statements with a comma or a semicolon), the output is the same. It will be more difficult to read, however. When you type the code into the Matlab Editor, the lines after a `for`-statement will automatically be indented.

Usually, loops appear within programs. However, it is quite possible to use them in the Command Window as well. If you want to type it there, using several lines before starting execution, instead of ending the lines with `<Enter>`, you press `<Shift>`-`<Enter>`. This just moves the cursor to the next line without starting execution. At the last line, you press `<Enter>` and Matlab executes the whole code.

As an alternative, you can end each line with `<Enter>`. Then, when issuing the first three lines of code, nothing seems to happen, except that the `>>` prompt, that usually appears at the beginning of each row, disappears. This is because Matlab waits for the `end` command that concludes the loop. Before that has been issued, Matlab does not know what to repeat. When, finally, the concluding `end` is issued, the whole loop is executed.

Yet another alternative is to enter the whole set of commands as one long line in the Command Window, separated by commas or semicolons. To give an example and, at the same time, give an idea of how time consuming it is not to preallocate space for variables that grow inside loops, we issue the following code in the Command Window.

```
>> tic, for run_nr=1:10000; fibonacci(1000); end, toc  
Elapsed time is 3.873173 seconds.
```

Here, `tic` saves the time when it was issued and `toc` calculates the amount of time that has passed since `tic` was issued, thereby imitating a stopwatch. Apart from that, a vector of the first 1000 Fibonacci numbers is created, and this is repeated 10000 times. All this takes approximately 3.9 seconds. Then we change the function line where `fib` is preallocated to `fib(1,:) = 1; fib(2,:) = 1;`, save the function, and rerun the code to time it again.

```
>> tic, for run_nr=1:10000; fibonacci(1000); end, toc  
Elapsed time is 17.703510 seconds.
```

Evidently, we save about 14 seconds, or almost 80%, in this case.

10.2 Relational and logical operators

Conditional statements are statements that are executed only if certain conditions are fulfilled. Hence, they enable programs to make decisions. Operators that are used in conditional statements fall in one of two categories, relational operators and logical operators. We have already used some of these in sections 3.5.2 and 4.2.3. Remember that the outcomes of operations with relational or logical operators are *true* or *false*, and that these are indicated by 1 or 0.

Relational operators

<code><</code> , <code><=</code>	“Is less than” and “is less than or equal”, respectively. For example, <code>2 < 3</code> is true and produces the output 1.
<code>></code> , <code>>=</code>	“Is greater than” and “is greater than or equal”, respectively. <code>2 > 3</code> is false and produces the output 0.
<code>==</code>	Is equal to. (Note: There are <i>two</i> equality signs to separate this operator from the assignment operator). <code>5 == 5</code> is true.
<code>~=</code>	Is not equal to. <code>5 ~= 5</code> is false.

Logical operators

<code>&</code>	Logical and This is true if both the statement to left and the one to the right of <code>&</code> are true, else it is false. <code>2<3 & 5>4</code> is true, since both <code>2<3</code> and <code>5>4</code> are true. When comparing scalars, Matlab prefers double <code>&:s</code> , as this has additional functionality. In the statement <code>0 & 4</code> , for example, the first part is false (0 is false). Then the rest of the statement does not have to be evaluated at all, since an and-statement where one element is false, by necessity has to be false. If you change this to <code>0 && 4</code> , Matlab takes advantage of this fact and only checks the second part when it is necessary (i.e., when the first part is true). Another benefit of this is that, if the second part is a variable that may be undefined, it does not have to be checked if the first part is false, thereby preventing an error message and termination of the program.
<code> </code>	Logical or This is true if either the statement to left or the one to the right of <code> </code> is true or if both statements are true, else it is false (i.e., it is only false if both statements are false). The character <code> </code> is usually produced by typing <code><Ctrl>-<Alt>-<</code> (i.e., the button for “less than”). <code>2<3 5>4</code> is true, since, again, both <code>2<3</code> and <code>5>4</code> are true. Similarly to <code>&</code> , when comparing scalars, it is preferable to use double <code> :s</code> . Then the second part is only evaluated if the first part is false.
<code>~</code>	Logical not This is true if the statement to the right of <code>~</code> is false, and false if it is true. The character <code>~</code> is usually produced by typing <code><Ctrl>-<Alt>-`</code> (i.e., the button for “dieresis”, often located to the left of the button <code><Enter></code>). For example, <code>~(3<2)</code> is true, since <code>3<2</code> is false.

