LOOPING

Looping or iterative functions are extremely useful in engineering problem solving. A nice example is in solving an integral numerically. In carrying out a numerical integration, we will try to approximate an analytical solution (one that solves the problem by using limits going to zero or infinity) using a numerical approach that will have a finite, though large number, of steps. Solving the integral by hand would include thousands or millions of calculations. We can use loops in a function so that MATLAB will solve the million calculations automatically with one command. Numerical integration will be discussed at a later date. First we must get used to using these loops.

for Loops

The *for* loop will continue to do a specified calculation for a certain amount of time. The general formula for a *for* loop is below:

```
for ii=a:b:c
        {statements}
end
```

This statement will be inserted somewhere in an m-file. This *for* loop will start at *a*, execute the statements, which are some calculations. When it gets to the end statement, it will add *b* to *ii* and then check to see if *ii* is less than or equal to *c*. If so, it will redo the problem over and over again until *ii* is greater than *c*. At that point it will stop.

It is important to note that the index, *ii*, can have any name just like any other variable. In programming circles, it is common to use *i* as an index. However, in MATLAB *i* is already defined as the square root of negative one, i.e. an imaginary number.

An example of a *for* loop is doing element by element operations. Remember that MATLAB by default will carry out vector/matrix operations. A 1 x 3 matrix can be multiplied by a 3 x 1 matrix, but a 1 x 3 cannot be multiplied by a 1 x 3. The tutorial on Vectors and Matrices covered this. In this example, an element by element squaring of a vector will occur. Sure there are easier ways to do it, but this is a nice demonstration of for loops. In pseudo-code the steps to square a three element vector, *a*, element by element to obtain vector *c* are:

1) define *a*
2) calculate *a(1)*^2 and call it *c(1)*
3) calculate *a(2)* ^2 and call it *c(2)*
4) calculate *a(3)* ^2 and call it *c(3)*

We will put the *for* loop in an m-file, so the first step will be an input to the m-file, then we will use a *for* loop to calculate *c* and do steps 2-4. The m-file would look like:

```
function [c]=ebyesquare(a)
for ii = 1:3
        c(ii)=a(ii)^2;
end
```

The function takes *a* as an input. Then it will calculate *c(1)*, at the end it will go back to calculate *c(2)*, and then go back and calculate *c(3)* and stop. Notice that the increment was not given in the line with the *for* statement. The default increment of 1 works for this situation.

Now consider squaring a vector element by element of undetermined length. If the vector had 100 elements the function above would stop at 3. The upper limit of the index, *ii*, could be changed, but this is tedious. A built-in shortcut to MATLAB is the *length* command.

```
length(a)
```

This command will give the largest dimension of *a*, which for a vector is the number of elements. It is okay to use this shortcut in the course. It is a good one to memorize, which will be easy after you use it many times. We could build a separate function that finds the length of a vector, but MATLAB has built in protections to make it difficult though definitely not impossible. It would actually require a *for* loop to do it.

Getting back to the problem at hand, we can use the *length* command to set the upper limit of the iterations on the *for* loop. A new m-file would look like:

```
function [c]=ebyesquare(a)
len=length(a);
for ii = 1:len
        c(ii)=a(ii)^2;
end
```

That is a very simple example of a *for* loop. A problem will be given at the end to test your abilities. First there is another type of loop to cover.

while Loops

while loops will carry out a set of commands as long as a certain condition is true. The general form is:

```
while {some condition}
        {statements}
end
```

The conditional statements will compare things and as long as the statement is true, the *while* loop will continue until it is not. Consider the previous problem of squaring a vector element by element. A *while* loop to do so could look like:

```
function [c]=ebyesquwhile(a)
len=length(a);
```

```
ii=1;
while ii<=len
        c(ii)=a(ii)^2;
        ii=ii+1;
end
```

For a *while* loop, the index was initialized before the *while* loop began. The loop compares *ii* and *len*, finds that the statement is indeed true, so it calculates *c* and increments the index by one. Adding the increment to the index is not automatic in a *while* loop. It then returns to the *while* statement, compares *ii* and *len*, and continues as long as the statement is true. When the statement is false, it stops. A complication of *while* loops is that it is possible to write a program that never ends. For example, if you omitted the line that adds one to the index, the loop will calculate *c(1)* an infinite number of times. If you suspect that you have started an infinite loop hold down **control and C** to stop a program that is being executed.

The conditions are given below. The column of the left hand side is to compare two values and those on the right hand side are to add multiple conditions:

<=	less than or equal to	&	and
>=	greater than or equal to	~	not
==	equal to	|	or
>	greater than		
<	less than		
~=	not equal to		

EXAMPLE
Write an m-file that performs the element by element multiplication of two vectors of unknown length, *a* and *b*, into a third vector, *c*. Carry out the function on the following two vectors:

```
a = [1 2 3 4];
b = [5 8 2 5];
```

Start by writing the m-file with *a* and *b* as inputs and *c* as the output:

```
function [c]=ebyemult(a,b)
len=length(a);
for ii=1:len
        c(ii)=a(ii)*b(ii);
end
```

You can then go to the command window to execute the function with the following commands:

```
<<a=[1 2 3 4];
<<b=[5 8 2 5];
<<[c]=ebyemult(a,b);
<<c
c =
        5 16 6 20
```

A complication could arise if *a* and *b* are not of the same length. How would you fix it? We will discuss this at a later date.

