Chapter 3

Scripts and Functions

An **M-file** is a plain text file containing MATLAB commands and saved with the filename extension .m. There are two types, **scripts** and **functions**. MATLAB comes with a good editor that is tightly integrated into the environment; start it by typing `edit`. However, you are free to use any text editor. An M-file should be saved in the path in order to be executed. Recall that the path is a list of directories (folders) in which MATLAB will look for files. Use `editpath` or menus to see and change the path.

There is no need to compile either type of M-file. Simply type in the name of the file (without the extension) in order to run it. Changes that are saved to disk will be included in the next call to the function or script.

An important type of statement in any M-file is a **comment**, which is indicated by a percent sign `%`. Any text on the same line after a percent sign is ignored, unless `%` appears as part of a string. Furthermore, the first contiguous block of comments in an M-file serves as documentation for the file and will be typed out in the command window if `help` is used on the file. For instance, say the following is saved as myscript.m on the path:

```
% This script solves the nasty homework problem
% assigned by Professor Driscoll.

x = rand(1);  % He'll never notice.
```

Then at the prompt one would could obtain

```
>> help myscript

  This script solves the nasty homework problem
  assigned by Professor Driscoll.
```

Normally, you will want to include syntax information as well as a short description within the documentation of your own functions.

3.1 Using scripts effectively

The contents of a script file are literally interpreted as though they were typed at the prompt. In fact, you might prefer to use MATLAB exclusively by typing all commands into scripts and running them. This technique makes it easier to create and revise a sequence of more than a few lines, and it helps you document and retrace your steps later. Note too that you can create a script from lines you have already entered by highlighting them in the Command History window of the desktop and right-clicking on them.

Another interesting use for scripts is to publish them as HTML files (Web pages) or in other available formats. This is a nice way to create reports based on MATLAB results, although support for mathematical expressions is currently rather limited. When a script is published, blocks of commented lines appear as regular text, and code and resulting output are set off in nice formats. To get the best results, use double percent signs at the beginning of a line to create a *cell boundary*. These boundaries cause output to be created before the next block of comments, code, and output begins.

For example, the script

```
%%
% A Hilbert matrix has entries that are reciprocals of
% integers.
format rat
H = hilb(4)

%%
% Although the inverse of a Hilbert matrix is difficult
% to find numerically by Gaussian elimination, an explicit
% formula is known.
invhilb(4)

%%
format short e
H*ans
```

will publish to look something like this:

A Hilbert matrix has entries that are reciprocals of integers.

```
format rat
H = hilb(4)

H =

       1             1/2           1/3           1/4
       1/2           1/3           1/4           1/5
       1/3           1/4           1/5           1/6
       1/4           1/5           1/6           1/7
```

Although the inverse of a Hilbert matrix is difficult to find numerically by Gaussian elimination, an explicit formula is known.

```
invhilb(4)


ans =

      16         -120          240         -140
    -120         1200        -2700         1680
     240        -2700         6480        -4200
    -140         1680        -4200         2800



format short e
H*ans


ans =

            0            0            0            0
            0            0            0            0
            0            0            0 -5.6843e-014
            0            0            0            0
```

In addition, cells allow you to create titled sections, complete with a hyperlinked table of contents. The built-in Editor is the best way to publish scripts. It will help you to create and work with cells and to clearly show cell boundaries and titles, and it lets you publish and view results with minimal fuss.

As a rule of thumb, call scripts only from the command line, and do not call other scripts from within a script. For automating individual tasks that fit within a larger framework, functions are the better choice, as explained in the next section.

3.2 Functions and workspaces

Both functions and scripts gather MATLAB statements to perform complex tasks. The most important distinguishing feature of a function is its **local workspace**. Any variables created while the function executes are available only within that invocation of the function, unless you go out of your way to bend the rules (see section 6.4). Conversely, the variables available to the command line—those in the **base workspace**—are normally not visible within the function. If other functions are called during a function's execution, each of those secondary calls also sets up a private local workspace. These restrictions on data access are called **scoping**, and they make it possible for you to write complex programs that use many components without worrying about variable name clashes. You can always see variables in the current workspace by typing `whos`, or in the Workspace window of the desktop. At the command line, the base workspace is ordinarily in context, but during function debugging you can inspect other local workspaces.

Each function starts with a line such as

```
function [out1,out2] = myfun(in1,in2,in3)
```

The name `myfun` should match the name of the file on disk. The variables `in1`, etc. are **input arguments**, and `out1`, etc. are **output arguments**. You can have as many as you like of each type (including zero) and call them whatever you want. Theoretically, the only communication between a function's workspace and that of its caller is through the input and output arguments, though there are some exceptions (see section 6.4). The values of the input arguments to a function are *copies* of the original data, so any changes you make to them will not change anything outside of the function's scope.[5]

Here is a function that implements the quadratic formula for finding the roots of $ax^2 + bx + c$:

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

d = sqrt(b^2 - 4*a*c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
```

(It must be pointed out that from a numerical standpoint, this is not a good algorithm; see Exercise 3.1.) With this text saved as quadform.m in the MATLAB path, you could then immediately enter

