6 FOR, IF, WHILE

Contents

```
% make sure the workspace is clear
if ~exist('____code____','var') ; clear ; end
% reduce needless whitespace
format compact
% reduce irritations (pausing and buffering)
more off
% start a diary (in the actual lecture)
%diary lectureN.txt
```

LESSON SUMMARY

This lesson is about programming. Programming includes using simple user-interaction functions like `disp` (displays text), `error` (same, and stops processing), `return` (stops processing the current script), `break` (stops processing the current `for` or `while` loop), `input` (gets data from the user), and `menu` (asks the user to choose an item from a menu). In addition to these elementary tasks, there are three different key programming constructs covered in this lesson.

The first key programming construct is the "`for` loop." A `for` loop allows you to do the same things for a number of different values of a counter. The generic form is:

```
for COUNTER = START:STEP:END
    THINGS_TO_DO
end
```

- `for` loops are a very good way to form matrices that have some systematic structure. You can address the different rows of the matrix using a `for` loop whose counter values are the row numbers. If needed, inside this `for` loop you can put another `for` loop whose counter values are the column numbers.
- `for` loops are also a very good way to do sums. You initialize the sum to zero, or to the first one or more terms. Then in a `for` loop you add the remaining terms. The counter of the `for` loop will be the term number. (Its values will depend on exactly how you decide to number the terms.)

- If the sum has infinitely many terms, like for a Taylor series, it would take infinitely long time for Matlab to do it. So you will need to restrict the summation `for` loop to an ending term number that is small enough that the summation does not take more time than you can reasonably afford to wait.

- In addition, while doing Taylor series, usually you should evaluate the successive terms by modifying the previous term. This can allow Matlab to evaluate the terms quicker while at the same time avoiding overflow and underflow problems.

The second key programming construct is an "`if`" statement. It allows you to do things, or not do things, depending on whether a certain condition is true. The simplest `if` statement takes the generic form

```
if CONDITION
   THINGS_TO_DO_IF_CONDITION_IS_TRUE
end
```

A more advanced construct is a compound `if` statement. This takes the general form

```
if CONDITION1
   THINGS_TO_DO_IF_CONDITION1_IS_TRUE
elseif CONDITION2
   THINGS_TO_DO_IF_CONDITION1_IS_FALSE
                _BUT_CONDITION2_IS_TRUE
elseif CONDITION3
   THINGS_TO_DO_IF_CONDITION1_IS_FALSE
                _AND_CONDITION2_IS_FALSE
                _BUT_CONDITION3_IS_TRUE
...
else
   THINGS_TO_DO_IF_ALL_CONDITIONS_ARE_FALSE
end
```

Do *NOT* put a space between `else` and `if`.

- Typical conditions in `if` statements include comparisons of numbers using < (less), > (greater), <= (less or equal), >= (greater or equal), == (equal), and ~= (not equal). You can also use the existence of a variable or file as a condition using the `exist` function, like in the initialization of this lesson.

- You can also combine simpler conditions into more complex ones using && (AND) and || (OR), and/or require that a condition is *not* true using ~ (NOT).

- Applications of `if` statements to the material of previous lessons include having Matlab autmatically check whether a condition number of a system of equations is excessive.

- Applications of `if` statements to the material of this lesson include having Matlab terminate summing an infinite sum as soon as the sum seems to be as accurate as you need it to be. If the terms in the sum alternate in sign, terminate the sum at term number n when the magnitude $|t_n|$ of that term reaches the tolerable error. Otherwise terminate when $n|t_n|$ reaches the tolerable error. You will need to use the `break` statement to stop the summing.

The third key programming construct is the "`while` loop." A `while` loop allows you to keep doing the same things as long as a condition is true. The generic form is:

```
while CONDITION
   THINGS_TO_DO
end
```

- Note that whatever you can do with a `while` loop, you can also do, and normally better, using a `for` loop. Unlike what you might conclude from the online book, the main purpose of `while` loops is to allow computer scientists to write impenetrable and poorly documented code.

- However, conceivably you might have a probem in which no reasonable counter can be identified. User interactions can be of that form. Under those circumstances, a properly commented `while` loop might produce more readable and understandable code.

