6 FOR, IF, WHILE

Contents

Initialization

```
% reduce needless whitespace
format compact
% reduce irritations
more off
% start a diary
%diary lectureN.txt
```

FOR LOOPS

For loops are useful if you want to do the same sort of things multiple or many times.

A very simple loop

```
disp(' ')

disp('Let''s try it!')
for counter=1:3
    disp('Matlab is great!')
end
disp('Done.')
disp('Note how the "execution pointer" has moved!')
```

```
Let's try it!
Matlab is great!
Matlab is great!
Matlab is great!
Done.
Note how the "execution pointer" has moved!
```

A slightly more elaborate version

```
disp(' ')

% set a repeat count
counterMax=5

% CREATE A TEST1.M SCRIPT FOR THE NEXT CODE:

% print out the message
fprintf('Remember fact %i about Matlab:\n',counterMax)
for counter=1:counterMax
    fprintf('%i: Matlab is great!\n',counter)
end
disp('Done.  Try another value for counterMax!')

% Note how Matlab processed those lines.  At the "for"
% command it did *not* set "counter" equal to the vector
% [1 2 3 4 5].  Instead it set counter equal to the first
% number, 1.  Then Matlab went on to the fprint
% statement.  But when it saw the "end" command, it
% jumped back to the "for" command, and set counter equal
% to the second number, 2.  And it repeated these steps
% for 3, 4, and 5.  But when it jumped back to the "for"
% command after the 5, there were no more numbers.  So
% Matlab then jumped past the "end" statement and went on
% with the "disp('done')" and beyond.
```

```
counterMax =  5
Remember fact 5 about Matlab:
1: Matlab is great!
2: Matlab is great!
3: Matlab is great!
4: Matlab is great!
5: Matlab is great!
Done.  Try another value for counterMax!
```

Handle repetitive operations neatly

Remember how messy it was in lesson2 to find and neatly print four frequencies for the flexibly suspended string? With a for loop we can easily find and print 10! Or much more still.

```
disp(' ')
```

```
% define function freqEq, the condensed version
freqEq=@(omega,k) sin(omega) + k*omega*cos(omega);

% set the flexibility
k=1

% set how many frequencies we want to print out
nMax=10

% CREATE A TEST2.M SCRIPT FOR THE NEXT CODE:

% print out the first 10 frequencies
for n=1:nMax
    guess=(n-0.5)*pi;
    omega=fzero(@(omega) freqEq(omega,k),guess);
    fprintf(...
        'Frequency %2i: guess: %6.3f; exact: %6.3f\n',...
        n,guess,omega)
end
disp('Done.  Try another value for nMax!')
```

```
k =  1
nMax =  10
Frequency  1: guess:  1.571; exact:  2.029
Frequency  2: guess:  4.712; exact:  4.913
Frequency  3: guess:  7.854; exact:  7.979
Frequency  4: guess: 10.996; exact: 11.086
Frequency  5: guess: 14.137; exact: 14.207
Frequency  6: guess: 17.279; exact: 17.336
Frequency  7: guess: 20.420; exact: 20.469
Frequency  8: guess: 23.562; exact: 23.604
Frequency  9: guess: 26.704; exact: 26.741
Frequency 10: guess: 29.845; exact: 29.879
Done.  Try another value for nMax!
```

Forming matrices

A `for` loop is often a great way to generate a matrix. For example consider the following 6×6 matrix

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & -3 & 2 & 0 & 0 & 0 \\ 0 & 2 & -5 & 3 & 0 & 0 \\ 0 & 0 & 3 & -7 & 4 & 0 \\ 0 & 0 & 0 & 4 & -9 & 5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{row number} \quad i = \begin{cases} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 = n \end{cases}$$

You might encounter such a matrix in, say solving a problem including both conduction and convection of heat. And it might be much bigger than 6×6; you could easily have n a thousand instead of 6, producing a 1000×1000 matrix, with a million numbers in it. So in general, typing the matrix completely out as written is not a realistic option.

Instead, note that the matrix components have some logic in it. First of all, note that almost all components are zero. So if you start the matrix off as all zeros, like in

```
A = zeros(n)
```

then you get most components correct right off the bat. You now only need to worry about fixing up the nonzero components.

Next note that there is a definite logic in the nonzero coefficients. Or at least there is if you ignore the first row, $i = 1$, and the last row, $i = n$. The intermediate rows, from $i = 2$ to $n - 1$, all have a similar structure. To describe this mathematically, first note that on the "main diagonal", which runs from top left corner to bottom right corner, all components are negative.