```
>> [r1,r2] = quadform(1,1,1)

r1 =
  -0.5000 + 0.8660i

r2 =
  -0.5000 - 0.8660i
```

The main use of a function is to compartmentalize a specific task. Any complex problem is decomposed into a series of smaller steps, and the scoping rules of functions allow you to deal with each step independently. They also let you exploit well-crafted solutions to fundamental tasks that appear over and over in different problems. Like a good theorem, a good function invites you to inspect the details of its construction once and then forget about them.

[5] Actually, the MATLAB interpreter avoids copying a function argument—that is, it "passes by reference" using a pointer—if the function never alters the value of that argument. Furthermore, in MATLAB 7.6 and later versions, you can force passing by reference even for modified values; see the help on `handle` for details.

Another important aspect of function M-files is that many of the functions that come with MATLAB are themselves M-files that you can read and borrow from. This is an excellent way to learn good programming practice—and dirty tricks.

3.3 Conditionals: `if` and `switch`

Often, a function needs to branch based on run-time conditions. MATLAB offers features for this that are similar to those in most languages.

Here is an example illustrating most of the features of `if`.

```
if isinf(x) || ~isreal(x)
  disp('Bad input!')
  y = NaN;
elseif (x == round(x)) && (x > 0)
  y = prod(1:x-1);
else
  y = gamma(x);
end
```

The conditions for `if` statements may involve the relational operators of Table 2.3 on page 17, or functions such as `isinf` that return logical values. Numerical values can also be used, with nonzero meaning true, but "`if x~=0`" is better form than "`if x`" when `x` is numeric. Take some care when using arrays to construct the condition of an `if` statement. When the condition is not a scalar, it is taken as true only when *all* of the elements are true/nonzero. To avoid confusion, it's best to use `any` or `all` to reduce logical arrays to scalar values.

Compound scalar conditions should be created using the operators `&&` (logical AND) and `||` (logical OR), rather than the array operators `&` and `|`. (The NOT operator `~` is the same in both cases.) The double-symbol operators can be **short-circuited**: if, as a condition is evaluated from left to right, it becomes obvious before the end that truth or falsity is assured, then evaluation of the condition is halted. This feature makes it convenient to write things like

```
if (length(x) > 2) && (x(3)==0), disp('xy plane'), end
```

that otherwise could create errors or be awkward to write.

In some situations, the `isequal` command is more robust than the `==` relational operator. For instance, the syntax `isequal(s,'foo')` returns false if `s` is not identical to the string `'foo'`, whereas `s=='foo'` would throw an error if `s` were not of size 1×3. Usually, the former behavior is more desirable.

The `if`/`elseif` construct is preferred when only a small number of alternatives are present. When a large number of options are needed, it's customary to use instead. For instance:

```
switch units
  case 'length'
    disp('meters')
  case 'volume'
    disp('liters')
  case 'time'
    disp('seconds')
  otherwise
    disp('I give up')
end
```

The switch expression can be a string or a number. The first matching `case` has its commands executed.[6] If `otherwise` is present, it gives a default option if no case matches.`switch`

3.4 Loops: `for` and `while`

Many algorithms require **iteration**, the repetitive execution of a block of statements. Again, MATLAB is similar to other languages in this area. It's worth repeating here that because `i` and `j` are very tempting as loop indices, you are advised to always use `1i` or `1j` as the imaginary unit $\sqrt{-1}$.

As an illustration of a simple `for` loop, consider this code for computing 10 members of the famous Fibonacci sequence:

```
>> f = [1 1];
>> for n = 3:10
     f(n) = f(n-1) + f(n-2);
   end
```

You can have as many statements as you like in the body of a loop. In this example, the value of the index `n` will change from 3 to 10, with an execution of the body after each assignment. Remember that `3:10` is really just a row vector. In fact, you can use *any* row vector in a `for` loop, not just one created by a colon. For example,

```
>> x = 1:100; s = 0;
>> for j = find(isprime(x))
     s = s + x(j);
   end
```

This finds the sum of all primes less than 100. For a better version, though, see page 64.

[6] Execution does not "fall through" as in C.

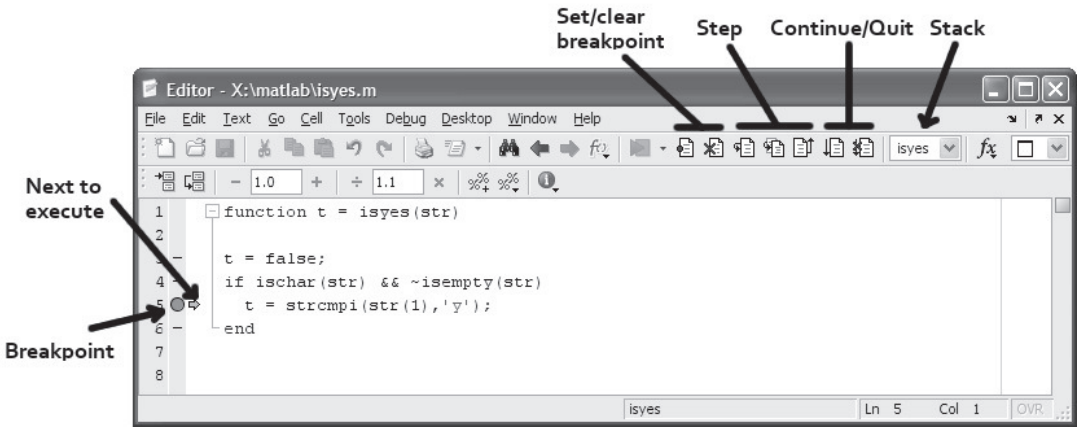

Figure 3.1. *Debugging tools in the MATLAB Editor.*

It is sometimes necessary to repeat statements based on satisfying a condition rather than a fixed number of times. This is done with `while`:

