Chapter 2

Functions

2.1 What is a function?

In the last chapter we learned the basics of how to use MATLAB as a calculator. In this chapter we'll learn how to extend MATLAB by writing our own functions that we can call. We've already seen how useful the built in functions such as `std`, `sin` and `hist` can be. By the end of this chapter you'll be able to write your own functions that are every bit as useful.

A function in MATLAB is a reusable piece of code that:

- Is configured using some *parameters* that specify what the function should do.
- Performs zero or more actions, such as printing text or plotting graphs.
- Computes the *return values* for the function.

For example in the MATLAB expression `sin(x)`, the function named `sin` takes a single parameter x, performs no actions and returns the sin of x computed assuming x is in radians.

The expression `hist(v,20)` takes two parameters, the first is the vector of data to plot and the second is the number of bars to show in the histogram. It performs one action, generating the histogram. Although one usually ignores its return value, it does actually compute an array of values indicating the size of each bar.

Here are some functions that we will write this chapter[1]:

(i) `cumulativeNormal(x)`. This will compute the cumulative distribution function for the normal distribution. This takes a single parameter x, performs no action and returns the value of $\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2)\,\mathrm{d}t$.

[1] In actual fact MATLAB already contains built in functions that perform pretty similar tasks to the functions we are going to write in this chapter. For the purpose of this chapter we'll pretend that it does not contain these functions so that we can work out how we would write them ourselves.

(ii) `blackScholesCallPrice( K, T, S, vol, r)`. This takes five parameters, the strike price, the time to maturity, the current stock price, the volatility and the risk free interest rate. It returns the current value of a call option given the parameters.

(iii) `integrateNumerically( f, a, b, N)`. This takes four parameters and returns an approximation to the $\int_a^b f(t)\,\mathrm{d}t$ computed using the rectangle method with N steps.

2.2 Writing a function

2.2.1 Motivation

Example 1: Make the substitution $t = x + 1 - \frac{1}{s}$ to transform the integral

$$\int_{-\infty}^{x} \exp(-t^2/2)\,\mathrm{d}t$$

to an integral of a finite interval. Hence use MATLAB to approximate the value of $\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2)\,\mathrm{d}t$ using the rectangle method when $x = 1.5$.

Solution: Making the suggested substitution we find:

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2)\,\mathrm{d}t = \frac{1}{\sqrt{2\pi}} \int_0^1 \frac{1}{s^2} \exp(-1/2(x + 1 - \frac{1}{s})^2)\,\mathrm{d}s$$

Let us recall the rectangle method of integration. Suppose $f : [a, b] \longrightarrow \mathbb{R}$ is a function we wish to integrate. We wish to approximate the area under f with N rectangles. Define $h = (b - a)/N$ and define $s_n = a + (n - 1/2)h$. Then the rectangle rule approximation for $\int_a^b f(s)\,\mathrm{d}s$ is given by

$$\int_a^b f(s)\,\mathrm{d}s \approx h \sum_1^N f(s_n).$$

See Figure 2.1.

For our problem, let us choose a large value of N, say 1000. Let us take $a = 0$ and $b = 1$ and define

$$f(s) = \frac{1}{s^2} \exp(-(x + 1 - \frac{1}{s})^2/2)$$

we can the compute the desired value using the following MATLAB code

```
x = 1.5;
a = 0;
```

```
b = 1;
N = 1000;
h = (b-a)/N;
s =  a + (1:N - 0.5) * h;
fValues = s.^(-2) .* exp( -(( x + 1 - 1./s).^2)/2 );
integral = h * sum( fValues );
result = 1 / sqrt( 2 * pi ) * integral
```

