CHAPTER 7

Function M-files

THE OBJECTIVE OF THIS CHAPTER IS TO ENABLE YOU TO:

- Write your own function M-files

CONTENTS

We have already seen that MATLAB has a number of *built-in* (compiled) functions, e.g., `sin`, `sqrt`, `sum`. You can verify that they are built-in by attempting to `type` them. Try `type sin` for example. MATLAB also has some functions in the form of *function M-files*, e.g., `fzero`, `why`. You can see what's in them with `type`, e.g., `type why`.

Functions are indispensable when it comes to breaking a problem down into manageable logical pieces. Hence, MATLAB enables you to create your own function M-files. A function M-file is similar to a script file in that it also has a `.m` extension. However, a function M-file differs from a script file in that a function M-file communicates with the MATLAB workspace only through specially designated *input* and *output arguments*.

We begin this chapter by extending the ideas about implementing function M-files introduced in Chapter 3, Section 3.2.2. We do this by example. We then deal with the basic rules and the various input/output possibilities available to you in the development of function M-files. For additional information and examples type `help function` in the Command Window and click on `doc function` at the bottom of the displayed information.

7.1 EXAMPLE: NEWTON'S METHOD AGAIN

Newton's method may be used to solve a general equation $f(x) = 0$ by repeating the assignment:

$$x \text{ becomes } x - \frac{f(x)}{f'(x)},$$

http://dx.doi.org/10.1016/B978-0-12-394398-9.00007-1

where $f'(x)$ (i.e., df/dx) is the first derivative of $f(x)$. The process continues until successive approximations to x are close enough.

Suppose that $f(x) = x^3 + x - 3$, i.e., we want to solve the equation $x^3 + x - 3 = 0$ (another way of stating the problem is to say we want to find the *zero* of $f(x)$). We have to be able to differentiate $f(x)$. That's quite easy here: $f'(x) = 3x^2 + 1$. We could write inline objects for both $f(x)$ and $f'(x)$, but for this example we will use function M-files instead.

Use the Editor to create and save (in the current MATLAB directory) the function file `f.m` as follows:

```
function y=f(x)
y=x^3+x-3;
```

Then create and save another function file `df.m`:

```
function y=df(x)
y=3*x^2+1;
```

Now write a *separate* script file, `newtgen.m` (in the same directory), which will stop either when the *relative* error in x is less than 10^{-8}, or after 20 steps, say:

```
% Newton's method in general
% excludes zero roots!
steps = 0;                          % iteration counter
x = input( 'Initial guess: ' );     % estimate of root
re = 1e-8;                          % required relative error
myrel = 1;

while myrel > re & (steps < 20)
  xold = x;
  x = x - f(x)/df(x);
  steps = steps + 1;
  disp( [x f(x)] )
  myrel = abs((x-xold)/x);
end

if myrel <= re
  disp( 'Zero found at' )
  disp( x )
else
  disp( 'Zero NOT found' )
end
```

Note that there are *two* conditions that will stop the `while` loop: Convergence, or the completion of 20 steps. Otherwise the script could run indefinitely.

Here is a sample run (with `format long e`), starting with $x = 1$:

```
Initial guess: 1
    1.250000000000000e+000    2.031250000000000e-001
    1.214285714285714e+000    4.737609329445558e-003
    1.213412175782825e+000    2.779086666571118e-006
    1.213411662762407e+000    9.583445148564351e-013
    1.213411662762230e+000   -4.440892098500626e-016

Zero found at
    1.213411662762230e+000
```

Note:

- The variable `y` in the function files `f.m` and `df.m` is the *output argument*. It is a dummy variable, and defines how output will be sent back to the outside world.

By the way, you realize of course that you can use your own functions from the command line, don't you? For example:

```
≫f(2)
```

should return 7 with `f.m` defined as above.

