Loops and branches

The M-files we examined in the previous practical are all a list of instructions carried out sequentially. For most numerical applications, however, we need more flexible codes. In particular, we need loops and branches. Loops are needed, for example, to converge to roots iteratively, while branches are necessary to allow the codes to flow in different directions when, for example, convergence has been reached.

For Loops

Suppose that we want to calculate \( y = \sum_{n=1}^{5} 2^n \). We could type the direct translation of the summation:

\[
>> y=2^1+2^2+2^3+2^4+2^5
\]

but another, more flexible, way is

\[
>> y=0; \text{for } k=1:5; y=y+2^k; \text{end} ; \text{disp}(y)
\]

This is a more flexible way, since it requires only a tiny change to calculate \( y = \sum_{n=1}^{10} 2^n \) instead of \( y = \sum_{n=1}^{5} 2^n \). We only need to alter the 5 in for \( k=1:5 \) into 10 to achieve this change.

So, what is happening? We can take a closer look at what is happening inside the loop by putting another disp statement inside the loop:

\[
>> y=0; \text{for } k=1:5; y=y+2^k; \text{disp(sprintf('%d%s%d',k,' ',y)); end} ; \text{disp}(y)
\]

When the for \( k=1:5 \) statement is reached, MATLAB sets \( k \) to the initial value of 1, and then continues carrying out the statements, until the end statement is reached. When MATLAB reaches the end statement, it first increases \( k \) by 1 to 2. Then it compares \( k \) with the final value of 5. If \( k \) is smaller than or equal to the final value, MATLAB jumps back to the beginning of the loop and continues working from there. If \( k \) is larger than the final value, then it goes out of the loop and continues from the end statement.

Do not worry about the sprintf statement: it is an aid to print the numbers on the screen. If you want an explanation: try the help facility.

If you want a different increment, say an increment of 0.5, then you could replace for \( k=1:5 \) with for \( k=1:0.5:5 \), where the middle value indicates the increment.

If you want \( k \) to decrease, then you simply choose a negative increment, but you should ensure that your initial value is larger than the final value. Replace for \( k=1:5 \) with for \( k=5:-1:1 \).

It is possible to nest loops, but each for loop needs its own matching end statement

\[
>> \text{for } k=1:3; \text{for } l=4:6; \text{disp(sprintf('%d%s%d',k,' ',l)); end} ; \text{end}
\]

While loops

A second type of loop is the while loop. This loop is of the form

\[
\text{previous statements}
\text{while (condition)}
\text{loop statements}
\text{end}
\text{following statements}
\]

Upon encountering this loop, MATLAB checks whether (condition) is true. If it is, then it carries out the loop statements. When MATLAB reaches end, MATLAB checks again whether (condition)
is true. If \( (\text{condition}) \) is true, the loop is carried out again, while if \( (\text{condition}) \) is false, MATLAB continues with the following statements. Therefore, this loop is carried out until \( (\text{condition}) \) is false either before entering the loop or at the end of the loop statements.

**WHILE** statements can be appreciated best in an M-file. The following program finds a solution to the equation \( x = \cos(x) \) using the fixed-point method with a precision of 0.01. Edit a new M-file, `whiletest.m`

```matlab
% WHILETEST - Example of the use of a while loop
xold=0;
x=cos(xold);
while (abs(x-xold) > 0.01)
    xold=x;
    x=cos(xold);
    disp(sprintf('%g %s %g',x,' ',abs(x-xold)));
end
```

Save it and run it by typing `whiletest`. If it doesn’t recognize `whiletest`, you may have stumbled on a MATLAB problem. Quit MATLAB, enter MATLAB, and try `whiletest` again. When it runs properly, you will note that the difference between \( x \) and \( xold \) decreases steadily and when it becomes less than 0.01, the loop terminates. This approach actually converges linearly to the proper solution and thus is a very slow approach. Aitken’s \( \Delta^2 \) method can significantly speed up the convergence for this problem!

Again, each while loop needs an end. While loops can be nested. They can also be nested within for loops and for loops can be nested within while loops.

Since the end statement is identical for for loops and while loops, the end point of each loop can become difficult to find if M-files are not edited properly. A good technique is to use a couple of spaces to indent the part of the code that is inside a loop. This helps to reduce the number of programming errors and helps to spot bugs in the program. The matlab editor will indent your code automatically, but ensure that the indentation is turned off properly for the end statements.

