6 FOR, IF, WHILE

Contents

Initialization

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

% ***USE testN.m SCRIPTS!***
```

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('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

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

```
n =  5
Remember these 5 facts about Matlab:
1: Matlab is great!
2: Matlab is great!
3: Matlab is great!
4: Matlab is great!
5: Matlab is great!
Done
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" command
and beyond.
```

This is great!

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

```
% define function freqEq, the condensed version
freqEq=@(omega,k) sin(omega) + k*omega*cos(omega);
% set the flexibility
k=1
% print out the first 10 frequencies
for n=1:10
    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
```

```
k =     1
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
```

Forming matrices

Remember the following matrix from lesson 5?

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

With a `for` loop, we can create it in a more systematic way that allows bigger matrices like that to be formed.

```
% size of the matrix
n=3
% create storage for the matrix
A=zeros(n);
% loop over the rows
```

```
for i=1:n
    % loop over the columns
    for j=1:n
        % give the right value
        A(i,j)=j+(i-1)*n;
    end
end
% print it out
A
% check that it is still singular
condA=cond(A)
```

```
n =   3
A =
     1     2     3
     4     5     6
     7     8     9
condA =      6.0262e+16
```

Note

Without the `A=zeros(n)` line above, and no existing matrix `A`, Matlab would on the first time in the loop reach the line `A(1,1)=1`. Since it has no matrix A, Matlab would then create storage for a matrix `A` of size 1 by 1. Then after reaching the `end` of the `for j` loop, it would return to `for j`, set `j` to its second value 2, which makes the next line it sees `A(1,2)=2`. Since this cannot be stored in the 1 by 1 matrix it has created, Matlab would then create storage for a bigger 1 by 2 matrix A and give it the two values 1 and 2, deleting the old 1 by 1 matrix A. And so on. After `j` has reached it final value `n`, Matlab would reach the second `end`, the one that ends the `for i` loop. It would then return to the `for i` and give `i` its second value 2. Next it gives `j` again its starting value 1, and then it would see the line `A(2,1)=n+1`. Since that would not fit inside the 1 by `n` matrix it has, it would delete that matrix after creating a new `2` by `n` matrix. All this creating and deleting matrices is very inefficient. It is much better to force Matlab to make matrix A the correct size immediately.
Also, without creating a new matrix A using `A=zeros(n)` the matrix would not shrink if we made `n` smaller. Matlab will make matrices bigger if it needs more storage locations, but it will not make matrices smaller by itself.

Try a bigger matrix like that

```
n=5
A=zeros(n);
for i=1:n
    % loop over the columns
    for j=1:n
        % give the right value
        A(i,j)=j+(i-1)*n;
    end
end
A
condA=cond(A)
disp('Yes, this bigger matrix is singular too.')
```

```
n =   5
A =
      1     2     3     4     5
      6     7     8     9    10
     11    12    13    14    15
     16    17    18    19    20
     21    22    23    24    25
condA =    8.3563e+17
Yes, this bigger matrix is singular too.
```

Doing sums

Let's say that we want to evaluate the sum

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

A `for` loop from 1 to 1000 will do it quite nicely. Note that term number `i` in the sum equals `1/i^2`.

```
% initialize the total of the terms summed so far to zero
total=0;
% in a for loop from 1 to 1000, add each term in turn
for i=1:1000
    total=total+1/i^2;
end
% print out the obtained sum
sum=total
disp('(It should be less than 1.6449)')
```

```
sum =   1.6439
(It should be less than 1.6449)
```

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^1 by summing its Taylor series. (We will ignore the fact that you could get the value much more simply as `exp(1)`.)
The Taylor series of e^x is according to calculus:

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

So term number `i` in the sum is `x^i/factorial(i)`. And there is in addition a starting "term 0" that is `1`.