What distinguishes this main diagonal mathematically is that on it, the "column number" j equals the row number i. And remember that in Matlab you can address the matrix component with row number i and column number j as `A(i,j)`. Mathematicians would indicate that same component as $a_{i,j}$; in other words they use a lower case letter and subscripts rather than upper case and parentheses. The bottom line is that in Matlab the component on the main diagonal in row i is indicated by `A(i,i)`. In mathematics, that is $a_{i,i}$. Note also that the components $a_{i,i}$ on the main diagonal are all negative.

Next note that the components immediately to the right of the main diagonal have the column number j one greater than i. So these components can be written as $a_{i,i+1}$. These components form what is called the "first superdiagonal". Now note that the *values* of these components are very simple: they are simply equal to the row number i:

$$a_{i,i+1} = i$$

Next note the components immediately to the left of the main diagonal. These components, with column number $j = i - 1$, are called the first subdiagonal. Note that their values are also simple. They are just one smaller than the row number:

$$a_{i,i-1} = i - 1$$

Finally, the components on the main diagonal equal minus the sum of the subdiagonal and superdiagonal components, so

$$a_{i,i} = -(2 * i - 1)$$

We can use the above three formulae to put all the correct nonzero components in the intermediate rows in matrix A. But we will have to do rows 1 and n separately, by looking at the matrix as written out above.

```
disp(' ')

% size of the matrix to create
n=6

% CREATE A TEST3.M SCRIPT FOR THE NEXT CODE:

% initialize the matrix as all zeros
A=zeros(n)

% set the nonzero components of first row i=1
i=1;
A(i,i)=1;
A(i,i+1)=-1;
A=A

% set the nonzero components of the intermediate rows
for i=2:n-1
    A(i,i-1)=i-1;
    A(i,i)=-(2*i-1);
    A(i,i+1)=i;
end
A=A

% set the nonzero components of last row i=n
i=n;
A(i,i)=1;
A=A
fprintf('Done creating matrix A for n = %i.\n',n)
disp('Try another value for n!')
```

```
n =   6
A =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
A =
     1    -1     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
A =
     1    -1     0     0     0     0
     1    -3     2     0     0     0
     0     2    -5     3     0     0
     0     0     3    -7     4     0
     0     0     0     4    -9     5
     0     0     0     0     0     0
A =
     1    -1     0     0     0     0
     1    -3     2     0     0     0
     0     2    -5     3     0     0
     0     0     3    -7     4     0
     0     0     0     4    -9     5
     0     0     0     0     0     1
Done creating matrix A for n = 6.
Try another value for n!
```

Another example matrix, using a nested loop

Remember the following matrix from lesson 5?

$$A_{\text{sing}} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad \text{row number} \quad i = \begin{cases} 1 \\ 2 \\ 3 = n \end{cases}$$

In that lesson we typed the matrix completely out as

```
ASing = [1  2  3;
         4  5  6;
         7  8  9]
```

But with `for` loops, we can create it in a more systematic way that allows bigger matrices like that to be formed.
To do so, first note that in any given row, when the column number j increases by 1, then the corresponding component $a_{i,j}$:

$$a_{i,j} = j + \text{something rather}$$

increases by 1. By looking at the first components of the first few rows, you can quickly identify "something rather": it is zero for row 1, and increases by n each time the row number increases by 1, So "something rather" must be $(i-1)n$, and so:

$$a_{i,j} = j + (i-1)n$$

The new programming thing is that in this case, we must look at not just all possible values of row number i, but also of column number j. So we need what is called a "nested" for loop.

```
disp(' ')

% size of the matrix to create
n=3

% CREATE A TEST4.M SCRIPT FOR THE NEXT CODE:

% create the correct amount of storage for the matrix
disp('Ensure that the size of the matrix is right')
ASing=zeros(n);
% Note: Without the above line, the *first time* around
% things would still work: Matlab would start with a
% zero-size matrix and increase its size as needed each
% time you add an component.  But these size increases
% would be extremely inefficient.  Also, if you
% subsequently tried a smaller value of n, Matlab would
% not reduce the matrix to the new smaller size,
% producing the wrong result.

% loop over the rows
for i=1:n
    % loop over the columns
    for j=1:n
        % give the right value
        ASing(i,j)=j+(i-1)*n;
    end
end

% print it out
ASing=ASing
fprintf('Done creating matrix A for n = %i\n',n)

% check that it is still singular
condASing=cond(ASing)
disp('Yes, still singular.')
disp('Try another value for n!')
```

```
n =   3
Ensure that the size of the matrix is right
ASing =
    1    2    3
    4    5    6
    7    8    9
Done creating matrix A for n = 3
condASing =      6.0262e+16
Yes, still singular.
Try another value for n!
```