7.2 BASIC RULES

Try the following more general example, which returns the mean (`avg`) and standard deviation (`stdev`) of the values in the vector `x`. Although there are two MATLAB functions to do this (`mean` and `std`), it is useful to have them combined into one. Write a function file `stats.m`:

```
function [avg, stdev] = stats( x )         % function definition line
% STATS        Mean and standard deviation                  % H1 line
%              Returns mean (avg) and standard            % Help text
%              deviation (stdev) of the data in the
%              vector x, using Matlab functions

avg = mean(x);                                             % function body
stdev = std(x);
```

Now test it in the Command Window with some random numbers, e.g.,

```
r=rand(100,1);
[a, s]=stats(r);
```

Note the following points about function M-files in general:

- **General form of a function**

A function M-file *name*.m has the following general form:

```
function [ outarg1, outarg2, ... ] = name( inarg1, inarg2, ... )
%  comments to be displayed with help
...
outarg1 = ... ;

outarg2 = ... ;
...
```

- **`function` keyword**
  The function file *must* start with the keyword `function` (in the function definition line).
- **Input and output arguments**
  The input and output arguments (*inarg1*, *outarg1*, etc.) are "dummy" variables, and serve only to define the function's means of communication with the workspace. Other variable names may therefore be used in their place when the function is called (referenced).
  You can think of the *actual* input arguments being copied into the dummy input arguments when the function is called. So when `stats(r)` is called in the above example, the actual input argument `r` is copied into the input argument `x` in the function file. When the function *returns* (i.e., execution of it is completed) the dummy output arguments `avg` and `std` in the function file are copied into the actual output arguments `a` and `s`.
- **Multiple output arguments**
  If there is more than one output argument, the output arguments *must* be separated by commas and enclosed in square brackets in the function definition line, as shown.
  However, when a function is called with more than one output argument, the actual output arguments may be separated by commas or spaces.
  If there is only one output argument square brackets are not necessary.
- **Naming convention for functions**
  Function names must follow the MATLAB rules for variable names.
  If the filename and the function definition line name are different, the internal name is ignored.
- **Help text**
  When you type `help` *function_name*, MATLAB displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line.
  The first comment line is called the *H1 line*. The `lookfor` function searches on and displays only the H1 line. The H1 lines of all M-files in a directory are displayed under the **Description** column of the Desktop Current Directory browser.
  You can make help text for an entire directory by creating a file with the name `Contents.m` that resides in the directory. This file must contain

only comment lines. The contents of `Contents.m` is displayed with the command

```
help directory_name
```

If a directory does not have a `Contents.m` file, this command displays the H1 line of each M-file in the directory.

- **Local variables: scope**
  Any variables defined inside a function are inaccessible outside the function. Such variables are called *local variables*—they exist only inside the function, which has its own workspace separate from the base workspace of variables defined in the Command Window.
  This means that if you use a variable as a loop index, say, inside a function, it will not clash with a variable of the same name in the workspace or in another function.
  You can think of the *scope* of a variable as the range of lines over which the variable is accessible.
- **Global variables**
  Variables which are defined in the base workspace are not normally accessible inside functions, i.e., their scope is restricted to the workspace itself, unless they have been declared `global`, e.g.,

```
global PLINK PLONK
```

  If several functions, and possibly the base workspace, all declare particular variables as global, then they all share single copies of those variables.
  MATLAB recommends that global variables be typed in capital letters, to remind you that they are global.
  The function `isglobal(A)` returns 1 if `A` is global, and 0 otherwise.
  The command `who global` gives a list of global variables.
  Use `clear global` to make all variables non-global, or `clear PLINK` to make `PLINK` non-global.
- **Persistent variables**
  A variable in a function may be declared `persistent`. Local variables normally cease to exist when a function returns. Persistent variables, however, remain in existence between function calls. A persistent variable is initialized to the empty array.
  In the following example, the persistent variable `count` is used to count how many times the function `test` is called:

```
function test
persistent count
if isempty(count)
    count = 1
else
    count = count + 1
end
```

Persistent variables remain in the memory until the M-file is cleared or changed, e.g.,

```
clear test
```

The function `mlock` inside an M-file prevents the M-file from being cleared. A locked M-file is unlocked with `munlock`. The function `mislocked` indicates whether an M-file can be cleared or not.
The Help entry on `persistent` declares confidently: "It is an error to declare a variable persistent if a variable with the same name exists in the current workspace." However, this is definitely not the case at the time of writing (I tried it!).