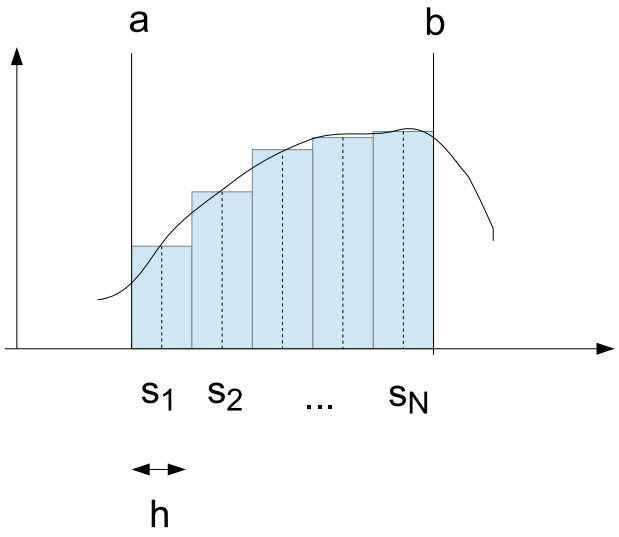

Figure 2.1: The Rectangle Rule

The result will be approximately 0.9331, which is accurate three significant figures.

This is pretty sophisticated code. We certainly wouldn't want to retype all of this code every time we wanted to change the value of x and rerun the computation. We will want to save this code to a file and we will want to find some way to reuse the code whenever we need to compute the cumulative distribution function of the normal distribution. Creating a function called `cumulativeNormal` is the perfect way to achieve both of these goals. We will describe how to do this over the next few sections.

As you will see, there is another advantage to using functions. The code given above is very hard to follow. This is because it does too much at once. Using functions we can break it into smaller components which will be easier to understand. Just as you break a complex theorem into simple lemma's, so you should break a complex function into simple functions.

2.2.2 Choosing the current folder

When you create a function you will usually want to keep the code you have written so you can call the function whenever you want. So the first step is to create a directory to store all your code in. Do the following using Windows Explorer. **THIS IS A DIFFERENT PROGRAM TO INTERNET EXPLORER! You can start it by pressing the Windows key and E**.

This is the standard program on Windows to manage your files. Use Windows Explorer to complete the following tasks:

- In your home area create a folder called `FM06`.
- In that folder create a sub-folder called `Lecture1`.

I recommend that each week you create a new folder inside the FM06 folder to hold the code you write for that weeks exercises. The folder that you are using to store your work is called the **Current Folder** in MATLAB and you need to tell MATLAB where it is located.

To tell MATLAB where you have put your **Current Folder**, return to MATLAB and take a look at the toolbar or ribbon at the top of the screen. You should see that there is box giving the name of a directory.

- On MATLAB 2009 this box is marked Current Folder. There is a button to the right of it marked mysteriously with just three dots. Click this to and browse to the `Lecture1` folder.
- On MATLAB 2013 this box is at the bottom of the "ribbon" containing all the buttons and isn't actually labelled itself. You will see a button to the left of the box showing a picture of a folder with a green arrow on it. Click on this button and browse to the `Lecture1` folder.

2.2.3 Creating and running a function

Once you have selected the Current Folder to be the folder where you wish to create your function you can create a file to hold your function.

Right click in the middle of the area of the screen on the left labelled Current Folder and select **New File→Function**.

Type in the name of the function in the edit box that appears in the Current Folder window. The correct text is `cumulativeNormal.m`. This is the name of the function we are going to create with the additional suffix `.m`. So we are creating a function called simply `cumulativeNormal`. Note that the name should be like a variable name — no spaces, long and self-explanatory, and using camel case to make different words distinct.

By typing in the name `cumulativeNormal.m` we are actually naming the file that will contain our function definition. The `.m` suffix simply marks the file as being a MATLAB file.

Once you have created the file and chosen its name, double click on the name of the file in the **Current Folder** window. This will open up a text editor where you can type in the MATLAB code for your function.

If you have followed my instructions to the letter the outline code that MATLAB has created should look like this:

```
function [ output_args ]=cumulativeNormal( input_args )
%CUMULATIVENORMAL Summary of this function goes here
%   Detailed explanation goes here


end
```

This is meant to be a helpful reminder to you of how you write functions in MATLAB, but until you know how to write a function in the first place it probably looks more confusing than helpful.