Key areas of the online book

Before the first lecture, in the online book do:

- 3.5 Basic input - The input function: all.

- 11.1 While loops: skip CA 11.1.2-end.

Before the second lecture, in the online book do:

- 10.1 If-else statement: all.

- 10.2 Relational operators: skip CA 10.2.3-end.

- 10.3 Multiple branches: skip CA 10.3.2-end.

- 10.4 Logical operators: skip CA 10.4.3-end.

FOR LOOPS

The first key programming construct to discuss is the "`for` loop." A `for` loop allows you to do the same things for a number of different values of a counter. The generic format is:

```
for COUNTER = START:STEP:END
   THINGS_TO_DO
end
```

A very simple example loop

Enter the next code interactively and see what Matlab does. Note that all that the `disp` statement does is show the quoted text on the screen. It acts as a simplified `fprintf` command.

```
% number of times we want to run the for loop
counterMax=3

% run a for loop for counter = 1, 2, ..., counterMax
for counter=1:counterMax
    disp('Matlab is great!')
end
```

```
counterMax =  3
Matlab is great!
Matlab is great!
Matlab is great!
```

Examine exactly what Matlab does

To follow more closely exactly what Matlab does, first put the below code in a script `test1.m`. Then use the "Breakpoint" edit toolbar button to set a break point just before the first `fprintf` statement. Run the script to get to the break point, then use the "Step" button to see how Matlab processes the rest of the script.

```
% print out a leading message
fprintf('Remember %i facts about Matlab:\n',counterMax)
% run a for loop for counter = 1, 2, ..., counterMax
for counter=1:counterMax
    fprintf('%i: Matlab is great!\n',counter)
end
% print out a trailing message
disp('Done.  Try another value for counterMax!')
```

```
Remember 3 facts about Matlab:
1: Matlab is great!
2: Matlab is great!
3: Matlab is great!
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]. Instead it set `counter` equal to the first number, 1. Then Matlab went on to the `fprintf` statement in the loop. After that, when it saw the `end` command, it jumped back to the `for` command. It set `counter` equal to the second number, 2, and then repeated the `fprintf` inside the loop. And it repeated all this once more for the final number 3. But when it jumped back to the `for` command after that, there were no more numbers for `counter`. So Matlab then jumped forward past the `end` statement and went on with the final `disp` statement.

Handle repetitive operations easily

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. As before, we will number the successive frequencies with a variable called `n`. We will also use `n` as our `for` loop counter.

Of course we will need the function `freqEqError` again. To keep it simple, we will make this now a handle to an anonymous function.

Also we need to set the stiffness of the attachment point and the number of frequencies to print:

```
% define freqEqError as a handle to an anonymous function
freqEqError=@(omega,k) sin(omega) + k*omega*cos(omega);

% set the stiffness
k=2

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

```
k =  2
nMax =  10
```

Next create a `test2.m` script containing the next code and run it. Feel free again to use debug to see more closely what happens.

```
% print out the first nMax frequencies
for n=1:nMax
    % our old guess for frequency omega_n
    omegaGuess_n=(n-0.5)*pi;
```

```
    % the range in which to find frequency omega_n
    omegaRange_n=[omegaGuess_n omegaGuess_n+pi];
    % get the accurate frequency from fzero
    omega_n=fzero(@(omega) freqEqError(omega,k),...
                  omegaRange_n);
    % print it out
    fprintf(...
        'Frequency %2i: guess: %6.3f; exact: %6.3f\n',...
        n,omegaGuess_n,omega_n)
end
```

```
Frequency  1: guess:  1.571; exact:  1.837
Frequency  2: guess:  4.712; exact:  4.816
Frequency  3: guess:  7.854; exact:  7.917
Frequency  4: guess: 10.996; exact: 11.041
Frequency  5: guess: 14.137; exact: 14.172
Frequency  6: guess: 17.279; exact: 17.308
Frequency  7: guess: 20.420; exact: 20.445
Frequency  8: guess: 23.562; exact: 23.583
Frequency  9: guess: 26.704; exact: 26.722
Frequency 10: guess: 29.845; exact: 29.862
```

Without the `for` loop, we would have had to write out the inside of the loop for ten different values of n. That would be a lot of typing. And we would need to make the script even bigger if we wanted a larger number of frequencies, like 20 say. Now we only need to change `nMax` to 20 and rerun the script. Try it!