- **Functions that do not return values**
  You might want to write a function that doesn't return values (such functions are called *procedures* or *subroutines* in languages like Pascal and Fortran, and *void* in C++ and Java). In that case you simply omit the output argument(s) and the equal sign in the function definition line. For example, the following function will display `n` asterisks:

```
function stars(n)
asteriks = char(abs('*')*ones(1,n));
disp( asteriks )
```

  Go back to Chapter 6 for an explanation of how it works.
  Write such a function file (`stars.m`), and test it out, e.g., `stars(13)` should produce 13 asterisks.
- **Vector arguments**
  It should come as no surprise that input and output arguments may be vectors, e.g., the following function generates a vector of *n* random rolls of a die:

```
function d=dice( n )
d=floor( 6 * rand(1, n)+1 );
```

  When an output argument is a vector, it is initialized each time the function is called, any previous elements being cleared. Its size at any moment is therefore determined by the most recent call of the function. For example, suppose the function `test.m` is defined as:

```
function a=test
a(3)=92;
```

  Then if `b` is defined in the base workspace as:

```
b =
     1     2     3     4     5     6
```

the statement:

```
b=test
```

results in,

```
b =
     0     0    92
```

- **How function arguments are passed**
  If a function changes the value of any of its input arguments, the change is *not reflected* in the actual input argument on return to the workspace (unless the function is called with the same input and output argument—see below). For the technically minded, input arguments appear to be passed *by value*.
  You might think that passing a large matrix as an input argument by value is wasteful of memory, and you would be correct. However, the designers of MATLAB were aware of this, and so an input argument is only passed by value if a function modifies it (although the modification is not reflected on return). If a function does not modify an input argument it is passed by reference.
- **Simulated pass by reference**
  A function may be called with the same actual input and output argument. For example, the following function `prune.m` removes all the zero elements from its input argument:

```
function y=prune(x)
y=x(x ~= 0);
```

  (if you can't figure out why, refer to **Subscripting with logical vectors** in Chapter 5).
  You can use it to remove all the zero elements of the vector `x` as follows:

```
x=prune(x)
```

- **Checking the number of function arguments**
  A function may be called with all, some, or none of its input arguments. If called with no arguments, the parentheses must be omitted. You may not use more input arguments than appear in its definition.
  The same applies to output arguments—you may specify all, some, or none of them when you use the function. If you call a function with no output arguments, the value of the first one in the definition is returned.
  There are times when a function may need to know how many input/output arguments are used on a particular call. In that case, the functions `nargin` and `nargout` can be used to determine the number of actual input and output arguments. For example:

```
function y=foo(a, b, c);
disp( nargin );
...
```

will display the number of input arguments present on each call of `foo`.

- **Passing a variable number of arguments**

  The functions `varargin` and `varargout` allow you to call a function with any number of input or output arguments. Since this facility involves packing arguments into a *cell array*, discussion of it is deferred to Chapter 10.

7.2.1 Subfunctions

A function M-file may contain the code for more than one function. The first function in a file is the *primary* function, and is the one invoked with the M-file name. Additional functions in the file are called *subfunctions*, and are visible only to the primary function and to other subfunctions.

Each subfunction begins with its own function definition line. Subfunctions follow each other in any order *after* the primary function.

7.2.2 Private functions

A private function is a function residing in a subdirectory with the name `private`. Private functions are visible only to functions in the parent directory. See Help for more details.

7.2.3 P-code files

The first time a function is called during a MATLAB session it is parsed ("compiled") into pseudocode and stored in memory to obviate the need for it to be parsed again during the current session. The pseudocode remains in memory until you clear it with `clear` *function_name* (see Help for all the possibilities with `clear`).

You can use the `pcode` function to save the parsed version of an M-file for use in later MATLAB sessions, or by users from whom you want to hide your algorithms. For example, the command:

```
pcode stats
```

parses `stats.m` and stores the resulting pseudocode in the file named `stats.p`.

MATLAB is very efficient at parsing so producing your own P-code files seldom makes much of a speed difference, except in the case of large GUI applications where many M-files have to be parsed before the application surfaces.