Simply replace it all with the code below:

```
function [ result ] = cumulativeNormal( x )
%CUMULATIVENORMAL computes c.d.f of normal distribution
a = 0;
b = 1;
N = 1000;
h = (b-a)/N;
s =  a + (1:N - 0.5) * h;
fValues = s.^(-2) .* exp( -(( x + 1 - 1./s).^2)/2 );
integral = h * sum( fValues );
result = 1 / sqrt( 2 * pi ) * integral;
end
```

You only need to copy the text. MATLAB provides the colour automatically. When you have replaced the code, save the file.

You can now test the code by executing the following command in the **Command Window**.

```
cumulativeNormal( 1.5 )
```

This should produce the answer 0.9331.

Tip: Help! my function doesn't work!

If your function isn't working, check the following:

- Did you give the file **exactly** the same name as the function except for the suffix `.m`? Did you make sure the name of the function did not contain any spaces?
- Is the function in the current folder? It is possible to open a file in the MATLAB editor that is not in the current folder.
- Have you saved the code for your function? MATLAB will let you rerun a function without saving it.

The advantages of making this a function are enormous

(i) The code is saved to a file for future reference.

(ii) We can easily use the same function to compute the cumulative normal distribution function for different values of x.

(iii) We can use this function from other functions *so we will never have to write the same code again.*

Let us understand what our function is doing.

The easiest bit of code to understand is the line beginning with a percentage signs and coloured in green. This is just a comment and is only there to remind you what the code is meant to do. MATLAB ignored any text after a percentage sign, so you can write whatever comments you like. It is always a good idea to add comments to your code so you can understand what it does when you come and look at it a year later. Choosing good function and variable names is probably even more important than writing comments, however.

The next easiest bit to understand is the main body of the code (the section after the comment but before the keyword `end`). This code is simply copied and pasted from our solution to the previous exercise. It works just like ordinary MATLAB code.

The really knew thing is the first line of the code. This is called a function declaration and it says that next piece of code (up to the keyword `end`) defines a function. This function is called "cumulativeNormal". It takes one parameter called x and computes one value called *result*. Notice that MATLAB colours in the keywords for you. This makes the code a little easier to read and reminds you not to use the keywords as variable names.

Tip: Red marks

Try deleting one of the semi-colons in our function definition. You should see that MATLAB underlines the code in red and puts a little red mark in the margin.

This is MATLAB's warning that you've probably made a bug in your code. When writing code you should make sure there are no red marks at all.

In this case, MATLAB is complaining because if there is a semi-colon missing, your function will print out part of its working. This is very confusing to someone using your function: how would you like it if the `sin` function printed out its working?

The general syntax for a function declaration is:

```
function [ <output values> ] = ...
 <functionName>( <input values> )
```

Here output values and input values should be lists of variable names separated by commas. The input values are the variable names that will be filled with the parameter values. The output values are the variable names that should be used to store the computed return values.

In our example, there is one input value called `x` and one output value called `result`.

As another example, here is a function that takes as input two polar coordinates and computes the Cartesian coordinates of the point. This function takes two parameters and returns two values.

```
function [ x, y ] = polarToCartesian( r, theta )
x = r * cos( theta );
y = r * sin( theta );
end
```

Here is how you would use this function:

```
r = 2.0;
theta = pi/2;
[ x, y ] = polarToCartesian( r, theta );
disp( x ); % Prints out the value of x
disp( y ); % Prints out the value of y

%If you don't need y
x = polarToCartesian( r, theta );

%If you don't need x
[~,y] = polarToCartesian( r, theta );
```

If you now look back at the code that was automatically generated when we created the file `cumulativeNormal` you should see that it contains an outline function for you to fill in. The phrase `input_args` in the input code is short for input arguments. An argument of a function is just another word for a parameter of a function.

Tip: Docking Windows

You will probably find that MATLAB opens a separate window for the code for your function. Personally I hate having lots of windows open and so I like to "dock" the editor window. You can do this by clicking on the window and pressing **CTRL+SHIFT+D**.

