Chapter 3

Flow of control

3.1 For loops

If you want to repeat the same piece of code multiple times, you can use a `for` loop. Here is a silly program that illustrates loops:

```
function bowieIsCool()
for j=1:100
    disp(j);
    disp('Bowie is cool!');
end
end
```

This prints out alternately the numbers from 1 to 100 and the phrase "Bowie is cool!". Feel free to write your own version of this program with whatever name you think is cool. The `for` loop executes the code

```
    disp(j);
    disp('Bowie is cool!');
```

one hundred times with the value of the variable j moving from 1 up to 100.

Incidentally, this is the first time we have used a string in our code. By enclosing text in single quotation marks, you create a string which MATLAB can then print out using the `disp` command. If you want to include a single quotation mark inside your string then simply double up the quotation marks. For example `'Bowie''s first album'`. MATLAB colours strings in for you so you can easily check if you've matched up the quote marks.

The code that is executed in a for loop is determined by the first `end` statement. I've indented the code to make it clearer which part is repeated, but

MATLAB actually just ignores the spaces. It is the `end` statement that determines where the code ends.

Here is a more sensible example that actually does something useful:

```
function result = computeFactorial( n )
current = 1;
for j=1:n
   current = current * j;
end
result = current;
end
```

If you have programmed before, this is probably very easy to follow. For the benefit of those who are new to programming let us consider what happens if we call this function with the value $n = 4$.

Then the first thing that will happen is that current will be assigned the value 1. Then we repeat the code inside the `for` loop 4 times each time with a different value of j. Each time the code is executed, the value of current is multiplied by the current value of j.

So `current` starts at 1. On the first iteration we multiply it by 1 to obtain `current=1`. On the second iteration we multiply this new value of `current` by 2 to obtain `current=2`. On the third iteration we mutliply current by 3 to obtain `current=6`. On the fourth iteration we multiply current by 4 to obtain 24. This is the desired result, so after the loop is completed we assign the value of `current` to result.

In the code below, we've added some `disp` statements to the code so you can see the functions working.

```
function result = computeFactorial( n )
current = 1;
disp( 'The initial value of current' );
disp( current );
for j=1:n
   disp( 'The value of j');
   disp( j );
   disp( 'The old value of current');
   disp( current );
   current = current * j;
   disp( 'The new value of current');
   disp( current );
end
disp( 'The final value of current');
result = current;
```

If you have never seen a loop in a computer program before, you should write down what you think this loop will print out and then try it to see if you are correct.

If you want to count down from 10 to 1. You could use the following code:

```
function launchRocket()
for j=10:-1:1
    disp(j);
end
disp('Blast off');
end
```

There's nothing special about the name `j` in the code above. We could also have written:

```
function launchRocket()
for number=10:-1:1
    disp(number);
end
disp('Blast off');
end
```

In general in MATLAB you can run loop through the elements of any vector. For example

```
function digitsOfPi()
nums = [ 3 1 4 1];
for j=nums
    disp(j);
end
end
```

You aren't restricted to using loops inside of the Command Window. For example you can type

```
nums = [ 3 1 4 1];
for j=nums
    disp(j);
end
```

straight into the command window. Notice that no output is printed by the loop until you type end.

3.2 If statements

A `for` loop is an example of a "flow of control" statement because it determines how MATLAB chooses which line of the program to execute next. In other words it decides what the next MATLAB statement is that has "control" of the program execution.

Another flow of control statement is an `if` statement. This makes a decision on whether MATLAB will execute the next line of code or not.

```
function max = maximum( a, b )

if a>b
    disp('a is bigger');
    max = a;
else
    disp('b is bigger');
    max = b;
end

disp('The maximum value is:');
disp( max );

end
```

The if statement chooses which branch of code to execute according to whether the statement `a>b` is true. If it is true, it executes the first branch of code. In this case the text "a is bigger" is printed out and max is assigned the value a. If the statement is false, the code in the `else` branch is executed.

Notice that like `for` loops, an `if` statement must be ended with an `end` statement.