Forming matrices

A `for` loop is usually a great way to generate big matrices. If we call the number of rows (and columns) in the matrix n, then `n` will be a large number in typical applications. But to keep it readable, in our example below, we will take the value of n initially just 6:

```
% size of the square matrix
n=6
```

```
n =  6
```

However, you should definitely try larger values of n too and see what happens. Next let's assume that the matrix of interest has a structure that for $n = 6$ looks like:

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

where i is our name for the row number. (Row numbers are commonly called i and column numbers j.) You might encounter a matrix like the one above in, say solving a problem including both conduction and convection of heat. But in such an application, you might want to take the number of rows n say 1000, rather than 6. That would then produce 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 to them. First of all, note that almost all components are zero. So if you start the matrix off as all zeros, like in (put this and the following code in a script `test3.m`):

```
% initialize matrix A to all zeros
A = zeros(n)
```

```
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
```

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

Next note from the $n = 6$ example,

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

that there is a definite logic to 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, row numbers $i = 2$ to $n - 1$, all have a similar structure. To describe this mathematically, first note that on the "main diagonal", which runs from the top left corner to the 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 a and subscripts rather than an upper case A and parentheses. The bottom line is that in Matlab the component on the main diagonal in row i is indicated by `A(i,i)`. Note: that is (i,i), not (1,1) as it looks like on my screen in low resolution! In mathematics, it is written as is $a_{i,i}$. For our particular example matrix, 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". Also note from the example matrix above, 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 that the components immediately to the left of the main diagonal have column number $j = i - 1$. These components, which can be written as $a_{i,i-1}$, form what is called the "first subdiagonal". Note from the example matrix above that the values of these components are again simple. They are just one smaller than the row number:

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

Finally, note that the components on the main diagonal equal minus the sum of the subdiagonal and superdiagonal components. So

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

We can use the above three formulae to put all the correct nonzero components in the intermediate rows in matrix A. All it needs is a `for` loop with limits `i = 2` to `i = n-1`:

```
% 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
% show the matrix after this
A
```

```
A =
     0     0     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
```

Now the only thing left to do is also fix up rows 1 and n. That is easy after looking at the example matrix as written out earlier:

```
% set the nonzero components of first row i=1
i=1;
A(i,i)=1;
A(i,i+1)=-1;
% show the matrix after this
A
% set the nonzero components of last row i=n
i=n;
A(i,i)=1;
% show the final matrix after this
A
```

```
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
```

Note that it is more readable to process the rows in the normal order. To do so, you should move setting the first row $i = 1$ before the `for` loop on the intermediate rows.

Now try running `test3`, and check that you get the given matrix. Then run `test3` again for smaller and larger values of n and check that you get smaller and larger versions of the same matrix.

Another example matrix, now requiring a nested loop

Remember the following bad (singular) matrix from lesson 5?

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

In that lesson we typed the matrix completely out as

```
ABad = [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 $a_{i,j}$ increases by 1. So apparently, the mathematical expression for the components is of the form:

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

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

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

The new programming concept is that in this case, we need to set components not just for all possible values of row number i, but also for all possible values of column number j. This can be done with what is called a "nested" `for` loop; in particular it can be done by a `for` loop on column number `j` *inside* a `for` loop on row number `i`.

First however, set the desired size of the matrix to create.

