Example: Here is a function that employs the if-else *structure* to determine if a score earned by a student is a passing score (60):

grader.m

```
function grader(score)
% This function determines if score is a passing grade.
% Note that if a function does not return a value, then
% the = sign is dropped.
if score >= 60
    disp('passing grade')
else
    disp('failing grade')
end
```

```
>> grader(58)
failing grade
>> grader(79)
passing grade
```

Example: Here is a function that utilizes the if structure to return a letter grade based on a numerical score:

letter_grade.m

```
function letter_grade(score)
% This function returns a letter grade based on an input score.
if score >= 95, disp('A'), end
if score >= 90 && score < 95, disp('A-'), end
if score >= 85 && score < 90, disp('B+'), end
if score >= 80 && score < 85, disp('B'), end
if score >= 75 && score < 80, disp('B-'), end
if score >= 70 && score < 75, disp('C+'), end
if score >= 65 && score < 70, disp('C'), end
if score >= 60 && score < 65, disp('C-'), end
if score < 60, disp('not passing!'), end
end
```

```
>> letter_grade(65.6)
C
>> letter_grade(59)
not passing!
>> letter_grade(78)
B-
```

The error, return, and nargin commands

Functions can be made more robust by including code that detects and reports error. For example, let us say we have a function that returns the value of the function $y(x) = \frac{1}{x}$,Obviously, we should not divide by zero. How can our function test the value of the input *x* and report an error if $x = 0$?

The error **Command.** The error('message') command: Can be used to trap an error by displaying a message and immediately aborting the function. Here is an example:

error_trap.m

```
function y=error_trap(x)
% This function returns an error upon trying to
% divide by zero.
if x == 0
    error('Attempt to divide by zero')
end
y=1/x;
end
```

```
>> error_trap(0)
Error using error_trap (line 5)
Attempt to divide by zero
>> error_trap(4)
ans =
    0.2500
```

Example: Write Matlab code for the following algebraic expression to be executed *if and only if* the denominator is different from zero and the quantity under the square root is positive or zero.

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Solution:

```
if (a==0) || (b^2-4*a*c < 0)
    error('division by zero or complex quantity')
else
    x=(-b+sqrt(b^2-4*a*c))/(2*a);
end
```

Note the use of parenthesis in the condition part of the `if` statement. As noted earlier, their usage is optional. However, the use of parenthesis is generally recommended.

The return **Command.** Reporting error and aborting a script or a function can also be implemented using the return command. return causes a return to the invoking program or to the keyboard. Normally, functions return when the end of the function is reached. A return command can be used to force an early return. Here is an example:

return_demo.m

```
% return_demo is a script that demonstartes the return command
if (a==0) || (b^2-4*a*c < 0)
    disp('division by zero or complex quantity')
    return;
else
    x=(-b+sqrt(b^2-4*a*c))/(2*a)
end
```

```
>> a=1; b=2; c=3;
>> return_demo
division by zero or complex quantity
>> a=1; b=5; c=1;
>> return_demo
x =
  -0.20871215252208
```

The following is the function version of the above script:

return_demo_f.m

```
function x=return_demo_f(a,b,c)
% return_demo_f is a function that demonstartes the return command
if (a==0) || (b^2-4*a*c < 0)
    disp('division by zero or complex quantity')
    return
else
    x=(-b+sqrt(b^2-4*a*c))/(2*a);
end
```

```
>> return_demo_f(1,2,3)
division by zero or complex quantity
>> return_demo_f(1,5,1)
ans =
  -0.20871215252208
```

The nargin **Command.** What would happen if the user forgets to input one or more function arguments? Matlab provides a nargin variable (number of arguments in the input) which is equal to the number of input arguments provided by the user in a function call.

The following is a script that sets the *default* number of terms in a geometric series sum to $m = 1000$ if the user does not provide a value for m.

series2_nargin.m*

```
function S = series2_nargin(a,m)
% sum of a geometric series of terms a^n from 1 to m
% employing the command "sum(x)"
if nargin==1
    m=1000;
    disp('m was set to 1000')
end
if nargin==0
    disp('missing input argument(s)')
    return
end
x=a*ones(1,m);
n=1:m;
S=sum(x.^n);
end
```

```
   >> series2_nargin(.25)
m was set to 1000
ans =
   0.333333333333333
   >> series2_nargin
missing input argument(s)
```

Example: Write a function that computes the factorial of an integer x. Compare your function to the Matlab built-in function factorial(x).

factorial_mh.m

```
function y = factorial_mh(x)
% factorial_mh(x) computes the factorial
% of integer x (x>0).
y=1;
if x==0, return, end
for n=1:1:x
    y=y*n;
end
```

```
>> factorial_mh(5)
ans =
   120
```

What happens if x is negative? Would including the following code help? If so, where do you place it within the function?

```
if x<0, disp('input must be positive'), return, end
```

What happens if x is not an integer, say, x = 4.1 in the above function? What is y?