You can use the operators `>`, `>=`, `<`, `<=` in the comparisons for if statements. You can also test if two numbers are equal using **two** equals signs as shown below:

```
function isValueSeven( value )
if value==7
   disp('The value is seven');
else
   disp('The value is not seven');
end
end
```

To test if two numbers are not equal use `~=` that is a tilde sign followed by equals.

You can use the operators && meaning "and" and || meaning "or" and ~ meaning "not" to construct more complex tests if you wish.

For example this function tests whether a value is 3 or 7:

```
function isValue3Or7( value )
if value==3 || value==7
   disp('The value is either 3 or 7');
else
   disp('The value is neither 3 nor 7');
end

end
```

You can build quite complex expressions. For example the following code prints out 'Test passed' if either:

- a and b are both positive
- or b is not equal to 7

```
function complexTest( a, b )
if (a>0 && b>0) || ~(b==7)
    disp('Test passed');
else
    disp('Test failed');
end
end
```

Notice the use of brackets in the expression. Without the brackets it is very hard to work out what the code will actually do.

The `if` statement is quite flexible. If you don't want anything to happen if the condition isn't true, you can omit the `else` block completely. You can also add in `elseif` blocks to perform additional tests.

Here is the general syntax for an if statement in MATLAB:

```
if test1
    statements
elseif test2
    statements
elseif test3
    statements
elseif test4
    ...
```

```
else
    statements
end
```

3.3 While loops

Another common programming construction is that you want to keep repeating some task until a condition is met. For this you can use while loops.

The general syntax for a while loop is

```
while conditionIsTrue
    statements
end
```

This will repeatedly execute the statements until the condition is no longer true.

For example here is how you can use a while loop to print the numbers from 1 to 10.

```
function countUsingWhile()
count = 1;
while count<=10
    disp(count);
    count = count + 1;
end

end
```

This code uses a counter called `count` which it increments on each step of the while loop until such time as `count<=10` evaluates to false.

Most people would agree that the equivalent code using a `for` loop is easier to understand. Nevertheless, while loops have their uses, particularly when you don't know in advance how big the loop needs to be.

For example, here is a function nextPrime that computes the next prime number:

```
function prime2=nextPrime(prime1)
current = prime1+1;
while ~isprime(current)
    current = current+1;
end
prime2 = current;
end
```

We're using the MATLAB function `isprime` here to test if a number is a prime number or not. This code keeps incrementing the variable `current` until such time as `isprime(current)` evaluates to `true`. (Recall that `~` means "not" in MATLAB).

At the end of the loop, we know that the value of the next prime number is contained in the variable current.

3.4 Accessing individual cells in a matrix

Suppose that we have created an $m \times n$ matrix called `A` then to find out the value of the cell at the coordinates `(i,j)` one simply types `A(i,j)`. Here is an example:

```
A = [ 1 2 3 4; 4 6 7 8];
A(2,3)
```

This code prints out the number 7.

If you try to access read the value in a cell that doesn't exist, MATLAB will produce an error.

As well as using this notation to find out the value of a cell, you can use it to set the value of a cell. For example

```
A = [ 1 2 3 4; 4 6 7 8];
A(2,3) = -7;
```

Would change `A` so it contains the values:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & -7 & 8 \end{pmatrix}$$

If you try to write a value to a cell that doesn't exist, MATLAB will make the matrix bigger to accomodate the value. It pads up the rest of the matrix with zeros. For example the code:

```
A = [ 1 2 3 4; 4 6 7 8];
A(3,6) = -7;
```

changes `A` to be the matrix:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 0 & 0 \\ 5 & 6 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

You can also access entire submatrices using the `:` operator. For example if `A` is a 3×7 matrix as above then to access the fourth column of A you can write `A(1:3,4)`.

Similarly if you wanted to access the second row you can write `A(2,1:6)`.

Since accessing entire rows and columns is so common, you can use the keyword `end` rather than having to type in the length of an array. For example to access the second row of `A` you can type `A(2,1:end)` and this will access the second row irrespective of the number of columns.

Similarly the expression `A(1:2,1:2)` refers to the two by two matrix in the top corner of `A`.