Doing sums with a known limit

Let's say that we want to evaluate the sum S given by

$$S = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{1000^2}$$

A `for` loop from 1 to 1000 will evaluate this quite nicely.
First however, we need to write the sum out mathematically with a summation symbol:

$$S = \sum_{i=1}^{i_{\max}} \frac{1}{i^2} \qquad i_{\max} = 1000$$

because that is the way it is programmed.

```
disp(' ')

% set the number of the last term to sum
iMax=1000

% CREATE A TEST5.M SCRIPT FOR THE NEXT CODE:

% initialize the sum to zero (no terms summed yet)
total=0;

% in a for loop from 1 to 1000, add each term in turn
for i=1:iMax
    % the term t(i) to add to sum
    ti=1/i^2;
    % the new value of sum is the previous value plus ti
    total=total+ti;
end
```

```
% print out the obtained sum
total=total
disp('(This should be less than 1.6449)')
disp('Try another value for iMax!')
```

```
iMax =   1000
total =   1.6439
(This should be less than 1.6449)
Try another value for iMax!
```

Summing a Taylor series

Not all mathematical functions are provided by Matlab, or any numerical software, in canned form. When you encounter such a function, one option to evaluate it is to sum its Taylor series. (That assumes that you know the Taylor series, but usually you do. For example, the function might be the integral of a function whose Taylor series you can easily find.)
As an example let's evaluate e^x by summing its Taylor series. (We will ignore the fact that you could get the value in Matlab much more simply as `exp(x)`. Instead we will use `exp(x)` to check the error in our result)
The Taylor series of e^x is according to calculus:

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

Writing this using a summation symbol gives

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Note also that we cannot really sum infinitely many terms. We will have to stop summing at some large value, call it $i_{\max}$ of i.

```
disp(' ')

% set the x value at which we want the Taylor series
x=1

% set the number of terms at which to stop summing
iMax=10

% COPY TEST5.M TO TEST6.M FOR THE NEXT CODE:

% initialize the sum to term t(0)
i=0;
```

```
ti=1;
total=ti;

% loop to add iMax more terms to the sum
for i=1:iMax
    % compute term t(i)
    ti=x^i/factorial(i);
    % add term t(i) to the sum
    total=total+ti;
end

% print out the obtained value
total=total

% see how big the error really is
totalError=total-exp(x)
disp('Done.  Try other values for x and/or iMax!')
```

```
x =   1
iMax =   10
total =   2.7183
totalError =    -2.7313e-08
Done.   Try other values for x and/or iMax!
```

A better way to do the Taylor series

The previous way of doing the Taylor series of e^x is not ideal. For one, evaluating x^i for large values of i is a slow process for Matlab. And so is evaluating $i!$. And far worse than that is that $i!$ will readily overflow for large values of i. And so will x^i if the magnitude of x exceeds 1.
So look once more at that Taylor series:

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{1\,2} + \frac{x^3}{1\,2\,3} + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Note that every term t_i in the sum, except the first, can be computed from the previous term by multiplying that previous term by x/i:

$$t_i = t_{i-1}\frac{x}{i}$$

That avoids overflow and is much more efficient for Matlab too.

```
disp(' ')

% set the x value at which we want the Taylor series
```

```
x=1

% set the number of terms to stop summing
iMax=10

% COPY TEST6.M TO TEST7.M FOR THE NEXT CODE:

% initialize the sum to term t(0)
i=0;
ti=1;
total=ti;

% loop to add iMax more terms to the sum
for i=1:iMax
    % compute term t(i) from the previous value
    ti=ti*x/i;
    % add term t(i) to the sum
    total=total+ti;
end

% print out the obtained value
total=total

% see how big the error really is
totalError=total-exp(x)
disp('Done.  Try other values for x and/or iMax!')
```

```
x =   1
iMax =   10
total =   2.7183
totalError =    -2.7313e-08
Done.  Try other values for x and/or iMax!
```

Save your workspace

```
disp(' ')
disp('Save your workspace!')
disp('And keep the test*.m scripts for now!')
```

```
Save your workspace!
And keep the test*.m scripts for now!
```

IF CONSTRUCTS

An `if` construct is useful if you want to do some things only under specific conditions.