```
while abs(x) > 1
  x = x/2;
end
```

The condition is evaluated before the body is executed, so it is possible to get zero iterations. It's often a good idea to limit the number of repetitions to avoid infinite loops (as could happen above if `x` is infinite). This can be done in a number of ways, but the most common is to use `break`.

```
n = 0;
while abs(x) > 1
  x = x/2;
  n = n+1;
  if n > 50, break, end
end
```

A `break` immediately jumps execution to the first statement after the loop. It's good practice to include some diagnostic output or other indication that an abnormal loop exit is taking place.

3.5 Debugging and profiling

MATLAB provides a debugging mode that lets you pause execution anywhere inside an M-file function. It's a great way to fix faulty code or to understand how a code works. A screenshot of the debugging tools is shown in Figure 3.1.

To enter debugging mode, you set **breakpoints** in one or more functions. In the Editor, you click on the dash next to a line number to do this, or you can use the Debug menu to create conditional breakpoints. When MATLAB stops at

a breakpoint, you can inspect and modify all the variables currently in scope—in fact, you can execute anything at all from the command line. You can then continue function execution normally or step by step. See `help debug` for all the details.

Another common need is to make working code run faster. The most efficient way to do so is to locate which parts are taking the most time to execute. These lines are then obvious candidates for optimization. You can find such lines by **profiling**, which keeps track of time spent on every line of every function. Profiling is also one way to determine function dependencies (who calls whom). Get started by typing `profile viewer`.

Exercises

3.1. Write a function `quadform2` that implements the quadratic formula differently from `quadform` above (page 30). First compute

$$x_1 = \frac{-b - \text{sign}(b)\sqrt{b^2 - 4ac}}{2a},$$

which is the root of largest magnitude, and then use the identity $x_1 x_2 = c/a$ to find x_2. Apply both `quadform` and `quadform2` to find the roots of $x^2 - (10^7 + 10^{-7})x + 1$. Do you see why `quadform2` is better?

3.2. The degree-n Chebyshev polynomial is defined by

$$T_n(x) = \cos\bigl[n \cos^{-1}(x)\bigr], \quad -1 \le x \le 1.$$

These satisfy $T_0(x) = 1$, $T_1(x) = x$, and the recursion relation

$$T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x), \quad n \ge 1.$$

Write a function `chebeval(x,N)` that evaluates all of the Chebyshev polynomials of degree less than or equal to `N` at all of the points in column vector `x`. The result should be an array of size `length(x)` by `N+1`.

3.3. One way to compute the exponential function e^x is to truncate its Taylor series expansion around $x = 0$,

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

Unfortunately, many terms are required for accuracy if $|x|$ is large. But a special property of the exponential is that $e^{2x} = (e^x)^2$. This leads to a *scaling and squaring* method: Divide x by 2 repeatedly until $|x| < 1/2$, use the Taylor series (16 terms should be more than enough), and square the result repeatedly. Write a function `expss(x)` that performs these three steps. (The functions `cumprod` and `polyval` can help with evaluating the Taylor expansion.) Test your function on x values -30, -3, 3, 30.

3.4. Let `x` and `y` be column vectors describing the vertices of a polygon, given in order. Write functions `polyperim(x,y)` and `polyarea(x,y)` that compute the perimeter and area of the polygon. For the area, use a formula based on Green's theorem:

$$A = \frac{1}{2}\left|\sum_{k=1}^{n} x_k y_{k+1} - x_{k+1} y_k\right|.$$

Here n is the number of polygon vertices, and by definition, $x_{n+1} = x_1$ and $y_{n+1} = y_1$. Test your functions on a square and an equilateral triangle.

3.5. Suppose a data source produces a series of characters drawn from a set of M distinct symbols. If symbol k is produced with probability p_k, the *first-order entropy* of the source is defined as

$$H_1 = -\sum_{k=1}^{M} p_k \log_2 p_k.$$

Essentially H_1 is the number of bits needed per symbol to encode a long message; that is, it measures the amount of information content, and therefore the potential success of compression strategies. The value $H_1 = 0$ corresponds to the case of only one symbol being produced—no information—while if all M symbols have equal probability, then $H_1 = \log_2 M$.

Write a function `[H,M] = entropy(v)` that computes entropy for a vector `v`. The probabilities should be computed empirically by finding the unique entries (using `unique`), then counting the occurrences of each symbol and dividing by the length of `v`. Try your function on some built-in image data by entering `load clown, v = X(:);`.