In actual fact, these are all special cases of the general syntax:

```
<matrix>( vectorOfRows>, <vectorOfColumns> )
```

which can be used to extract any desired submatrix from a matrix.

MATLAB treats row and column vectors as special cases of matrices, so you can access their elements using indices just as you do for matrices. However, if you like you can also specify just a single coordinate. For example `v(5)` accesses the fifth element of a vector `v`.

In fact, you can use a single coordinate for all matrices, not just vectors. Each cell in a matrix has an associated number, the numbers start at 1 in the top left, then increase by one up each column in turn. Here is a diagram showing how a 5×3 matrix is ordered.

$$\begin{pmatrix} 1 & 6 & 11 \\ 2 & 7 & 12 \\ 3 & 8 & 13 \\ 4 & 9 & 14 \\ 5 & 10 & 15 \end{pmatrix}$$

Try to work out what the code below will print out and see if you are correct.

```
A = [ 1 2 3 4; 4 6 7 8];
A(3,6) = -7;
A
A(1:3,4)
A(2,1:6)
A(2,1:end)
A(1:2,1:2)

rows = [1 3];
columns = [2 4];
A(rows, columns)

A(8)
```

3.5 Putting it all together

We can combine `for` loops, `if` statements and accessing elements of arrays to write interesting programs.

Example 1: Without using the `sum` function, write a function called `mySum` which takes a vector `x` as parameter and adds up all the cells of `x`.

Solution: We use a variable `total` to store our working and iterate through all the elements of `x` using a `for` loop increasing `total` as we go:

```
function [total] = mySum( x )
total = 0;
for j=1:length(x)
    total = total + x(j);
end
end
```

Example 2: Write a function `primesUpTo` that takes a parameter `n` and returns a vector containing all the prime numbers less than `n`.

Solution: We use the function `isprime` mentioned above. We use a for loop to run through all the integers from 2 up to n. We start with an empty array primes and add a new element on the end whenever we find a new prime. In order to decide where to add the new element we use a variable called `counter` which keeps track of the number of primes found so far.

```
function primes = primesUpTo( n )
counter = 1;
for j=2:n
   if (isprime(j))
      primes( counter ) = j;
      counter = counter + 1;
   end
end
end
```

Try entering this code in MATLAB.

Whenever we add a new prime to the vector `primes`, MATLAB has to resize the vector. This is a bit slow since to resize a vector you often have to shuffle round quite a bit of computer memory. For this reason MATLAB prefers it if you decide how large the vector should be at the beginning. This is why the line `primes( counter )=j` is marked with a red line in the MATLAB editor. Here is some code without the red line:

```
function primes = primesUpTo( n )
primes = zeros( 1, n );
counter = 1;
for j=2:n
   if (isprime(j))
      primes( counter ) = j;
      counter = counter + 1;
   end
end
primes = primes(1, 1:(counter-1));
end
```

3.6 Exercises

1) Write a function `myProd` to compute the product of all the elements in a vector.

2) Write a function to find the maximum value in a vector. You are not allowed to use the MATLAB `max`, `min` or `sort` functions!

If you are new to programming, you may find this question difficult. If you struggle, imagine you were given one thousand cards each with a different number printed on it. How would you find the maximum? Write down detailed instructions for how you would do this in English and then try to convert them into MATLAB code.

3) Modify the `integrateNumerically` function from the last chapter so that it uses a `for` loop rather than a `sum` statement. The benefit of this is that `integrateNumerically` will now work for functions like `cumulativeNormal` that can only process a single argument rather than a vector of values.

4) In the game paper-scissors-stone, let the number 0 represent paper, the number 1 represent scissors and the number 2 represent stone.

Write a function `hasPlayerAWon( A, B)` that uses `if` statements to decide who has won given the numbers representing the selections of player A and player B.

5) You can use the value `inf` to represent infinity and the value `-inf` to represent negative infinity in MATLAB.

Given this, write a function `integrateNumericallyVersion2(f, a, b, N)` that allows you to specify infinite values for the integration range `[a,b]`. You will need to perform appropriate substitutions before calling the old function `integrateNumerically` with a finite range.