```
% the x value at which we want the Taylor series
x=1
% initialize the total of the terms so far to term 0
total=1;
% loop to add 100 more terms to total
for i=1:100
    % add term number i to the total
    total=total+x^i/factorial(i);
end
% print out the obtained value
expValue=total
% see how big the error really is
expError=exp(x)-total
```

```
x =   1
expValue =   2.7183
expError =    -4.4409e-16
```

A better way to do the Taylor series

The previous way of doing the Taylor series of e^x is not ideal. For one, to evaluate `x^i` requires (in the simplest case) that Matlab does `i-1` multiplications $x * x * x * \ldots * x$. Similarly, finding `factorial(i)` requires `i-1` multiplications. That is a lot of multiplications for Matlab to do when `i` becomes larger. Not to mention that `factorial(i)` "overflows" (becomes too big to store) for `i` greater than 170. Similarly, if `x` was `10` instead of `1`, `x^i` would overflow at `i` equal to 307. The summing would crash. That is why above we took the maximum value of `i` equal to 100 instead of, say, 1000.
Look once more at that Taylor series:

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

Note that in every case term `i` equals the previous term times `x/i`. Evaluating term `i` that way requires just one multiplication and one division. That is a lot better than 2 times `i-1` multiplications and a division. *And* it will no longer produce infinite values. To do the sum this way does require that we store the successive terms in a variable, which we will call `term`.

```
% the x value at which we want the Taylor series
x=1
% initialize term 0
term=1;
% initialize the total of the terms so far to term 0
total=term;
% the number of terms we can do can now be much bigger
imax=10000;
% in a for loop from 1 to imax, add imax more terms
for i=1:imax
    % compute the new term to add from the previous one
    term=term*x/i;
    % add it to the total
    total=total+term;
end
% print out the obtained value
expValue=total
% see how big the error really is
expError=exp(x)-total
```

```
x =   1
expValue =  2.7183
expError =    -4.4409e-16
```

IF CONSTRUCTS

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

A couple of very simple examples

```
disp('Let''s try it!')
if 1 > 2
    disp('Hey, one is greater than two!')
end
if 2 > 1
    disp('Hey, two is greater than one!')
end
disp('Done.')
```

```
Let's try it!
Hey, two is greater than one!
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`.

```
% try it
disp('Let''s try it!')
if 1 > 2
    disp('Hey, one is greater than two!')
elseif 2 > 1
    disp('Hey, two is greater than one!')
else
    disp('Hey, one and two are equal!')
end
disp('Done.')
```

```
Let's try it!
Hey, two is greater than one!
Done.
```

Relational operators

The standard "relational operators" are

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

```
% Let's try it
disp('Let''s try it!')
% let's compute two numbers that are roughly the same
halfpi=pi/2;
rt2=sqrt(2);
if halfpi > rt2
    disp('Hey, pi/2 is greater than sqrt(2)!')
elseif halfpi < rt2
    disp('Hey, pi/2 is less than sqrt(2)!')
elseif halfpi==rt2
    disp('Hey, pi/2 is equal to sqrt(2)!')
else
    disp('Matlab has gone crazy!')
end
disp('Done.')
```

```
Let's try it!
Hey, 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.

```
% let's try it
disp('Let''s try it!')
% 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
```

```
Let's try it!
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!
```

This is great!

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.

```
% 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];
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
        disp('Warning: the above solution may have
            significant error!')
    end
end
disp('Let''s redo the singular equations too:')
A(2,1)=4;
condA=cond(A);
relErrorMatlab=condA*eps(1)
if relErrorMatlab >= 0.1
    disp('There is no reasonable solution to this system!
        ')
else
    x = A \ b;
    x'
    if relErrorMatlab > 0.001
```

```
        disp('Warning: the above solution may have
            significant error!')
    end
end
```

```
Let's redo the solution of the linear equations:
relErrorMatlab =     8.4241e-15
x =
    1
   -2
    2
Let's redo the singular equations too:
relErrorMatlab =   13.381
There is no reasonable solution to this system!
```

Use it also in summing

Earlier in this lesson, we did the sum

$$\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots$$

to one thousand terms. This time, however, we would like to check that if we sum infinitely many terms, we really get $\pi^2/6$. But of course, that is not possible. It would take infinitely much time for Matlab to sum infinitely many terms. And then there is round-off errors.
Realistically, the best that we can do is check that if we sum enough terms we can get $\pi^2/6$ to a "tolerated" error of say 0.0001 maximum. We can do that if we put an `if` statement in the `for` loop that terminates the loop when the estimated error in doing so is smaller than the tolerance 0.0001. To terminate a loop, use the `break` command.
For now, let's assume that we can stop when the next term to add is less than the tolerance. In other words, let us estimate the error in terminating the sum as being the first neglected term.