A couple of very simple examples

```
disp(' ')

% CREATE A TEST8.M SCRIPT FOR THE NEXT CODE:

% see whether 1 or 2 is bigger
if 1 > 2
    disp('The class is wrong, 1 is bigger than 2!')
end
if 2 > 1
    disp('The class is right, 2 is bigger than 1!')
end
disp('Done.')
```

```
The class is right, 2 is bigger than 1!
Done.
```

A more sophisticated example

You can do the above much nicer with an

```
if CONDITION1
    DOSOMETHING1
elseif CONDITION2
    DOSOMETHING2
else
    DOSOMETHING3
end
```

Note: You can have more than one `elseif` in a row, or none at all. But you **cannot** have a space between `else` and `if`.

```
disp(' ')

% COPY TEST8.M TO TEST9.M FOR THE NEXT CODE:

% see whether 1 or 2 is bigger
if 1 > 2
    disp('The class is wrong, 1 is bigger than 2!')
```

```
elseif 2 > 1
    disp('The class is right, 2 is bigger than 1!')
else
    disp('The class is wrong, 1 is equal to 2!')
end
disp('Done.')
```

```
The class is right, 2 is bigger than 1!
Done.
```

Relational operators

The standard "relational operators" are

Symbol	Meaning
>	greater
<	less
>=	greater or equal
<=	less or equal
==	equal
~=	not equal

```
disp(' ')

% CREATE A TEST10.M SCRIPT FOR THE NEXT CODE:

% let's compute two numbers that are roughly the same
halfpi=pi/2;
rt2=sqrt(2);

% now see which one is really the biggest
if halfpi > rt2
    disp('pi/2 is greater than sqrt(2)!')
elseif halfpi < rt2
    disp('pi/2 is less than sqrt(2)!')
elseif halfpi==rt2
    disp('pi/2 is equal to sqrt(2)!')
else
    disp('Matlab has gone crazy!')
end
disp('Done.')
```

```
pi/2 is greater than sqrt(2)!
Done.
```

Logical operators

The standard "logical operators" are:

```
Symbol       Meaning
--------------------------------
  ~          logical NOT
  &          logical AND
  |          logical OR
```

There is also XOR, but you rarely need it if you do normal engineering things. The above operators are in order of precedence. Use parentheses as needed to be safe and for readability.
If the & or | is doubled, they become short-circuiting:
For CONDITION1 && CONDITION2, if CONDITION1 is found to be false, Matlab never looks at CONDITION2, because the combined expression is false already.
For CONDITION1 CONDITION2, if CONDITION1 is found to be true, Matlab never looks at CONDITION2 because the combined expression is true already.

```
disp(' ')

% CREATE A TEST11.M SCRIPT FOR THE NEXT CODE:

% we *need* the parentheses below???
if halfpi>1 & halfpi<2 & ~ (halfpi==1.5)
    disp('pi/2 is between 1 and 2 and not 1.5!')
end

% the next might be more readable?
if (halfpi>1) & (halfpi<2) & ~ (halfpi==1.5)
    disp('pi/2 is between 1 and 2 and not 1.5!')
end

% definitely the below is more readable
if (halfpi>1) & (halfpi<2) & (halfpi~=1.5)
    disp('pi/2 is between 1 and 2 and not 1.5!')
end

% recommended by Matlab for if or while:
if (halfpi>1) && (halfpi<2) && (halfpi~=1.5)
    disp('pi/2 is between 1 and 2 and not 1.5!')
```

```
end
```

```
pi/2 is between 1 and 2 and not 1.5!
pi/2 is between 1 and 2 and not 1.5!
pi/2 is between 1 and 2 and not 1.5!
pi/2 is between 1 and 2 and not 1.5!
```