Note that all numbers, except zero, are true. Zero is false. For example, the statement $2 \ \& \ 3$ is true while the statement $2 \ \& \ 0$ is false.

Also note that, matrices are compared element-by-element, as we saw in Section 5.1. So the statement $[1 \ 2] \ > \ [0 \ 5]$ evaluates to a logical 1x2 matrix, where the first element is true and the second is false: $[1 \ 0]$.

We can now extend the list of the order of precedence from Section 3.1 to include the relational and logical operators.

1. parentheses
2. exponentiation
3. logical not; \sim
4. multiplication, division
5. addition, subtraction
6. relational operators
7. logical and; $\&$
8. logical or; $|$

As before, if there is a tie, the expression is evaluated from left to right. Again, if you want to be absolutely certain about the order, use parentheses, which can also improve readability.

For example, the statement $\sim(0 \ >= \ 0)$ is false. This is because the expression within the parentheses, $0 \ >= \ 0$ (which is true), has precedence over \sim (which makes the true statement false). However, the statement $\sim 0 \ >= \ 0$ (without parentheses) is true. This is because \sim now has precedence over $>=$. ~ 0 is first evaluated to be 1, and $1 \ >= \ 0$ is true. In this latter case, note that when the operator needs numerical data ($>=$ compares numbers, not true/false), the logical variables are interpreted as numbers (i.e., true is interpreted as 1 and false as 0).

10.3 Conditional statements

We can now use the relational and logical operators to write conditional statements. The form of a conditional statement is

```
if ...  
    :  
    :  
elseif ...  
    :  
    :  
else  
    :  
    :  
end
```

The command `if` indicates that a conditional statements begins. After this, a condition that is true or false follows on the same line. On the following lines(s), there is code that is executed if the condition is true. After this follows either an `end`-statement or an `else`-statement. If there is an `else`-statement, the code following the statement is executed if and only if the initial condition is false. The `else`-statement comes in two versions. Either it is an unqualified `else`-statement, or it contains additional conditions. In the latter case, it is issued as `elseif` and followed by the additional conditions. The whole section must end with an `end`-statement, so that Matlab knows where the conditional part ends.

The following function provides an example of using conditional statements. The input is the score from a test, and the output is a grade from A to C, or “fail”, depending on how high the score was.

```
function grade = score2grade(testScore)
% SCORE2GRADE
% Calculates grade from the score at a test.
if testScore >= 90
    grade = 'A';
elseif testScore >= 80
    grade = 'B';
elseif testScore >= 70
    grade = 'C';
else
    grade = 'fail';
end
```

In the fourth line, the value of `testScore` is compared to 90. If it is greater than or equal to this number, it continues with the following line until it reaches the statement `elseif`, where it skips everything until the end-statement. If, on the other hand, `testScore` is not greater than or equal to 90, the program skips all lines until it reaches the first `elseif`-statement, where it compares `testScore` to 80. Again, if it is greater than or equal to 80, it continues with the following lines until it reaches the next `elseif`-statement, where it skips to the end, etc. Note that this implies that grade B is assigned test scores between 80 up to, but not including, 90. If none of the conditions applies, the grade is set to `fail`.

Note that only the `for`- and `end`-statements are required; the `elseif`- and `else`-statements are optional. Note also that, it is possible to nest conditional statements, so that you can have another `if... else... end` structure within another.

10.4 More about flow control

- Note that the argument in a `for`-loop (i.e., the `1:5` part of `for counter = 1:5`) is a vector. You do not have to use a vector that is created with the colon operator. You can, for example, enter a vector manually, such as `counter = [1 5 2 4 3]` if this is the order you want. You can even enter a matrix as argument, in which case the loop-variable becomes a column vector, and is executed as many times as there are columns in the matrix.
- In Matlab, ordinary loops can often be replaced with vector operations. For instance, creating a column vector of squares from the numbers 1 to 100 can be done in a loop. Nevertheless, it is faster and simpler to do it as `(1:100)' .^2`.
- Another type of loop is the `while`-loop. This kind continues to loop until a certain condition is not fulfilled any more. For instance, `while x<10... end` loops as long as `x` is less than 10. This is useful when the number of loops is indeterminate. See `help while` for more information.