2.3 Exercises

1) Check that you can create and run the function `polarToCartesian` and test that it works using the code above. (You may notice that it doesn't give precisely the correct answers, this is because MATLAB only stores numbers up to a certain accuracy.)

2) Create and run an inverse function called `cartesianToPolar`

3) Write a function that allows you to solve the quadratic equation $ax^2+bx+c=0$. It should take three parameters a, b and c and return two values.

4) Write a function that computes the price of a call option using the Black–Scholes formula.

It should be invoked as follows: `blackScholesCallPrice( K, T, S, vol, r )`.

2.4 Writing tests

Code that doesn't give the right answers is useless. Nobody in their right mind would trust your pricing code using the Black Scholes formula and less you could ensure them that it was thoroughly tested. As you gain experience in programming, you will quickly learn that the following is (nearly) universally true:

Tip: Universal Law of Programming

Code that is not tested does not work.

When you write code, you should also write test functions for your code. These test functions are called "unit tests". Some advantages of writing your tests as MATLAB functions are:

1. It is quick and easy to rerun the tests whenever you change the code.
2. It provides documented proof that your code is tested and what those tests are.
3. The tests for a function provide excellent documentation for how that function should be used.

In this course I will often ask you how you would test your code. I will expect you to be able to devise testing strategies and write tests for your code. I may well make you do so in your exam.

As an example, let us consider how to test the `cumulativeNormal` function. After a little thought I came up with the following tests:

(a) `cumulativeNormal(x)` should be between 0 and 1 for aribtrary x.

(b) For $x = -1000.0$, `cumulativeNormal(x)` should be nearly 0.

(c) Since the normal distribution is symmetric, `cumulativeNormal(-x)` should be approximately equal to `1-cumulativeNormal(x)`.

(d) It is well known that 2σ events happen about 5% of the time. So `cumulativeNormal(2.0)` should be about 0.975.

Here is a test function for `cumulativeNormal` that checks all these behaviours.

```
function testCumulativeNormal()
x = 0.3;
assert( cumulativeNormal(x) > 0.0 ); %a
assert( cumulativeNormal(x) < 1.0 ); %a
assert( abs( cumulativeNormal(-20.0) ) < 0.001 ); %b
assert( abs( cumulativeNormal(-x) + ...
    cumulativeNormal(x) - 1 )<0.001); %c
assert( abs(cumulativeNormal( 2.0 ) - 0.975) < 0.01 ); %d
end
```

You can enter this code directly into MATLAB, it understands the `...` to mean that the code is continued on the next line.

Notice that in this example, the function `testCumulativeNormal` takes no parameters and doesn't return a value: all it does is perform the action of running some tests. This is typical of a test function.

Notice also that if you run the function, it doesn't do anything. This is the standard for tests. If they pass they print nothing out, but they print out error messages if something goes wrong. That way if you run lots of tests its easy to spot if any of them have failed.

The function `assert` is very useful for testing. It stops the program with an error if the assertion isn't true. So the two lines marked with the comment `\% (a)` simply check that for the given value of x, `cumulativeNormal` lies between 0 and 1.

More generally you can use the operators `<`, `>`, `<=` and `>=` for testing statements.

The function `abs` computes the absolute value of a number. So to check statement (b) when we test that a number is approximately 0 we need to check that its absolute value is near zero.

I recommend whenever you write a function you write a corresponding test function with a name beginning "test".

2.5 Using functions to simplify code

Our test code in the previous section became a little hard to follow towards the end. In particular you definitely need to apply your brain to work out that this line of code

```
assert( abs( cumulativeNormal(-x) + ...
    cumulativeNormal(x) - 1 )<0.001; % (c)
```

really is testing the fact that `cumulativeNormal(-x)` is approximately equal to `1-cumulativeNormal(x)`.