7.2.4 Improving M-file performance with the profiler

The MATLAB Profiler enables you to see where the bottlenecks in your programs are, e.g., which functions are consuming most of the time. With this information you can often redesign programs to be more efficient. To find out more about this utility open the **MATLAB Help** documentation via the "?" at the top of the desktop and type "Profiler" in the search space.

7.3 FUNCTION HANDLES

Our script file `newtgen.m` in Section 7.1 solves the equation $f(x)=0$, where $f(x)$ is defined in the function file with the specific name `f.m`. This is restrictive because, to solve a different equation, `f.m` has to be edited first. To make `newtgen` even more general, it can be rewritten as a function M-file itself, with a *handle* to `f.m` as an input argument. This process is made possible by the built-in function `feval`, and the concept of a function handle, which we now examine.

Try the following on the command line:

```
fhandle = @sqrt;
feval(fhandle, 9)
feval(fhandle, 25)
```

Can you see that `feval(fhandle, x)` is the same as `sqrt(x)`? The statement:

```
fhandle = @sqrt
```

creates a handle to the function `sqrt`. The handle provides a way of referring to the function, for example, in a list of input arguments to another function. A MATLAB function handle is similar to a pointer in C++, although more general.

If you have still got a function file `f.m` defined for $f(x)=x^3+x-3$, verify that:

```
feval(@f, 2)
```

for example, returns the same value as `f(2)`.

- In general, the first argument of `feval` is a handle to the function to be evaluated in terms of the subsequent arguments of `feval`.

You can use `feval` inside a function to evaluate another function whose handle is passed as an argument, as we will see now. As an example, we would like to rewrite our `newtgen` script as a function `newtfun`, to be called as follows:

```
[x f conv] = newtfun( fh, dfh, x0 )
```

where `fh` and `dfh` are handles for the M-files containing $f(x)$ and $f'(x)$ respectively, and `x0` is the initial guess. The outputs are the zero, the function value

at the zero, and an argument conv to indicate whether or not the process has converged. The complete M-file newtfun.m is as follows:

```
function [x, f, conv] = newtfun(fh, dfh, x0)
% NEWTON      Uses Newton's method to solve f(x) = 0.
%             fh is handle to f(x), dfh is handle to f'(x).
%             Initial guess is x0.
%             Returns final value of x, f(x), and
%             conv (1 = convergence, 0 = divergence)

steps = 0;                          % iteration counter
x = x0;
re = 1e-8;                          % required relative error
myrel = 1;

while myrel > re & (steps < 20)
    xold = x;
    x = x - feval(fh, x)/feval(dfh, x);
    steps = steps + 1;
    disp( [x feval(fh, x)] )
    myrel = abs((x-xold)/x);
end;

if myrel <= re
    conv = 1;
else
    conv = 0;
end;

f = feval(fh, x);
```

Try it out. Verify that you can call newtfun with less than three output variables. Also check help newton.

A function handle gives you more than just a reference to the function. Open the help document and search for "function handles" to find more information on this topic.

Functions, such as feval, fplot, newtfun, etc. which take function handles as arguments are referred to by MATLAB as *function functions*, as opposed to functions which take numeric arrays as arguments.

Use of a function handle to evaluate a function supersedes the earlier use of feval where a string containing the function's name was passed as an argument.

7.4 COMMAND/FUNCTION DUALITY

In the earliest versions of MATLAB there was a clear distinction between *commands* like:

```
clear
save junk x y z
whos
```

and *functions* like

```
sin(x)
plot(x, y)
```

If commands had any arguments they had to be separated by blanks with no brackets. Commands altered the environment, but didn't return results. New commands could not be created with M-files.

From Version 4 onwards commands and functions are "duals," in that commands are considered to be functions taking string arguments. So:

```
axis off
```

is the same as,

```
axis('off' )
```

Other examples are <

```
disp Error!
hold('on')
```

This duality makes it possible to generate command arguments with string manipulations, and also to create new commands with M-files.

7.5 FUNCTION NAME RESOLUTION

Remember that a variable in the workspace can "hide" a built-in function of the same name, and a built-in function can hide an M-file.