- Note that it is slightly complicated to compare strings in conditional statements. For example, `'YES'=='Yes'` evaluates to `[1 0 0]`, since each letter is compared individually, element-by-element. Furthermore, letters in small caps and letters in large caps are not equal to each other. To compare strings, use a construction like `isequal(upper('Yes'),upper('YES'))`. `upper()` changes text to upper case and `isequal` compares the whole string instead of the individual elements.
- The command `switch` can be used for executing different code parts depending on a small number of different cases. See `help switch`.
- Loops can have certain conditions inserted where they either break the loop completely, or move on to the next loop without finishing the present one. See `help break` and `help continue`. Another type of break is to leave the entire program that is being executed. For that, see `help return`.

11 Numerical analysis and curve fitting

Now that we have learnt to program functions, we are able to use Matlab's numerical analysis and curve fitting tools. Here, we describe how to numerically solve equations and find local minimum points, as well as how to perform numerical integration.

11.1 Solving equations

There are several ways to define functions in Matlab. In Section 9, we learnt how to write user defined functions. Let us use the function we plotted in Figure 7-3, define that as a user defined function, and then solve it (i.e., find the point where the function equals zero). We write the function, named `waveFunction`, such that we supply an X-value and then get back the corresponding Y-value. For such a simple function, this is easily done.

```
function y = waveFunction(x)
y = sin(x) + exp(-x);
```

To start searching for a solution, we need a starting value. Looking at Figure 7-3, we see that the function equals zero where x is roughly equal to 3. Then we issue the command `fzero()` as¹⁴

```
>> x_zero = fzero('waveFunction',3)
x_zero =
    3.1831
```

The first input to `fzero()` is the name of the function we want to solve, within single quotes, and the second is a "best guess" where a solution will be found. The present function has many solutions, but only one that is close to 3. Entering other start values can produce other solutions. The output, `x_zero`, is the value of x close to 3 that makes y equal to zero. In our case, the minimum point is found here x equals 3.1831.

11.2 Finding a function minimum point

It is equally easy to find a local minimum of a function. We see in Figure 7-3 that there should be a local minimum where the red circle marker is, somewhere between 4 and 5. The command for finding a minimum between two values on the X-axis is `fminbnd()`.

```
>> [x_min, fval] = fminbnd('waveFunction', 4, 5)
x_min =
    4.7213
fval =
   -0.99106
```

The second and third input arguments are the end values of the region along the X-axis where we want to search for a minimum. One is found at 4.7213, and the corresponding Y-value, `fval`, at that point is -0.99106. (Note that, this is slightly smaller than the value we found when plotting the minimum point in the figure in Section 7.3. The difference depends mainly on how fine grained the x-vector was there.)

11.3 Numerical integration

To numerically integrate our function between -1 and 2π (i.e., the plotted region), there are several different methods to choose from. For instance, we can issue

```
>> area = quad('waveFunction', -1, 2*pi)
area =
    2.2567
```

The area beneath the function is approximately 2.2567.

11.4 Curve fitting

Matlab also has a convenient tool for curve fitting. If we have two vectors, `x` and `y`, with paired observations, we can approximate the functional relation between them with a polynomial of some degree. If the degree is 1, the relation is linear; if it is 2, the relation is quadratic, etc. This can be done with the function `polyfit()`. The following script estimates the coefficients of polynomials of order 1, 2, and 3, for a given set of observations, and plots the results in three graphs.

```
x = [1 2 3 4 5 6 7 8 9]; y = [2 3 3 5 7 8 8 9 7];
x_val = linspace(0,10,100);
for degree=1:3
    poly = polyfit(x,y,degree);
    disp(['Coeff., case ' num2str(degree) ': ' num2str(poly)])
    y_val = polyval(poly,x_val);
    subplot(3,1,degree)
    plot(x,y,'r*'), axis([0 10 0 10])
    hold on
    plot(x_val,y_val)
    ylabel(['Degree: ' num2str(degree)])
end
```