```
% size of the matrix to create
n=3
```

```
n =  3
```

The code to create the matrix, with its nested `for` loops, is as shown below. Put this code in a script `test4.m` and then run it:

```
% initialize the matrix to the correct size, wrong values
ABad=zeros(n);
% process all row numbers i
for i=1:n
    % set the values for all column numbers j in row i
    for j=1:n
        % set the right value of the component
        ABad(i,j)=j+(i-1)*n;
```

```
    end
end
% print out the result
fprintf('Done creating matrix ABad for n = %i:\n',n)
ABad
% check whether it is singular
condABad=cond(ABad)
```

```
Done creating matrix ABad for n = 3:
ABad =
   1   2   3
   4   5   6
   7   8   9
condABad =    6.0262e+16
```

Also try running `test4` for smaller and larger values of n and check that the matrix is singular for all values of n greater than 2.

Doing sums with a known limit

A `for` loop is also a great way to do sums. For example, suppose 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 with a term number `n` from 1 to 1000 will evaluate this quite nicely.

First however, we need to write the sum out mathematically with a summation symbol:

$$S = \sum_{n=1}^{n_{\max}} t_n \qquad t_n = \frac{1}{n^2} \qquad n_{\max} = 1000$$

because that is the way it is programmed.

Now first set the number of terms to sum:

```
% the last term to sum
nMax=1000
```

```
nMax =  1000
```

Next put the following code in a script `test5.m` and run it. (Note that we call the sum `total` rather than `sum`. The reason is that the name `sum` is already used for something else in Matlab.)

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

```
% in a for loop from 1 to nMax, add each term in turn
for n=1:nMax
    % compute term t_n
    t_n=1/n^2;
    % add term t_n to the sum
    total=total+t_n;
end

% print out the obtained sum
total
```

```
total =  1.6439
```

As a sanity check, this should be somewhat less than the value of the infinite sum, which is 1.6449.

Also try a bigger value for nMax and see what happens.

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 a very simple 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:

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

Writing this using a summation symbol gives

$$e^x = \sum_{n=0}^{\infty} t_n \qquad t_n = \frac{x^n}{n!}$$

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

As an example, we will initially take x equal to 1 and $n_{\max}$ equal to 10:

```
% set the x value at which to do the Taylor series
x=1
% set the number of the term at which to stop summing
```

```
nMax=10
```

```
x =  1
nMax =  10
```

Next put the following code in a script `test6.m` and run it. (Save `test5.m` as `test6.m` and then modify that.)

```
% initialize the sum to term t_0 = 1
n=0;
t_n=1;
total=t_n;

% loop to add nMax more terms t_1, t_2, ... to the sum
for n=1:nMax
    % compute term t_n
    t_n=x^n/factorial(n);
    % add term t_n to the sum
    total=total+t_n;
end

% print out the obtained value
total

% see how big the error really is
totalError=total-exp(x)
```

```
total =  2.7183
totalError =   -2.7313e-08
```

Note that we did not initialize `total` to zero but to the first term t_0 in the sum, which is 1. Then we started the `for` loop at $n = 1$ instead of $n = 0$. Doing this avoids a problem when $x = 0$. Can you see what problem that is?

Next try a smaller value for $n_{\max}$, like 5, and see what happens to the error. Reset $n_{\max}$ to 10 and try different values for x, like $x = 0.5$ and $x = 2$. Then try $x = 10$ and note that now there is almost 100% error. To fix this, increase $n_{\max}$ to 30. Then try $x = -10$ and note that now again there is almost 100% error.

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^n for large values of n is a slow process for Matlab. And so is evaluating $n!$.

And far worse than that is that $n!$ will readily overflow for large values of n (above $n = 170$ in Matlab). And so will x^n if the magnitude of x exceeds 1.

So look once more at that Taylor series:

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

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

$$t_n = t_{n-1}\frac{x}{n}$$

That avoids overflow and is much more easy to compute for Matlab too.

While we cannot evaluate the first term t_0 this way, we were already summing term t_0 separately anyway.

Let's try it. First reset the values of x and $n_{\max}$ to what they were earlier:

```
% set the x value at which to do the Taylor series
x=1
% set the number of the term at which to stop summing
nMax=10
```

```
x =  1
nMax =  10
```

Next put the following code in a script `test7.m` and run it. (Save `test6.m` as `test7.m` and then modify that.)