Checking condition numbers

Remember how we had to check the solution of the linear system of equations in lesson5? Now we can do this in a much clearer and better way. In particular, we can avoid wasting time and paper computing a useless solution.

```
disp(' ')

% recreate the system
disp('Let''s redo the solution of the linear equations:')
A = [1 2 3;
      0 5 6;
      7 8 9];
b = [3;
      2;
      9];

% CREATE A TEST12.M SCRIPT FOR THE NEXT CODE:

% check the error in the solution due to Matlab
condA=cond(A)
relErrorMatlab=condA*eps(1)
if relErrorMatlab >= 0.1
    disp('There is no reasonable solution to this system!
        ')
else
    x = A \ b
    if relErrorMatlab > 0.001
        fprintf('Warning: estimated error %1E%%!\n',...
                relErrorMatlab*100)
    end
end
disp('Done.  Try another matrix!')
```

```
Let's redo the solution of the linear equations:
condA =  37.939
```

```
relErrorMatlab =      8.4241e-15
x =
    1
   -2
    2
Done.  Try another matrix!
```

Doing infinite sums to a given accuracy

Earlier in this lesson, we did the sum

$$S = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{i_{\max}^2} = \sum_{i=1}^{i_{\max}} \frac{1}{i^2}$$

to $i_{\max} = 1000$ terms.
This time, however, we would like to see what we get when we sum *infinitely many* terms. But of course, that is not possible. It would take infinitely much time for Matlab to sum infinitely many terms.
Instead what we can do is try to sum to some small remaining error that we are willing to accept. Such an acceptable error is called a "tolerance". For example, in this case we might decide that a remaining error of 0.0001 is tolerable.
To see whether we have reached the tolerance at any given term number i, however, requires that we estimate the error that comes from not summing the remaining terms. Estimating that error can only be an educated guess. (We would only know for sure if we really summed the remaining terms, which is exactly what we cannot do.)
So how should we estimate the remaining error in the sum at any stage in the summing?

1. One way is to simply assume that the magnitude $|t_i|$ of the term currently being added to the sum gives the estimated error. However, using $|t_i|$ as estimated error only works correctly if the sum is "alternating", i.e. the terms t_i change sign all the time.

2. Otherwise it works much better if you assume that the estimated error is about i times the magnitude of the term t_i currently being added to the sum.

Note that a lot of people who should know better simply use $|t_i|$ as estimated error, and *produce completely wrong results*. In this class you *must* use $|it_i|$ as estimated error if it is not an alternating series.
An `if` statement can be used to check whether the estimated error has become smaller than the tolerance. If it has, you can use the "`break`" statement. A `break` statement will terminate the loop that it is in, and with it, any summing done inside it.
In the current example sum, we can check whether we are doing things right because the value of the infinite sum is actually known: it should be $\pi^2/6$.

```
disp(' ')

% the allowed tolerance in value
tol=0.0001

% the maximum number of terms we would *ever* want to sum
iMax=100000

% COPY TEST5.M TO TEST13.M FOR THE NEXT CODE:

% initialize the sum to zero (no terms summed yet)
total=0;

% add terms until it seems accurate but no more than iMax
for i=1:iMax
    % the term t(i) to add to sum
    ti=1/i^2;
    % the new value of sum is the previous value plus ti
    total=total+ti;
    % find the current estimated error the required way
    estError=i*abs(ti);
    % test whether we can stop summing
    if estError <= tol
        % stop summing ("jump out of the for loop")
        break
    end
end

% print out the results
fprintf('The found infinite sum is %.4f\n',total)
exactTotal=pi^2/6;
trueError=abs(total-exactTotal);
fprintf('The exact infinite sum is %.4f\n',exactTotal)
fprintf('The estimated error is %.1E\n',estError)
fprintf('The   true    error is %.1E\n',trueError)

% interpret the results
if estError > tol
    disp('*** Requested accuracy not met, even after')
    fprintf('    summing %i terms!\n',i)
    disp('Try a still larger number of terms?')
elseif trueError > 10*estError
    disp('Maybe your estimated error is no good?')
else
    fprintf('Needed to sum %i terms.\n',i)
    disp('How about trying a bad estimated error?')
    disp('Maybe along with a 1/i sum?')
end
```

```
 tol =     1.0000e-04
 iMax =   100000
 The found infinite sum is 1.6448
 The exact infinite sum is 1.6449
 The estimated error is 1.0E-04
 The   true    error is 1.0E-04
 Needed to sum 10000 terms.
 How about trying a bad estimated error?
 Maybe along with a 1/i sum?
```

Warning!!!

Warning: Students who end up with frozen homework programs, or messages that Java/Adobe is misbehaving have incorrectly implemented the `break` *command, or even omitted it completely.* Trying to `publish` 100,000 message lines is a sure recipe for crashing something. Please check operation of your `break` command *before* seeing TA or instructor. And make sure all semicolons are there.