**Branches**

Another way in which the flow of the program can be changed is using branches. For example, the absolute value of a number can be determined through \( y = \text{abs}(x) \), but you can also write a small M-file `absolx.m`

```matlab
% ABSOLX - put the absolute value of x in m
y=x;
if (y < 0)
    y=-x;
end
disp(y)
```

or `absolx2.m`

```matlab
% ABSOLX2 - put the absolute value of x in m
if (x >= 0)
    y=x
else
    y=-x
end
```

The relational operators, such as \( > \) and \( < \), are given in Pratap’s book on page 54. You can also combine several relational statements using logical operators, such as AND, as given on page 56.

Try the codes. Give \( x \) a certain value and then run `absolx` and `absolx2`. Notice again that each if statement needs a matching end statement. In conclusion, the if statement carries out a certain number of statements if a condition is met. Optionally, you can also include statements that are carried out if the condition is not met.
If statements can also be nested. In fact, for, while and if statements can be nested as you want, as long as the required number of matching end statements is provided.

Sometimes, there are cases when you want to exit a for or while loop when the regular conditions for exiting have not yet been met. You can enforce this by using the break statement. If loops are nested, than break will only terminate the innermost loop.
Exercises

These pages gave an overview of the most important control-flow statements in MATLAB. The understanding of these statements is essential for development of any (numerical) code. The only way to develop this understanding is by developing small programs yourself. Try the following exercises, using the for, the while and the if statements. If you stumble on one problem, an answer sheet will be available at the end of the practical.

1. Write a program that approximates

   \[ y = \int_0^\pi \sin(x)dx \]  
   \[ y = \frac{\pi}{2N} \left( \sin(0) + \sin(\pi) \right) + \frac{\pi}{N} \sum_{n=1}^{N-1} \sin\left(\frac{n\pi}{N}\right). \]

   \( N \) is the number of points used in the integration. You may need to use a for loop from 1 to N-1. Calculate the approximation using N = 5, 10, 20, and 50. Does the answer converge to the one you expect?

2. One of the more surprising mathematical equalities is

   \[ \sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}. \]

   Using a while loop, calculate this summation until the deviation from the exact answer is less than 0.1. Up to what n-value do you need to carry out the summation so that this deviation is achieved? How about a deviation of 0.01 and 0.001? Can you explain this answer?

3. Write a program that, given two positive integers N1 and N2, calculates their greatest common divisor. The greatest common divisor can be calculated easily by iteratively replacing the largest of the two numbers by the difference of the two numbers until the smallest of the two numbers reaches zero. When the smallest number becomes zero, the other gives the greatest common divisor.

   You will need to use a while loop and an if-else statement. The largest of two numbers can be obtained using the built-in function MAX(A,B). The smallest of two numbers can be obtained using the built-in function MIN(A,B).
Answers

1. % INTSIN - integrate \( \sin(x) \) from 0 to \( \pi \)
   
   \( N = 5; \)
   \[
y = (\sin(0) + \sin(\pi)) \pi / (2 \times N);
   \]
   for \( n=1:N-1 \)
   \[
y = y + \sin(n \times \pi / N) \pi / N;
   \]
   end
   disp(N)
   
   The integral should approach 2.

   The above program can be made significantly more efficient by for example setting \( p_n = \pi / N \)
   and then using \( p_n \) everywhere instead of \( \pi / N \). Optimization of numerical codes is another
   important aspect of good programming. However, at the present, we will not worry about
   this, since we are at present more worried about learning the techniques of programming.

2. % PI2OVER6 - Approximates \( (\pi)^{2 / 6} \)
   
   \[
y = 0;
   \]
   aim=pi*pi/6;
   n=0;
   while \( \text{abs}(\text{aim} - y) > 0.1) \)
   \[
n = n + 1;
   \]
   \[
y = y + 1 / (n + n);
   \]
   disp(sprintf('%g%s%g',n,' ','y,'',aim-y))
   end

   Since for large \( n \), the summation approaches
   
   \[
   \sum \frac{1}{n^2} \approx \int \frac{1}{n^2} dn = -d\left(\frac{1}{n}\right)
   \]

   the difference will approximate \( 1/N \) with \( N \) the final \( n \) in the summation. Thus one needs
   roughly 10, 100 and 1000 steps to obtain an accuracy of 0.1, 0.01 and 0.001 respectively.

3. % GRCODI - determine the greatest common divisor
   
   \[
   N1=6876;
   \]
   \[
   N2=45432;
   \]
   disp(sprintf('%g%s%g',N1,' ',N2))
   while \( \text{MIN}(N1,N2) > 0) \)
   \[
   \text{if } (N1 > N2) \)
   \[
   N1=N1-N2;
   \]
   \[
   \text{else } \)
   \[
   N2=N2-N1;
   \]
   \[
   \text{endif}
   \]
   disp(sprintf('%g%s%g',N1,' ',N2))
   end
   disp(sprintf('%s%g','Their greatest common divisor is ',MAX(N1,N2)))