Specifically, when MATLAB encounters a name it resolves it in the following steps:

1. Checks if the name is a variable.
2. Checks if the name is a *subfunction* of the calling function.
3. Checks if the name is a *private function*.
4. Checks if the name is in the directories specified by MATLAB's search path.

MATLAB therefore always tries to use a name as a variable first, before trying to use it as a script or function.

7.6 DEBUGGING M-FILES

Run-time errors (as opposed to syntax errors) which occur inside function M-files are often hard to fix, because the function workspace is lost when the error forces a return to the base workspace. The Editor/Debugger enables you to get inside a function while it is running, to see what's going wrong.

7.6.1 Debugging a script

To see how to debug interactively, let's first try the script `newtgen.m` in Section 7.1. Go through the following steps:

- Open `newtgen.m` with the MATLAB Editor/Debugger. (Incidentally, have you found out that you can run the Editor directly from the command line, e.g., with `edit newtgen`)?

You will have noticed that the lines in the Editor window are numbered. You can generate these line numbers from the command line for reference purposes with the command `dbtype`:

```
dbtype newtgen

1     % Newton's method in general
2     % exclude zero roots!
3
4     steps = 0;                          % iteration counter
5     x = input( 'Initial guess: ' );     % estimate of root
6     re = 1e-8;                          % required relative error
7     myrel = 1;
8
9     while myrel > re & (steps < 20)
      xold = x;
11      x = x - f(x)/df(x);
12      steps = steps + 1;
13      disp( [x f(x)] )
14      myrel = abs((x-xold)/x);
15    end;
16
17    if myrel <= re
18      disp( 'Zero found at' )
19      disp( x )
20    else
21      disp( 'Zero NOT found' )
22    end;
```

- To get into Debug mode you need to set a *breakpoint* just before where you think the problem might be. Alternatively, if you just want to "step through" a script line by line, set a breakpoint at the first executable

statement. The column to the right of the line numbers is called the *breakpoint alley*. You can only set breakpoints at executable statements—these are indicated by dashes in the breakpoint alley.
Set a breakpoint at line 4 (`steps=0;`) by clicking in the breakpoint alley. You can remove a breakpoint by clicking on the breakpoint icon, or using the Editor's **Breakpoints** menu (this menu also allows you to specify stopping conditions). You can also set/clear breakpoints on the current line with the set/clear breakpoint button on the toolbar.

- Having set your breakpoints, run the script in the Editor by clicking the run button in the toolbar, or with **Debug -> Run (F5)**.
  You can also run the script from the command line.
- When the script starts to run, there are two things in particular to notice. First, the symbol `K` appears to left of the command-line prompt to remind you that MATLAB is in debug mode.
  Second, a green arrow appears just to the right of the breakpoint in the Editor. The arrow indicates the next statement which is *about to be* executed.
- Now step through the script with **Debug -> Step (F10)**. Note when line 5 is executed you need to enter the value of `x` at the command line.
- When you get to line 11 (`x=x - f(x)/df(x);`) in this way, use **Debug -> Step In (F11)** and the Debugger will take you into the functions `f.m` and `df.m`.
- Continue with **F10**. Note that output appears in the Command Window as each `disp` statement is executed.
- There are a number of ways of examining variable values in debug mode:
  1. Position the cursor to the left of the variable in the Editor. Its current value appears in a box—this is called a datatip.
  2. Type the name of the variable in the Command Window.
  3. Use the Array Editor: open the Workspace browser and double-click a variable. If you arrange your windows carefully you can watch the value of a variable change in the Array Editor while you step through a program.
     Note that you can view variables only in the current workspace. The Editor has a **Stack** field to the right of the toolbar where you can select the workspace. For example, if you have stepped into `f.m` the current workspace is shown as `f`. At this point you can view variables in the `newtgen` workspace by selecting `newtgen` in the **Stack** field.You can use the Array Editor or the command line to change the value of a variable.

  You can then continue to see how the script performs with the new value.
- Another useful debugging feature is **Debug -> Go Until Cursor**. This enables you to continue running the script to the line where you've positioned the cursor.