```
% initialize the sum to term t_0 = 1
n=0;
t_n=1;
total=t_n;

% loop to add nMax more terms t_1, t_2, ... to the sum
for n=1:nMax
    % compute term t_n from the previous one
    t_n=x/n*t_n;
    % add term t_n to the sum
    total=total+t_n;
end

% print out the obtained value
total

% see how big the error really is
totalError=total-exp(x)
```

```
total =  2.7183
```

```
totalError =    -2.7313e-08
```

SAVE YOUR WORKSPACE

At the end of the first lecture that covers this lesson, be sure to save your workspace:

```
save lectureN
```

and keep the `test....m` scripts for now.

Then at the start of the second lecture, you can use the

```
load lectureN
```

command to continue right where you left off.

IF CONSTRUCTS

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

A couple of examples of simple if statements

A simple `if` statement takes the generic form:

```
if CONDITION
   THINGS_TO_DO_IF_CONDITION_IS_TRUE
end
```

For example, assuming that 1 and 2 are different numbers, which one is greater? You say 2? Well, let's see whether Matlab agrees with you, shall we?

Create a script `test8.m` containing the following code and run it:

```
% if 1 is bigger than 2, tell the class that it is wrong
if 1 > 2
    disp('The class is wrong, 1 is bigger than 2!')
end
% if 2 is bigger than 1, tell the class that it is right
if 2 > 1
    disp('The class is right, 2 is bigger than 1!')
end
disp('Good that that is settled.')
```

```
The class is right, 2 is bigger than 1!
Good that that is settled.
```

An example of a compound if statement

The most general compound `if` statement takes the generic form:

```
if CONDITION1
   THINGS_TO_DO_IF_CONDITION1_IS_TRUE
elseif CONDITION2
   THINGS_TO_DO_IF_CONDITION1_IS_FALSE
                _BUT_CONDITION2_IS_TRUE
elseif CONDITION3
   THINGS_TO_DO_IF_CONDITION1_IS_FALSE
                _AND_CONDITION2_IS_FALSE
                _BUT_CONDITION3_IS_TRUE
...
else
   THINGS_TO_DO_IF_ALL_CONDITIONS_ARE_FALSE
end
```

Note: You can have more than one `elseif` part in a row, or none at all. And you can leave the final `else` part away. But you *cannot* have a space between `else` and `if`. That would make the `if` part a part of THINGS_TO_DO, which is different.

As an example, let's do a bit better job in comparing 1 and 2. Put the next code in a script `test9.m`. (Save `test8.m` as `test9.m` and then modify that.)

```
% 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('Good that that is settled.')
```

```
The class is right, 2 is bigger than 1!
Good that that is settled.
```

Relational operators

The standard "relational operators" are

Symbol	Meaning
>	greater
<	less

```
>=      greater or equal
<=      less or equal
==      equal
~=      not equal
```

Note that to check equality, you need two equal signs. A single equals sign would be an assignment statement, not a test for equality.

As an example, let's see which number is bigger, $\pi/2$ or $\sqrt{2}$. To do so, create a `test10.m` script for the next code and run it:

```
% evaluate the two numbers
halfpi=pi/2;
rt2=sqrt(2);
% now see which one is 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('Good that that is settled.')
```

```
pi/2 is greater than sqrt(2)!
Good that that is settled.
```

Logical operators

Logical operators allow you to create more complex conditions from simpler ones. The standard logical operators are ~(NOT), && (AND), and || (OR):

```
% Logical NOT: CONDITION must not be true
~ CONDITION
```

```
% Logical AND: Both conditions must be true
CONDITION1 && CONDITION2
```

```
% Logical OR: At least one condition must be true
CONDITION1 || CONDITION2
```

The above operators are in order of precedence. Common sense would further indicate that logical operators should always have lower precedence than numerical comparisons. However, crazy as it may be, Matlab gives ~ precedence

over numerical comparisons. So be sure to use parentheses as needed to be safe and for readability.

Note that a single & or | is equivalent to && or || as far as we are concerned. However, use of a single & or | is not recommended by Matlab as these operators may behave differently in a more general context.

(If not in an `if` or `while`, & and | behave "non-short-circuiting"; Matlab will continue to evaluate `CONDITION2` even if it already knows the final result of the combined condition from `CONDITION1`. For example, if not used in an `if` or `while` statement, a condition like