Would including the following code help? What does `rem(x,1)` return?

```
if rem(x,1)~= 0
   disp('input must be an integer')
   return
end
```

Consider the function series2_mh that was introduced in the previous lecture:

series2_mh.m

```
function S = series2_mh(a,m)
% sum of a geometric series of terms a^n from 1 to m
% employing the command "sum(x)" which adds the
% elements of vector x.
x=a*ones(1,m);
n=1:m;
S=sum(x.^n);
end
```

We can rewrite the above function using a for loop to compute the series sum, as follows:

series_mh.m

```
function sum = series_mh(a,m)
% sum of a geometric series of terms a^n from 1 to m.
sum=0;     % initialization of variable sum.
for n=1:m
    sum=sum+a^n;
end
```

```
>> series_mh(0.25,20)
ans =
   0.333333333333030
```

(Note, we cannot use i (or j) as index in for loop because i and j are used by Matlab to represent the complex unit ($\sqrt{-1}$))

The tic and toc commands can be used to compute the execution time of a given set of instructions. The tic command saves the current time that toc later employs to display the elapsed time. Here is a speed comparison between two above functions:

series_speed.m

```
function sum = series_speed(a,m)
% sum of a geometric series of terms a^n from 1 to m.
tic
sum=0;     % initialization of variable sum.
for n=1:m
        sum=sum+a^n;
end
toc
end
```

series2_speed.m

```
function S = series2_speed(a,m)
% sum of a geometric series of terms a^n from 1 to m
% employing the command "sum(x)" which adds the
% elements of vector x.
tic
x=a*ones(1,m);
n=1:m;
S=sum(x.^n);
toc
end
```

```
   >> series_speed(.25,1000)
Elapsed time is 0.000261 seconds.
ans =
   0.333333333333333
   >> series2_speed(0.25,1000)
Elapsed time is 0.000327 seconds.
ans =
   0.333333333333333
   >> series_speed(.25,10000)
Elapsed time is 0.002027 seconds.
ans =
   0.333333333333333
   >> series2_speed(0.25,10000)
Elapsed time is 0.001275 seconds.
ans =
   0.333333333333333
```

(**Note:** The elapsed time values are machine dependent)

For the above functions and with m=10,000, first function without for loop is about 37% faster than second function (looping)! **Your turn:** Write the formula that leads to the 37% value.

Here is the series_while function which rewrites the series_mh function employing a while loop:

series_while.m

```
function sum = series_while(a,m)
% sum of a geometric series of terms a^n from 1 to m
sum=0;     % initialization of variable sum
n=1;       % initialization of index n
while n<=m
    sum=sum+a^n;
    n=n+1;
end
```

Notice how the index is now incremented inside the loop.

As an option, the condition part of the while loop (also, of the for loop) can be enclosed inside parentheses:

While (n <= m)

The following is another example of the while loop (a script of four lines entered directly at the Matlab prompt):

```
    >> x=8
while x>0
x=x-3
end
x =

     8

x =

     5

x =

     2

x =

    -1
```

Alternatively, we may type all commands, separated by commas, on a single Matlab line:

```
    >> x=8, while x>0, x=x-3, end
x =

     8

x =

     5

x =

     2

x =

    -1
```

Why is the last x value negative?

Interrupting Loops: The continue and break Commands

A particular iteration of the for and while loops can be skipped if a condition inside the loop is met. This can be implemented using the continue command.

Let us say we want to compute the square root of the positive-valued elements in a vector x, and ignore the negative ones. Let x = [1 4 -4 16 -2 9]. Here are two scripts (one with a for loop and another with a while loop):

A script and its output (with a for loop):

```
>> format short
x=[1 4 -4 16 -2 9];
m=length(x);
disp('Sqrt(x)=')
for n=1:m
    if x(n)<0
        continue
    end
    y=sqrt(x(n));
    disp(y)
end
Sqrt(x)=
     1
     2
     4
     3
```

A script and its output (with a while loop):

```
>> format short
x=[1 4 -4 16 -2 9];
m=length(x);
n=1;
disp('Sqrt(x)=')
while n<=m
    if x(n)<0
        n=n+1;
        continue
    end
    y=sqrt(x(n));
    n=n+1;
    disp(y)
end
Sqrt(x)=
     1
     2
     4
     3
```

Note that the script with the for loop is more concise. Also, note how the index n is incremented in two places inside the while loop. Why?

The break **command.** The break command inside a loop forces the program to abort the loop, and continue with the instruction immediately after the loop's end command.

```
format short
disp('x =')
x=[1   4    -4      16   -2   9];
m=length(x);
for n=1:m
  if x(n)<0
      break
  end
  y=sqrt(x(n));
  disp(y)
end
```

```
x =
     1
     2
```

```
format short
disp('x =')
x=[1   4    -4      16   -2   9];
m=length(x);
n=1;
while n<=m
  if x(n)<0
    n=n+1;
    break
  end
  y=sqrt(x(n));
  disp(y)
  n=n+1;
end
```

```
x =
     1
     2
```

In the above code, the index increment statement n=n+1, just before break, is redundant. Why?

Your turn: Write a function that computes the sum of the *factorial* of the components of a vector x. The function should skip negative and non-integer components of x. Example for input x = [3 -2 5.1 4] the function should return y=3!+4!=30.