- To quit debugging click the exit debug mode button in the Editor/Debugger toolbar, or select **Debug -> Exit Debug Mode**.

  If you forget to quit debugging you won't be able to get rid of the `K` prompt on the command line!

7.6.2 Debugging a function

You can't run a function directly in the Editor/Debugger—you have to set a breakpoint in the function and run it from the command line. Let's use `newtfun.m` as an example.

- Open `newtfun.m` in the Editor/Debugger.
- Set a breakpoint at line 8 (`steps=0;`).
- In the Command Window set up function handles for `f` and `df` and call `newtfun`:

```
fhand = @f;
dfhand = @df;
[x f conv] = newtfun(fhand, dfhand, 10)
```

- Note that MATLAB goes into debug mode and takes you to the breakpoint in `newtfun`. Now you can continue debugging as before.

  Debugging may also be done from the command line with the debug functions. See `help debug`.

7.7 RECURSION

Many (mathematical) functions are defined *recursively*, i.e., in terms of simpler cases of themselves, e.g., the factorial function may be defined recursively as:

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

as long as $1!$ is defined as 1. MATLAB allows functions to call themselves; this process is called *recursion*. The factorial function may be written recursively in an M-file `fact.m` like this:

```
function y = fact(n)
% FACT       Recursive definition of n!

if n > 1
  y = n * fact(n-1);
else
  y = 1;
end;
```

Recursive functions are usually written in this way: An `if` statement handles the general recursive definition; the `else` part handles the special case ($n = 1$).

Although recursion appears deceptively simple, it is an advanced topic, as the following experiment demonstrates. Insert the statement `disp(n)` into the definition of `fact` immediately *above* the `if` statement, and run `fact(5)` from the command line. The effect is what you might expect: the integers 5–1 in descending order. Now move `disp(n)` to *below* the `if` statement, and see what happens. The result is the integers 1–5 in *ascending* order now, which is rather surprising.

In the first case, the value of `n` is displayed each time `fact` is called, and the output is obvious enough. However, there is the world of difference between a recursive function being *called*, and *executed*. In the second case, the `disp` statement is only executed after the `if` has finished executing. And when is that exactly? Well, when the initial call to `fact` takes place, `n` has the value 5, so the first statement in the `if` is invoked. However, the value of `fact(4)` is not known at this stage, so a *copy* is made of all the statements in the function which will need to be executed once the value of `fact(4)` is known. The reference to `fact(4)` makes `fact` call itself, this time with a value of 4 for `n`. Again, the first statement in the `if` is invoked, and MATLAB discovers that it doesn't know the value of `fact(3)` this time. So another (different) copy is made of all the statements that will have to be executed once the value of `fact(3)` is known. And so each time `fact` is called, separate copies are made of all the statements *yet to be executed*. Finally, MATLAB joyfully finds a value of `n` (1) for which it actually knows the value of `fact`, so it can at last begin to execute (in reverse order) the pile of statements which have been damming up inside the memory.

This discussion illustrates the point that recursion should be treated with respect. While it is perfectly in order to use it in a example like this, it can chew up huge amounts of computer memory and time.

SUMMARY

- Good structured programming requires real problem-solving programs to be broken down into function M-files.
- The name of a function in the function definition line should be the same as the name of the M-file under which it is saved. The M-file must have the extension `.m`.
- A function may have input and output arguments, which are usually its only way of communicating with the workspace. Input/output arguments are dummy variables (placeholders).