```
% the allowed tolerance in value
tol=0.0001
% initialize the total of the terms summed so far to zero
total=0;
% the maximum number of terms we would ever want to do
imax=100000;
% in a for loop from 1 to imax, add each term in turn
for i=1:imax
    % compute the new term
    term=1/i^2;
    % stop the for loop if it seems small enough
    if term < tol
```

```
        fprintf('Stopped summing after %i terms.\n',i-1)
        break
    end
    % otherwise add it to the total and keep going
    total=total+1/i^2;
end
% print out the sum
sum=total
% print out the actual error
actualError=pi^2/6-sum
% check that we are about as close as expected
if abs(actualError) > 5*tol
    disp('Oops!  Nowhere close!')
end
```

```
tol =      1.0000e-04
Stopped summing after 100 terms.
sum =  1.6350
actualError =  0.0099502
Oops!  Nowhere close!
```

Oops!

Estimating the actual error as the first neglected term was a not a good approximation. The reason is that we did not just ignore `1/101^2`, but also `1/102^2`, `1/103^2`, ... The combined sum is much bigger than just 1/101^2.
Note that if this was an "alternating" series, whose terms are alternately positive and negative, we would not have this problem, and what we did would have worked fine.
But in this case, the terms are all positive. We can make a crude correction for the accumulation of terms if we estimate the error not just as the first neglected term, but as *the term number* `i` *times* the first neglected term. Let's try that:

```
% the allowed tolerance in value
tol=0.0001
% initialize the total of the terms summed so far to zero
total=0;
% the maximum number of terms we would ever want to do
imax=100000;
% in a for loop from 1 to imax, add each term in turn
for i=1:imax
    % compute the new term
    term=1/i^2;
    % stop the for loop if it really seems small enough
    if term*i < tol
        fprintf('Stopped summing after %i terms.\n',i-1)
        break
    end
    % otherwise add it to the total and keep going
    total=total+1/i^2;
end
% print out the sum
sum=total
% print out the actual error
actualError=pi^2/6-sum
% check that we are about as close as expected
if abs(actualError) > 5*tol
    disp('Oops!  Nowhere close!')
end
```

```
tol =     1.0000e-04
Stopped summing after 10000 terms.
sum =  1.6448
actualError =     9.9995e-05
```

Taylor series done more efficiently

If we want to sum a Taylor series, we probably want the most accurate answer we can possibly get. To do so notice that in a convergent Taylor series, the terms become smaller and smaller. Eventually 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 the sum no longer changes. Let's try it:

```
% the x value at which we want the Taylor series
x=1
% initialize term 0
term=1;
% initialize the total of the terms so far to term 0
total=term;
% the maximum number of terms we would ever want to do
imax=100000;
% in a for loop from 1 to imax, add up to imax more terms
for i=1:imax
    % compute the new term to add from the previous one
    term=term*x/i;
    % add it to the total
    oldtotal=total;
    total=total+term;
    % stop if there is no longer a change
    if total == oldtotal
        fprintf('Done summing at term %i.\n',i)
        break
    end
end
% print out the obtained value
expValue=total
% see how big the error really is
expError=exp(x)-total
```

```
x =  1
Done summing at term 18.
expValue =  2.7183
expError =   -4.4409e-16
```

WHILE LOOPS

The `while` command is similar to the `for` command in that it loops, but it stays looping as long as some condition remains true. It can be appropriate in cases where you have no clue when looping will stop.

A simple example

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

```
% get the user's name
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=menu(header,...
                'Matlab is horrible.',...
                'Matlab is too much work.',...
                'Matlab is OK.',...
                'Matlab is great!')
    header='Wrong answer.  Try again:';
end
```

```
% run lesson6a.m
%lesson6a
```

Doing the 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(1) using a `while` loop as shown below. It works just like the `for` loops.

```
% the x value at which we want the Taylor series
x=1
% initialize term 0
term=1;
% initialize the total of the terms so far to term 0
total=term;
% the maximum number of terms we would ever want to do
imax=100000;
% initialize a counter of how many terms we have added
i=0;
% in a while loop, add up to imax more terms
while total ~= oldtotal
    % each time through, increase the i value by one
    i=i+1;
    % compute the new term to add from the previous one
    term=term*x/i;
    % add it to the total
    oldtotal=total;
    total=total+term;
    % stop if it takes too many terms
    if i >= imax
        fprintf('Must stop summing after %i terms.\n',i)
        break
    end
end
% print out the obtained value
expValue=total
% see how big the error really is
expError=exp(x)-total
```

```
x =  1
expValue =  2.7183
expError =    -4.4409e-16
```

End lesson 6