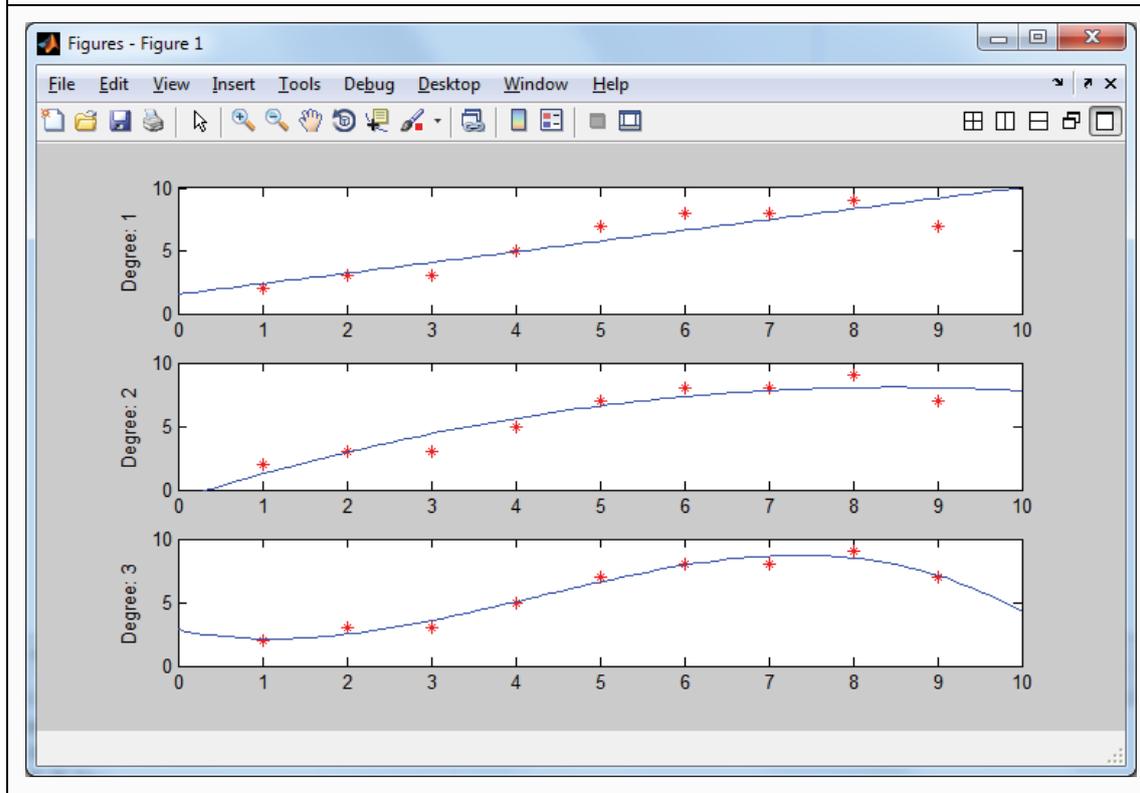
The output is

```
Degree 1:    0.85        1.5278
Degree 2:   -0.12229    2.0729    -0.71429
Degree 3:   -0.053872   0.68579   -1.3318    2.8413
```

The first two inputs to `polyfit()` are the vectors of X- and Y-values, and the third is the degree of the polynomial (i.e., the highest value of the exponent). The function responds with a matrix that holds one more element than the degree. The elements of the matrix are the coefficients of the estimated polynomial. For example, in the third case above, the output should be interpreted as $y = -0.053872x^3 + 0.68579x^2 - 1.3318x + 2.8413$.

Matlab also provides a tool for calculating the values of a polynomial, given a vector of coefficients. The function `polyval()` uses a matrix of coefficients, `poly` above, and returns Y-values for given X-values. As in the above case, the X-values can be a vector. Figure 11-1 shows the resulting three plots. The red markers are the same in all three cases, but the curves correspond to the fitted polynomials.

Figure 11-1: Examples of curve fitting



11.5 More about numerical analysis

- There are several more methods for numerical integration. See, for instance, `help quadgk` and `help quadl`.
- Matlab also supports several methods to solve differential equations. Information is best found in the Matlab documentation. See Section 13.2.1 for tips on how to search it.