- Comment lines up to the first non-comment line in a function are displayed when `help` is requested for the function.
- Variables defined inside a function are local variables and are inaccessible outside the function.
- Variables in the workspace are inaccessible inside a function unless they have been declared `global`.
- A function does not have to have any output arguments.
- Input arguments have the appearance of being passed by value to a function. This means that changes made to an input argument inside a function are not reflected in the actual input argument when the function returns.
- A function may be called with fewer than its full number of input/output arguments.
- The functions `nargin` and `nargout` indicate how many input and output arguments are used on a particular function call.
- Variables declared `persistent` inside a function retain their values between calls to the function.
- Subfunctions in an M-file are accessible only to the primary function and to other subfunctions in the same M-file.
- Private functions are functions residing in a sub-directory named `private` and are accessible only to functions in the parent directory.
- Functions may be parsed (compiled) with the `pcode` function. The resulting code has the extension `.p` and is called a P-code file.
- The Profiler enables you to find out where your programs spend most of their time.
- A handle for a function is created with `@`. A function may be represented by its handle. In particular the handle may be passed as an argument to another function.
- `feval` evaluates a function whose handle is passed to it as an argument.
- MATLAB first tries to use a name as a variable, then as a built-in function, and finally as one of the various types of function.
- Command/function duality means that new commands can be created with function M-files, and that command arguments may be generated with string manipulations.
- The Editor/Debugger enables you to work through a script or function line-by-line in debug mode, examining and changing variables on the way.
- A function may call itself. This feature is called recursion.

CHAPTER EXERCISES

7.1 Change the function `stars` of Section 7.2 to a function `pretty` so that it will draw a line of any specified character. The character to be used must be passed as an additional input (string) argument, e.g., `pretty(6, '$')` should draw six dollar symbols.

7.2 Write a script `newquot.m` which uses the Newton quotient $[f(x+h)-f(x)]/h$ to estimate the first derivative of $f(x)=x^3$ at $x=1$, using successively smaller values of h: 1, 10^{-1}, 10^{-2}, and so on. Use a function M-file for $f(x)$.
Rewrite `newquot` as a function M-file able to take a handle for $f(x)$ as an input argument.

7.3 Write and test a function `double(x)` which doubles its input argument, i.e., the statement `x=double(x)` should double the value in `x`.

7.4 Write and test a function `swop(x, y)` which will exchange the values of its two input arguments.

7.5 Write your own MATLAB function to compute the exponential function directly from the Taylor series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

The series should end when the last term is less than 10^{-6}. Test your function against the built-in function `exp`, but be careful not to make x too large—this could cause a rounding error.

7.6 If a random variable X is distributed normally with zero mean and unit standard deviation, the probability that $0 \leq X \leq x$ is given by the standard normal function $\Phi(x)$. This is usually looked up in tables, but it may be approximated as follows:

$$\Phi(x) = 0.5 - r(at + bt^2 + ct^3),$$

where $a=0.4361836$, $b=-0.1201676$, $c=0.937298$, $r = \exp(-0.5x^2)/\sqrt{2\pi}$, and $t=1/(1+0.3326x)$.
Write a function to compute $\Phi(x)$, and use it in a program to write out its values for $0 \leqslant x \leqslant 4$ in steps of 0.1. Check: $\Phi(1)=0.3413$.

7.7 Write a function:

```
function [x1, x2, flag] = quad( a, b, c )
```

which computes the roots of the quadratic equation $ax^2+bx+c=0$. The input arguments `a`, `b` and `c` (which may take any values) are the coefficients of the quadratic, and `x1`, `x2` are the two roots (if they exist), which may be equal. See Figure 3.3 in Chapter 3 for the structure plan.

The output argument `flag` must return the following values, according to the number and type of roots:

0:	no solution ($a=b=0$, $c\neq 0$);
1:	one real root ($a=0$, $b\neq 0$, so the root is $-c/b$);
2:	two real or complex roots (which could be equal if they are real);
99:	any x is a solution ($a=b=c=0$).

Test your function on the data in Exercise 3.5.

7.8 The Fibonacci numbers are generated by the sequence:

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

Can you work out what the next term is? Write a recursive function `f(n)` to compute the Fibonacci numbers F_0 to F_{20}, using the relationship:

$$F_n = F_{n-1} + F_{n-2},$$

given that $F_0 = F_1 = 1$.

7.9 The first three Legendre polynomials are $P_0(x) = 1$, $P_1(x) = x$, and $P_2(x) = (3x^2-1)/2$. There is a general *recurrence* formula for Legendre polynomials, by which they are defined recursively:

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0.$$

Define a recursive function `p(n,x)` to generate Legendre polynomials, given the form of P_0 and P_1. Use your function to compute `p(2,x)` for a few values of `x`, and compare your results with those using the analytic form of $P_2(x)$ given above.