To make the code easier to understand, we could have simplified it by writing a new function as follows:

```
function assertApproxEqual( x, y, tolerance )
assert( abs( x-y)< tolerance );
end
```

With this function the somewhat confusing line of code

```
assert( abs( cumulativeNormal(-x) + ...
    cumulativeNormal(x) - 1 )<0.001; % (c)
```

could have been rewritten as:

```
assertApproxEqual( cumulativeNormal(-x), ...
```

```
                    1-cumulativeNormal(x), 0.001 );
```

This reads almost like the English statement of the test.

2.6 Passing functions as arguments

We can also simplify our code that uses the rectangle rule by writing a function `integrateNumerically( f, a, b, N)` which takes as parameters the function, the limits of the integration and the number of steps to take.

Here is the relevant code.

```
function [r]= integrateNumerically ( f, a, b, N )
h = (b-a)/N;
s = a + ((1:N) - 0.5) * h;
r = h * sum( f(s) );
end
```

For example to compute an approximation to $\int_0^1 \sin(t)\,\mathrm{d}t$ you would type `integrateNumerically( @sin, 0, 1, 1000 )`. When you pass a function as an argument you need to put the `@` symbol in front of the function name.

Notice that our function assumes that the function `f(x)` takes a vector of points `x` and returns a vector of the computed values `f(x)`. That's the way built in functions such as `sin` and `exp` all work in MATLAB. However, are function `cumulativeNormal` doesn't work like this at the moment, so it won't work for that function.

When you want to pass a function as a parameter to another function, that function has to be defined. Let us consider the problem we looked at earlier of computing

$$\int_0^1 s^{-2} \exp(-((x+1-1/s)^2)/2)$$

. We need to define a new function for the integrand in order to perform the desired integration. Here's how we could do this:

```
function result=cumulativeNormalVersion2( x )

function r = integrand( s )
    r = s.^(-2) .* exp( -(( x + 1 - 1./s).^2 )/2 );
end

NSteps = 1000;
result = 1/sqrt(2*pi) ...
    * integrateNumerically( @integrand, 0, 1, NSteps);

end
```

Notice that this code contains a *nested function* declaration. The syntax for nested functions is just the same as for ordinary functions except that a nested function is declared inside another function. Notice the way the nested function can use the variable x defined in the outer function.

Technically speaking when you pass a function as a parameter you need to pass a "function handle" this is just MATLAB's term for a reference to a function. The symbol means "compute the handle of this function". If you like you can store function handles in variables and arrays. This can be useful in more advanced programs, but this isn't a feature we will use in this course.

2.7 Scripts

An alternative way to save your MATLAB code is to write a *script*. A script is just a collection of MATLAB scripts that you can execute together.

You create a script by right clicking in the **Current Folder** and selecting **New→Script** and then calling it whatever you like. You run the script by opening it and then pressing the "Run" button in the toolbar (you can find this under the Editor tab in MATLAB 13).

Scripts are a good way to experiment with MATLAB code. When you change a variable in a script, you can see the changes in the **Workspace**. This might help you follow what is going on.

You can also run a script line by line by highlighting the text you want to run and clicking the "Run selection" button on the toolbar. By working through a script line by line, you may find it easier to grasp. Notice that this button also works for stepping through your functions so you can use it to help understand how functions work.

If you are new to MATLAB you may find scripts easier to use than functions, but it turns out that they aren't much use for writing serious code. The problems are:

- In scripts you see the whole computation "at once". Functions on the other hand are modular, the problem is broken down into small pieces. Understanding the whole computation in one go is usually difficult. Once it is broken down into simpler steps everything is simpler.
- Scripts do not let you reuse your code. If you use scripts, you will find you spend a lot of time cutting and pasting code from one to the other. With functions this is unnecessary.
- You cannot automate tests for scripts.
- Different scripts share the same variables - the ones you can view in the **Workspace**. Functions start with a fresh set of variables each time. This means that scripts can interact with each other in strange ways whereas functions are more predictable. For example it is easy to write a script that works the first time you call it and then starts failing.

In summary scripts are hard to debug and prevent code reuse. These are two very bad things!

If you find scripts helpful and want to use them, you can do so. But please write functions for all the exercises. Since scripts are harder to debug than functions, it is unreasonable to ask a tutor to help you debug your scripts when you could have written functions in the first place!