```
exist('myfun.m','file')&(myfun(1)>1)
```

would create a Matlab error if function file `myfun.m` does not exist; Matlab would still try to evaluate the second condition even though it already knows that function `myfun` does not exist. Using && instead of & prevents that.)

There is also XOR, (exactly one condition must be true), but you rarely want it, and if you do, you can do the same thing using ~, &&, and ||.

The next examples explore whether π is in between 3 and 4 but not 3.2 as the "Indiana pi bill" would have it. Put them in a script `test11.m` and run it:

```
% we *need* the parentheses; ~ takes precedence over ==!
if pi>3 && pi<4 && ~ (pi==3.2)
    disp('pi is between 3 and 4 and not 3.2!')
end

% the next might be more readable?
if (pi>3) && (pi<4) && ~ (pi==3.2)
    disp('pi is between 3 and 4 and not 3.2!')
end

% definitely the below is more readable
if (pi>3) && (pi<4) && (pi~=3.2)
    disp('pi is between 3 and 4 and not 3.2!')
end
```

```
pi is between 3 and 4 and not 3.2!
pi is between 3 and 4 and not 3.2!
pi is between 3 and 4 and not 3.2!
```

Checking condition numbers

Remember solving the linear system of equations in lesson5? The coefficient matrix and right hand side of the system were:

```
% coefficient matrix
A = [1 2 3;
     0 5 6;
     7 8 9];

% right hand side vector
b = [3;
     2;
     9];
```

At that time we had to manually check the condition number of the matrix and draw conclusions. Now, using `if` statements, we can let Matlab check the condition number and automatically take appropriate action based on the value.

One reasonable way to do so is shown below. Put it in a script `test12.m` and run it.

```
% find the condition number of matrix A
condA=cond(A)

% the relative error in the solution due to inaccuracy
xRelerrDueToMatlab=condA*eps

% draw an appropriate conclusion
if xRelerrDueToMatlab >= 1
    % refuse to solve the system
    disp('*** Error: There is no reasonable solution!')
else
    % solve the system
    x = A \ b
    % if the error seems significant, warn
    if xRelerrDueToMatlab > 0.001
        fprintf(...
            '** Warning: estimated error %1E%%!\n',...
            xRelerrDueToMatlab*100)
    end
end
```

```
condA =  37.939
xRelerrDueToMatlab =     8.4241e-15
x =
    1
   -2
    2
```

Next try to set component `A(2,1)` to 4 and then run the script again. The script should refuse to process the matrix. Then set component `A(2,1)` to `4-100*eps` and see what happens if you run the script again. It should warn for excessive Matlab error.

Note that, at least in my opinion, it is ugly to have so much code inside an `else` clause. If you replace the `disp` function by the `error` function, you do not need the `else`. That is shown in this alternative version of `test12.m`:

```
% find the condition number of matrix A
condA=cond(A)

% the relative error in the solution due to inaccuracy
xRelerrDueToMatlab=condA*eps

% draw an appropriate conclusion
if xRelerrDueToMatlab >= 1
    % refuse to solve the system and stop executing
    error('*** Error: There is no reasonable solution!')
end

% solve the system
x = A \ b

% if the error seems significant, warn
if xRelerrDueToMatlab > 0.001
    fprintf(...
        '** Warning: estimated error %1E%%!\n',...
        xRelerrDueToMatlab*100)
end
```