Taylor series done better

If we want to sum a Taylor series, we probably want the most accurate answer we can possibly get. To achieve this, note that in a convergent Taylor series, eventually the terms become smaller and smaller. Finally they "underflow" and become zero. After that point, it is obviously useless to keep summing. However many times you add zero, it is not going to change the value.
But even when the terms are not yet underflowing, they may be too small to further change the value of the sum. That is because numbers on a computer have round-off errors. As soon as the individual terms in the sum become smaller than the round off error in the accumulated sum, they are already unable to change the sum.
So the smart way to do Taylor series is to keep summing until you have ensured that the sum can no longer change. Let's try it for e^x:

```
disp(' ')

% the x value at which we want the Taylor series
x=1

% the maximum number of terms we would *ever* want to sum
iMax=100000
```

```
% COPY TEST7.M TO TEST14.M FOR THE NEXT CODE:

% initialize the sum to term t(0)
i=0;
ti=1;
total=ti;

% loop to add up to iMax more terms to the sum
for i=1:iMax
    % remember the last value of sum
    totalOld=total;
    % compute term t(i) from the previous value
    ti=ti*x/i;
    % add term t(i) to the sum
    total=total+ti;
    % see whether we can stop
    if total==totalOld
  break
    end
end
if total==totalOld
    fprintf('Converged after %i terms.\n',i)
else
    fprintf('*** Not converged after %i terms!\n',i)
end

% print out the obtained value
total=total

% see how big the error really is
totalError=total-exp(x)
disp('Done.  Try other values of x, like negative ones!')
```

```
x =   1
iMax =   100000
Converged after 18 terms.
total =   2.7183
totalError =     4.4409e-16
Done.  Try other values of x, like negative ones!
```

WHILE LOOPS

The `while` command is similar to the `for` command in that it loops. However, `while` does not perform a given number of loops. Instead `while` stays looping as long as some condition remains true. The `while` command can be appropriate in cases where you have no clue when looping will stop. (But I disagree with the preference the online book displays for while statements. In most cases `for` is the better choice. For one, a `while` loop can stay looping forever if something goes wrong.)

Getting input from a script user using while

Let's keep looping until the user admits that Matlab is great.

```
disp(' ')

% CREATE A TEST15.M SCRIPT FOR THE NEXT CODE:

% get the user's name
name=getenv('USER');      % Unix version
%name=getenv('USERNAME') % DOS version
% in the script, replace the above lines by
%name=input('Please enter your name: ','s');

% define a menu header
header=[name ' admits that:'];

% loop until we get the right answer
choice=0;
while choice~=4
    choice=4;
    % in the script, replace the above line by
    %choice=menu(header,...
    %           'Matlab is horrible.',...
    %           'Matlab is too much work.',...
    %           'Matlab is OK.',...
    %           'Matlab is great!')
    header='Wrong answer.  Try again:';
end
```

Doing a sum with a while loop

You can do with `while` loops whatever you can do with `for` loops. For example, we can evaluate the Taylor series for exp(x) using a `while` loop as shown below. It works just like the earlier `for` loop.

```
disp(' ')

% the x value at which we want the Taylor series
x=1

% the maximum number of terms we would ever want to sum
iMax=100000

% COPY TEST14.M TO TEST16.M FOR THE NEXT CODE:

% initialize the previous value of the sum to infinity
totalOld=Inf;

% initialize the sum to term t(0)
i=0;
ti=1;
total=ti;

% in a while loop, add terms until the sum stops changing
while total ~= totalOld
    % remember the last value of sum
    totalOld=total;
    % each time through, increase the i value by one
    i=i+1;
    % compute term t(i) from the previous value
    ti=ti*x/i;
    % add term t(i) to the sum
    total=total+ti;
    % stop if it takes too many terms
    if i >= iMax
        break
    end
end
if total==totalOld
    fprintf('Converged after %i terms.\n',i)
else
    fprintf('*** Not converged after %i terms!\n',i)
end

% print out the obtained value
total=total

% see how big the error really is
totalError=total-exp(x)
disp('Done.  Try other values of x, like negative ones!')
```

```
x =  1
iMax =  100000
Converged after 18 terms.
total =  2.7183
totalError =    4.4409e-16
Done.  Try other values of x, like negative ones!
```

End lesson 6