6) The Fibonnacci sequence is defined by $x_1 = 1$, $x_2 = 1$ and thereafter by $x_n = x_{n-1} + x_{n-2}$. Write a function `fibonnacci(n)` that computes the n-th Fibonnaci number x_n.

3.7 Logical values

Expressions such as `3>5` have values of either `true` or `false`. These are called logical values or "boolean values" after the English mathematician George Boole.

Most of the functions we have written so far have all returned numeric values, but functions can return logical values if you like. The function `isprime`, for example, returns whether or not a given number is a prime number.

When MATLAB prints out a logical value it prints the number 1 for the value "true" and 0 for the value "false".

In general MATLAB is quite happy to use non-zero numbers to mean `true` and the number zero to mean `false`. So for example the code:

```
if 3
    disp('3 is non-zero');
end
```

will print out the message "3 is non-zero". The code isn't very readable though, is it? So I recommend that you don't use this technique.

Like most other MATLAB operators you can use operators such as `>`, `<` and `==` on matrices. This will then produce matrices of logical values which are displayed as matrices of zeros and ones.

Here is an example:

```
v = [-3 -2 -1 0 1 2 3];
isPositive = v > 0;
disp( isPositive );
```

If you run the code above you will see that the vector `isPositive` is assigned the values `[0 0 0 0 1 1 1]`.

You could have written the code above using a `for` loop. Here is the equivalent code:

```
v = [-3 -2 -1 0 1 2 3];
isPositive = zeros( 1, length(v));
for j=1:length(v)
    isPositive(j) = v(j) > 0;
end
disp( isPositive );
```

In MATLAB, the first version of the code would be preferred: as well as being shorter it actually runs marginally faster.

3.8 Matrix Programming

In general `for` loop statements are quite slow in MATLAB, but statements which perform repeated the same operation on all the elements of a matrix are fast.

The reason why vector operations are fast is that computer processors contain special optimizations for performing such repeated tasks.

If you ask MATLAB to perform a computation using matrix operations, it will use these optimizations. If you write a `for` loop, MATLAB won't use these optimizations. So it can often be useful to rewrite your code to avoid unnecessary for loops (however, see the tip below). Rewriting your code in this way is called *matrix programming* or sometimes *vectorization*.

I'll mostly present efficient code in the lectures, but you may find it tricky to write vectorized code yourself at first. I've collected together in this section some tips that you can use to eliminate unnecesary `for` loops.

Tip: Don't optimize unless you need to

It is more important that your code works than that it works quickly. You should not worry about speed when you write your code, only worry about it if it proves to be too slow for your needs. Don't waste time and energy optimizing code that is fast enough already.

3.8.1 Use MATLAB's libraries when possible

The first tip for matrix programming your code is to use MATLAB's libraries wherever possible. They are well written and use vector operations wherever possible.

For example, a simple improvement to the `blackScholesCallPrice` from the last chapter is to make it use the MATLAB library function `normcdf` rather than our `cumulativeNormal` function. This function will be both more accurate and more efficient than the `cumulativeNormal` function.

3.8.2 Make your functions work with vectors

The second tip is to write all your functions so that they can take arrays of values and perform the same computation for the entire array. This is the way that MATLAB functions such as `sin` and `exp` all work. You should try to write functions like this too.

As an example here is a function `computeInterest(P,r,t)` that computes the interest earned on a principal `P` over time `t` given that the continuously compounded interest rate is r.

```
function interest = computeInterest( P, r, t )
interest = P * (exp( r * t )-1);
end
```

We can write an almost identical function which takes an array of principals, an array of time periods and an array of interest rates and computes the interest earned in each case.

```
function interest = computeInterest( P, r, t )
interest = P .* (exp( r .* t )-1);
end
```

The only difference in the code is a few dots. However, if you have to calculated interest on a lot of accounts (as a bank might well have to do), the second function can allow these similar calculations to be vectorized.

3.8.3 Using arithmetic with logical values

Since MATLAB treats the value `true` almost interchangeably with the value `1`, you can often get rid of `if` statements and replace them with arithmetic calculations.