```
condA =  37.939
xRelerrDueToMatlab =    8.4241e-15
x =
   1
  -2
   2
```

If there is no reasonable solution, the `error` function will print the message and terminate execution of the script. So the system will not be solved in that case even without the `else`.
A somewhat different way to do it is to still use the `disp` command, but follow it immediately by a `return` command. The `return` command simply stops execution of the script (but does not inform Matlab behind the scenes that it was due to an error).

Doing infinite sums to a given accuracy

Earlier in this lesson, we did the sum

$$S = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n_{\max}^2} = \sum_{n=1}^{n_{\max}} \frac{1}{n^2}$$

to $n_{\max} = 1000$ 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 n, however, requires that we estimate the error that comes from not summing the remaining terms. Of course, estimating that error can only be an educated guess. (We would know the exact error only 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 $|t_n|$ of the term currently being added to the sum gives the estimated error. This or its equivalent is being done, and published in leading journals, by uncounted numbers of "numerical analysts" all over the place. In the large majority of the cases it does not work. If the "analyst" is lucky, it just means that the true error in the analyst's result is *vastly* larger than the analyst believes. However, there are well known and common cases where using the $|t_n|$ error estimate, "analysts" found and published solutions *that did not exist.* From calculus you should still know that using $|t_n|$ as estimated error only works correctly if the sum is "alternating", i.e. the terms t_n change sign all the time.

2. If the sum is not alternating, the usual case, you should assume that the estimated error is about n times the magnitude $|t_n|$ of the term currently being added to the sum. The factor n makes no big difference if the sum "rapidly converges", i.e. it takes a relatively small number of terms to get an accurate answer. (You should be so lucky.) In the usual case of a slowly converging sum, the factor n makes a big difference. And the factor n works very well for the sums you typically encounter. If you want a common sense derivation of that: "If you have already summed a 1,000 terms and only made slow progress, do you *really* think that another single term, or even 10 of them, are going to make all the difference?"

Using the appropriate termination criterion, 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 to stop the summation. More precisely, 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 $\pi^2/6$.

First we need to set the tolerance we allow. Also, we should set some absolute limit on the number of terms that Matlab may sum. Otherwise we may wait 10 years for a result that never materializes.

```
% the allowed tolerance in value
tol=0.0001

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

```
tol =     1.0000e-04
nMax =   100000
```

Next put the following code in a file `test13.m` and run it. (Save `test5.m` as `test13.m` and then modify that.)

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

% add terms until it seems accurate but no more than nMax
for n=1:nMax
    % compute term t_n
    t_n=1/n^2;
    % add term t_n to the sum
    total=total+t_n;
    % find the current appropriately estimated error
    estError=n*abs(t_n);
    % 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)
totalExact=pi^2/6;
trueError=abs(total-totalExact);
```

```
fprintf('The exact infinite sum is %.4f\n',totalExact)
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',n)
    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',n)
end
```

```
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.
```

Note that while in this sum the estimated error is the same as the true error, this is a coincidence. In general the estimated error is only a rough ballpark of the true error.

You should also try what happens if you use the bad estimated error $|t_n|$. Do you still get the sum to about the requested tolerance?

Next try also summing the sum with terms $1/n$ instead of $1/n^2$. Use tolerance `0.001`. Try both the bad and the good estimated error. What is the exact sum?

Warning!!!

Students who end up with frozen homework programs, or messages that Java or Adobe are misbehaving: Matlab is not to blame, nor are Java or Adobe. The students themselves are to blame.

These students have incorrectly implemented the `break` command, or even omitted it completely.

If they get messages about Java or Adobe, then they have also emitted a semicolon. 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. If you do want to see some numbers that

are produced while doing the sum, you must reduce `nMax` to some reasonable number like 100 instead of 100,000.

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 our previous example of e^x.

First we need to again set a desired value for x and a limit on what number of terms we could possible sum in a reasonable time.

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

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

```
x =  1
nMax =  100000
```

Next put the following code in a script `test14.m`. (Save `test7.m` as `test14.m` and then modify that.)

```
% initialize the sum to term t_0 = 1
n=0;
t_n=1;
total=t_n;

% loop to add up to nMax more terms to the sum
for n=1:nMax
    % compute term t_n from the previous one
    t_n=x/n*t_n;
    % see whether the term can still change the sum
    totalChange=(total+t_n)-total;
    % if it cannot, we can stop
```

```
    if totalChange==0
        break
    end
    % add term t_n to the sum
    total=total+t_n;
end
if totalChange==0
    fprintf('Converged after %i terms.\n',n)
else
    fprintf('*** Not converged after %i terms!\n',n)
end