Now let's solve with an m-file using a *while* loop:

```
function [c]=ebyemultwhile(a,b)
len=length(a);
ii=1;
while ii<=len
        c(ii)=a(ii)*b(ii);
        ii=ii+1;
end
```

Then in the command window run the following commands:

```
<<a=[1 2 3 4];
<<b=[5 8 2 5];
<<[c]=ebyemultwhile(a,b);
<<c
c =
        5 16 6 20
```

DO IT YOURSELF

1) Write an m-file that calculates the factorial of a number. Try it once using a *for* loop and a second time using a *while* loop.
2) Given the for loop below, what is *a*:

```
for ii=1:6
        a(ii)=ii^2-3;
end
```

ELSE IF STATEMENTS

else-if statements can be used to compare two numbers and give options depending on whether or not certain conditions are met. The general form is:

```
if {condition #1}
        {statement #1}
elseif {condition #2}
        {statement #2}
```

```
elseif {conditions #3}
        {statement #3}
else
        {final statement}
end
```

There are several important things to note. There may be many or no *elseif* statements, i.e. an *if* and *else* without the *elseif*'s will work. The *elseif* command is one word. If you write it as two words, it will create a nested *if* statement. The condition that leads to a certain statement can actually be a set of multiple conditions. Finally, the *else* statement does not require a condition. The *else* statement will include all other possibilities not covered in the previous conditions. The *else* statement is also optional. However, if none of the conditions in the *if* and *elseif* statements are met, no output will be returned with the potential of causing errors.

NESTING

It is common to see *if* statements included in loops, so the following example will do so. A good example is a discontinuous function. This sometimes occurs in chemical reactions when a reaction is zero order. (Don't worry about what a zero order reaction is.) It leads to an equation for the concentration of a reactant that is disappearing, we will call A, with respect to time of:

$$C_A = C_{Ao} - kt$$

In this equation, C_A is the concentration, C_{Ao} is the initial concentration, k is a rate constant, and t is the time. If you were to plot this function, you would find that when kt is greater than C_{Ao} the concentration goes negative. This clearly is not possible and is obvious to you when you see it. However, to the computer it is not obvious, so you must make sure it solves it correctly. This is done by adding the stipulation that the above equation is only true when C_{Ao} is greater than or equal to kt. When C_{Ao} is less than or equal to kt, C_A is equal to zero. Here is how you would write the m-file, if C_{Ao} and the final time were inputs:

```
function [CA]=zeroorder(CAo,t)
k=5; % this is value of the rate constant for our particular reaction
for tt=0:t
        if tt<=CAo/k
                CA(tt+1)=CAo-k*tt;
        else
                CA(tt+1)=0;
        end
end
time=0:t;
plot(time,CA);
```

This m-file will plot the function correctly as evidenced by the generated plot. Try running the program with C_{Ao} = 20 and t = 10. Notice how the index was changed in the definition of C_A to avoid referencing the 0^{th} element. Also notice how the time had to be redefined.

When loops and *if* statements are nested together, the inner most loop or *if* statement will be computed to completion before returning to the next outer loop.

Another example of nesting is finding the maximum of a matrix. Since there are multiple dimensions, multiple *for* loops must be used. The matrix we will look at is a series of temperatures, where four temperatures were taken each day for three days. The rows are the set of temperatures for a day and the columns are the four different times the temperatures were taken:

```
>>A=[21 35 55 60;22 40 65 64; 20 50 60 61;];
```

Now an m-file can be written to find the maximum:

```
function [Amax]=matrixmax(A)
[m n]=size(A);
Amax=-500;
for ii=1:m
        for jj=1:n
                if A(ii,jj)>Amax
                        Amax=A(ii,jj);
                else
                        Amax=Amax;
                end
        end
end
```

This will find the maximum of the matrix. The *size* function finds the number of rows and columns of the matrix and sets the number of rows to *m* and the number of columns to *n*. It is analogous to the *length* command for vectors. The initial value of the maximum temperature was set to -500, because it is impossible to have a lower temperature in Fahrenheit. This could have been accomplished in other ways. Finally, the ordering of the *for* loops is not important. You can start by analyzing row by row or column by column. Both should give the same answer.

DO IT YOURSELF

By nesting *for* or *while* loops and *if* statements, write an m-file that takes *n* as an input and gives an *n* x *n* identify matrix as an output. The identity matrix is a matrix with a value of one when the number of the row is equal to the number of the column and a value of zero at all other points in the matrix.