As an example, suppose that we wish to write a function `computeProfit` that calculates the profit made after tax for a number of scenarios simultaneously. The function takes three parameters: a vector representing the earnings in each scenario; a vector representing the costs in each scenario; the the tax on positive profits as a proportion.

Here is a first version of the code that uses a for loop:

```
function netProfit = ...
          computeNetProfit( earnings, costs, tax )
grossProfit = earnings-costs;
taxPayable = zeros(length( grossProfit), 1 );
for j=1:length(grossProfit)
    if (grossProfit>0)
        taxPayable(j) = tax * grossProfit;
    end
end
netProfit = grossProfit - taxPayable;
end
```

This can be rewritten without the for loop as follows:

```
function netProfit = ...
         computeNetProfit( earnings, costs, tax )
grossProfit = earnings-costs;
isTaxPayable = grossProfit>0;
taxPayable = isTaxPayable .* grossProfit .* tax;
netProfit = grossProfit - taxPayable;
end
```

The trick here is that, viewed as a number, `isTaxPayable` is 0 if there is no tax payable. In this case `totalTax` will come to 0. On the other hand when `isTaxPayable` is equal to 1, totalTax will simplify to just `grossProfit .* tax`.

Here's another example which gives a general recipe for getting rid of if statements.

```
if test1
   v = value1;
elseif test2
   v = value2;
else
   v = value3;
end
```

Can be replaced with

```
v = test1 * value1 ...
    + (~test1)*( test2*value2 + (~test2)*value3 );
```

You can often then simplify the expression for `v` further.

Of course there is always a danger that a change like this will make your code harder to understand. This is why you should only optimize if there is an actual performance problem.

3.8.4 Using logical indices

We have seen how to access elements of a matrix by specifying the coordinates of a particular cell. Another approach is called logical indexing. To access an array, `A`, using logical indexing, you supply an array of `true`/`false` values and ask MATLAB to return you the vector consisting of all the elements of `A` where the logical value is true.

For example given the array `A=[-1 3 5 7]` you could specify the logical indices `[true true false true]` in order to access the subvector `[-1 3 7]`.

Here is some MATLAB code to try to demonstrate this:

```
A = [-1 3 5 7];
cellsToAccess = [true true false true];
A(cellsToAccess)
```

Here is an example of how you can use logical indexing to rewrite the function `primesUpTo` that we saw earlier:

```
function primes = primesUpTo( n )
allIntegers = 1:n;
logicalIndices = isprime( allIntegers );
primes = allIntegers( logicalIndices );
end
```

Notice that the function now contains no `for` loop. Notice that in this case the code is not really actually appreciably quicker. This is because the part of the code that tests if a number is prime is much slower than the code needed for a loop. This gives another example of why you shouldn't optimize your code unless there is a problem.

Nevertheless, there are occasions when this technique can give a performance boost.

3.8.5 How to work out if code can be improved using matrix programming

Not all `for` loops can be replaced with a matrix calculation. A loop can be replaced with a matrix calculation if all the calculations in the loop are independent of each other. If you need the result from iteration $n-1$ to perform iteration n then you probably can't make it more efficient.

3.9 Exercises

1) Write your own function `myisprime` that tests if a number is a prime or not. You can use the function `rem` which computes the remainder of two numbers after a division.

2) Modify the function `blackScholesCallPrice` from the last chapter so that it can take a vector for each parameter and so compute call option prices for a variety of scenarios all with one function call.

3) Write a function `blackScholesPrice` which behaves like `blackScholesCallPrice` except that it also takes an array of logical values indicating whether the option is a put or a call and prices the option accordingly.

Can you write this code so that it operates on vectors of parameters without using any `for` loops? To do so you will need to vectorize any `if` statements.

4) Without using a `for` loop, find the sum of all the numbers $\sin(n)$ where n is between 1 and 100 (inclusive) and $\sin(n)$ is greater than one half.

3.10 Summary

We have learned how to use `for` loops, `while` loops and `if` statements to control the behaviour of our programs and to automate repetitive tasks.

We have learned how to specify parts of a matrix using indexes, vector indices and logical indices.

We have learned some tricks to avoid writing loops in order to make MATLAB code faster. The process of getting rid of loops is called matrix programming.