2.8 Exercises

1) Use the function `assertApproxEqual` to simplify the function testCumulativeNormal. You should be able to make three different simplifications. Notice how much more readable the code becomes.

2) Write a test function for your Black Scholes formula. It should test the "static bound" that the price of a call option is always greater than $S - exp(-rT)K$. It should also check that very near to maturity the price is well approximated by the immediate exercise value. Can you think of any other tests?

3) Use the function `integrateNumerically` to compute $\int_0^1 \sin(s)\,\mathrm{d}s$ and also to compute $\int_1^3 (x^2 - 2x + 2)\,\mathrm{d}x$.

4) Write some automated tests for the function `integrateNumerically`.

5) Write a function `integrateFromMinusInfinity( f, x, N)` which makes the substitution $t = x+1-\frac{1}{s}$ and uses this to evaluate the integral $\int_{-\infty}^{x} f(t)\,\mathrm{d}t$ using the rectangle method with N steps. This function should itself call `integrateNumerically`. Test your function. Modify the cumulativeNormal function so that it calls this function.

6) Write a function `normalDensity` which computes the probability density function of the normal distribution. Modify the cumulativeNormal function so that it calls this function.

2.9 What are the benefits of functions?

> Design principle: The once only principle
>
> You should never write the same code twice. Functions are designed to stop you having to ever write the same code twice. You break functions into pieces so you never even write the same couople of lines twice. You pass functions as arguments so you never have to write an algorithm twice.

We have now arranged our code so that it is written several small functions each of which should be easy to understand but which together perform a complex task. This task is to compute the price of call option using the Black–Scholes formula. You can find the final version of the code on the Keats page.

In this final version of the code, the function `blackScholesCallPrice` depends the `cumulativeNormal` which depends on `normalDensity` and `integrateFromMinusInfinity`. The latter depends on `integrateNumerically`.

I want to convince you that splitting the code into lots of pieces like this has made it much simpler to understand.

First notice in the process of smashing our functions into small pieces, we have written some functions that we will want to reuse in the future. For example, we would want to use cumulativeNormal when pricing a put option or digital option. Similarly our integration functions are obviously generally useful.

Second, the functions we end up with are much easier to understand. For example, here is a final version of the cumulativeNormal function:

```
function result=cumulativeNormal( x )
NSteps = 1000;
result = integrateFromMinusInfinity( ...
           @normalDensity, x, NSteps);
end
```

All of the complexity such as the rectangle rule and the integration by substitution have been put in separate functions, so this code contains very little other than the definition of the cumulative distribution function.

Third, if we decide to improve one of our functions, say by using a faster algorithm, then all the code that uses those functions will become better too. For example we might change the function `integrateNumerically` so it uses the trapezium rule, or Simpson's method. All our other code would benefit immediately.

A fourth advantage of writing lots of small functions is that we can test each piece of code separately. It requires a bit of ingenuity to think of tests for `cumulativeNormal` and a lot of ingenuity to think of tests for `blackScholesCallPrice`. On the other hand, testing component pieces such as `integrateNumerically` is pretty simple.

Also notice that in order to break our code into small pieces, we have been constantly modifying our original functions. This process of introducing new functions and polishing your code to make it easier to understand is called "refactoring". Notice how helpful it is to have automated tests when you are refactoring your code. You can quickly rerun all the tests and make sure that your code changes haven't broken anything.

Tip: Using functions

- When you write code write lots of small functions.
- Try to reuse functions you have already written rather than always writing new code.
- Write automated tests for each function that you write.
- Give your functions and variables descriptive names so that your code reads like English.
- If your code is too complicated to quickly understand, break it into small functions.

2.10 Summary

We have learned how to use functions to achieve the following:

- **Reuse**. Functions allow us to reuse code rather than constantly rewrite it.
- **Modularity**. We can break our programs into small pieces each of which is easy to understand - ideally into pieces that look like they're written in plain English.
- **Testing**. Small functions are easy to test. Our tests are themselves written as functions.