% print out the obtained value
total

% see how big the error really is
totalError=total-exp(x)
```

```
Converged after 18 terms.
total =  2.7183
totalError =    4.4409e-16
```

Additional remark

In some sums, the terms t_n might contain oscillating factors, like $\sin(nx)$, say. When you compute `totalChange`, leave out these oscillating factors from t_n. Otherwise you might incorrectly conclude that the sum has converged at some value of n, just because at that particular value of n, $\sin(nx)$ happens to be very small or zero.

You might want to remember that such oscillating factors are very common in a special kind of sum called a "Fourier Series". Yes, they are worse than Taylor series.

WHILE LOOPS

The `while` command is similar to the `for` command in that it loops. However, `while` stays looping as long as some condition remains true. The generic form is:

```
while CONDITION
    THINGS_TO_DO
end
```

Note that if there is a mistake in `CONDITION`, a `while` loop can keeping looping forever. In most cases `for` is a better choice than `while` for looping. But a `while`

command can be appropriate in cases where you have no clue when looping will stop, like in user interaction.

Getting input from a user using while

Let's keep looping until the user admits that Matlab is great. Put the next code in a script `test15.m` and run it:

```
% Do NOT delete: preprepared version of test15.m

% ask the user for his/her name
name=input('Please enter your name: ','s');

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

% loop until we get the right answer
choice=0;
while choice ~= 4

    % show the menu and get the user's choice
    choice=menu(header,...
                'Matlab is horrible.',...
                'Matlab is too much work.',...
                'Matlab is OK.',...
                'Matlab is great!');

    % change the header in case that the answer is wrong
    header='Wrong answer.  Try again:';

end

% confirm that the user made the right choice
disp('It is good to see you agree with that!')
```

Unfortunately, an interactive script like the above one cannot be published in Matlab. So below is a fake one that shows roughly what happens:

```
% ask the user for his/her name
fprintf('Please enter your name: ')
% in this fake script get the user's name from the system
name=getenv('USER');      % Unix version
%name=getenv('USERNAME') % Microsoft version
% complete the prompt line with the name
fprintf('%s\n \n',name)

% define the menu header
```

```
header=[name ' admits that:'];

% loop until we get the right answer
choice=0;
while choice ~= 4

    % show a fake menu
    disp(header)
    disp('  [ 1] Matlab is horrible.')
    disp('  [ 2] Matlab is too much work.')
    disp('  [ 3] Matlab is OK.')
    disp('  [ 4] Matlab is great!')
    disp(' ')
    fprintf('Select a number: ')

    % also fake the users choice to be 4
    choice=4;
    fprintf('%i\n \n',choice)

    % change the header in case that the answer is wrong
    header='Wrong answer.  Try again:';

end

% confirm that the user made the right choice
disp('It is good to see you agree with that!')
```

```
Please enter your name: dommelen

dommelen admits that:
  [ 1] Matlab is horrible.
  [ 2] Matlab is too much work.
  [ 3] Matlab is OK.
  [ 4] Matlab is great!

Select a number: 4

It is good to see you agree with that!
```

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.

First we need to again set a desired value for x and a limit on what number of terms we could possible sum in a reasonable time.

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

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

```
x =  1
nMax =  100000
```

Next put the following code in a script `test16.m` and run it. (Save `test14.m` as `test16.m` and then modify that.)

```
% initialize the sum to term t_0 = 1
n=0;
t_n=1;
total=t_n;

% make sure that summing does not stop immediately
totalChange=1;

% in a while loop, add terms as long as the sum changes
while totalChange ~= 0
    % stop if it takes too many terms
    if n >= nMax
        break
    end
    % each time through, increase the n value by one
    n=n+1;
    % compute term t_n from the previous value
    t_n=t_n*x/n;
    % see whether the term can still change the sum
    totalChange=(total+t_n)-total;
    % add term t_n to the sum
    total=total+t_n;
end
if totalChange==0
    fprintf('Converged after %i terms.\n',n)
else
    fprintf('*** Not converged after %i terms!\n',n)
end

% print out the obtained value
total
```

```
% see how big the error really is
totalError=total-exp(x)
```

```
Converged after 18 terms.
total =  2.7183
totalError =    4.4409e-16
```

Clearly, this is uglier than using the `for` loop. For example, the above code is longer and messier than with the `for` loop. And it still adds the final t_n to the sum even though it cannot change the sum.

Unless you have a good reason, avoid `while` loops.