# Introduction to MATLAB Programming

Loops

# The for loop

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a script or function, and when it *is* known ahead of time how many times the statements will be repeated. The statements that are repeated are called the action of the loop. For example, it may be known that the action of the loop will be repeated five times. The terminology used is that we ***iterate*** through the action of the loop five times.

The variable that is used to iterate through values is called a ***loop variable***, or an ***iterator variable***. For example, the variable might iterate through the integers 1 through 5 (e.g., 1, 2, 3, 4, and then 5). Although variable names in general should be mnemonic, it is common for an iterator variable to be given the name *i* (and if more than one iterator variable is needed, *i*, *j*, *k*, *l*, etc.) This is historical, and is because of the way integer variables were named in Fortran. However, in MATLAB both **i** and **j** are built-in values for $\sqrt{-1}$ , so using either as a loop variable will override that value. If that is not an issue, then it is acceptable to use *i* as a loop variable.

# Loops

There are two basic kinds of loops in programming: ***counted loops***, and ***conditional loops***.

A counted loop is one that repeats statements a specified number of times (e.g., ahead of time it is known how many times the statements are to be repeated).

A conditional loop also repeats statements, but ahead of time it is not known *how many* times the statements will need to be repeated.

With a conditional loop, for example, you might say "repeat these statements until this condition becomes false." The statement(s) that are repeated in any loop are called the action of the loop.

# The for loop

The general form of the **for** loop is:

```
for loopvar = range
    action
end
```

where *loopvar* is the loop variable, *range* is the range of values through which the loop variable is to iterate, and the action of the loop consists of all statements up to the **end**.

```
for i = 1:5
   fprintf('%d\n',i)
end
```

# Quick Question

How could you print this column of integers?

```
  0
 50
100
150
200
```

# Finding Sums and Products (1)

$$\sum_{i=1}^{n} i$$

sum_1_to_n.m

```
function runsum = sum_1_to_n(n)
% This function returns the sum of
% integers from 1 to n
runsum = 0;
for i = 1:n
  runsum = runsum + i;
end
```

# Finding Sums and Products (2)

myfact.m

```
function runprod = myfact (n)
% This function returns the product of
% integers from 1 to n, or n!
runprod = 1;
for i = 1:n
  runprod = runprod * i;
end
```

### The Efficient Method

MATLAB has a built-in function, **factorial**, that will find the factorial of an integer n:

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

# Sums and Products with Vectors (1)

The vector is passed as an argument to the function. The function loops through all the elements of the vector, from 1 to the length of the vector, to ad them all to the running sum.

myvecsum.m

```
function outarg = myvecsum(vec)
% This function sums the elements in a vector
outarg = 0;
for i = 1:length(vec)
  outarg = outarg + vec(i);
end
```

Any vector could be passed to this function; for example, just specifying values for the elements in square brackets:

```
>> myvecsum([5 9 4])
ans =
    18
```

## The Efficient Method

MATLAB has a built-in function, **sum**, that will sum all values in a vector. Again, any vector can be passed to the **sum** function:

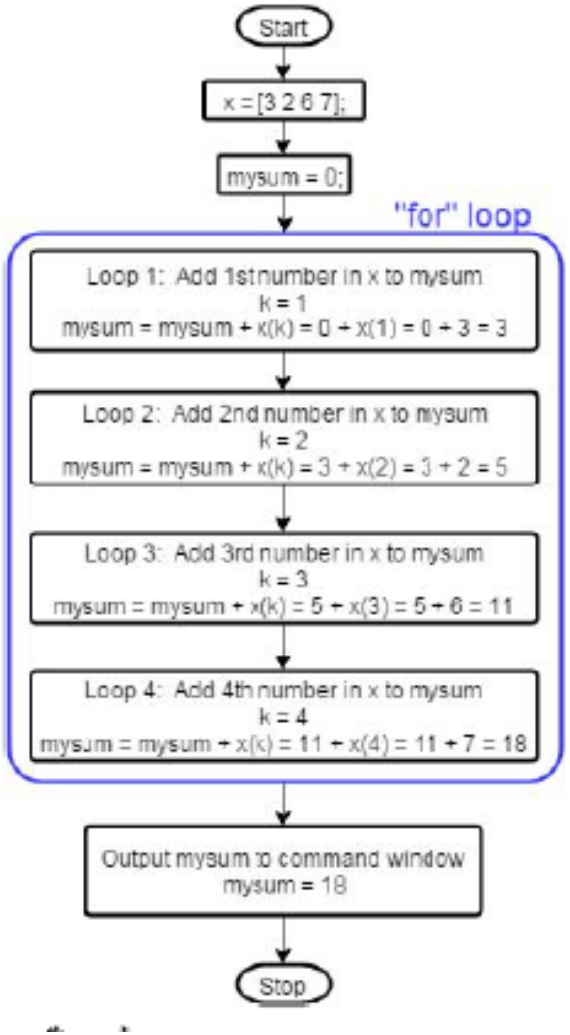
```
>> sum([5 9 4])
ans =
    18
```

# Another for Example

```
clear all;clc;
x = 0;  % initial x before loop starts
loopNum = 1;  % used to keep track of # of loops
for counter = 9:-1:7 % counts down this time
    disp(['During loop number ' num2str(loopNum)]);
    disp(['  The for-loop cntr variable is ' num2str(counter)]);
    x = x + counter; % add counter to x
    disp(['  "x = x + counter" changes x to ' num2str(x)]);
    loopNum = loopNum+1; % increment loop number
end
```

Running the code generates the following:

```
During loop number 1
  The for-loop cntr variable is 9
  "x = x + counter" changes x to 9
During loop number 2
  The for-loop cntr variable is 8
  "x = x + counter" changes x to 17
During loop number 3
  The for-loop cntr variable is 7
  "x = x + counter" changes x to 24
```



# Sums and Products with Vectors (2)

The vector is passed as an argument to the function. The function loops through all the elements of the vector, from 1 to the length of the vector, to multiply them all by the running product.

myvecprod.m

```
function outarg = myvecprod(vec)
% This function finds the product of
% the elements in a vector
outarg = 1;
for i = 1:length(vec)
  outarg = outarg * vec(i);
end
```

```
>> myvecprod(1:5)
ans =
  120
>> myvecprod([5 9 4])
ans =
  180
```

## The Efficient Method

MATLAB has a built-in function, **prod**, that will return the product of all values in a vector.

```
>> prod([5 9 4])
 ans =
     180
```

# Quick Question

How could we write a function *prod_m_to_n* to calculate and return the product of the integers m to n without assuming a specific order of the arguments? In other words, both the function calls *prod_m_to_n(3,6)* and *prod_m_to_n(6,3)* would return the result of 3 * 4 * 5 * 6 or 360.

prod_m_to_n.m

```
function runprod = prod_m_to_n(m,n)
% Product of m to n using a for loop

% Make sure m is less than n
if m > n
  temp = m;
  m = n;
  n = temp;
end
% Loop to calculate the running product
runprod = 1;
for i = m:n
  runprod = runprod * i;
end
```

## The Efficient Method

prod_m_to_nii.m

```
function outprod = prod_m_to_nii(m,n)
% Product of m to n using : and prod
if m < n
  outprod = prod(m:n);
else
  outprod = prod(m:-1:n);
end
```

# Preallocating a Vector

There are essentially two programming methods that could be used to simulate the **cumsum** function. One method is to start with an empty vector and concatenate each running sum value to the vector. Extending a vector, however, is very inefficient. A better method is to ***preallocate*** the vector to the correct size and then change the value of each element to be successive running sums. Both methods will be shown here.

myveccumsum.m

```
function outvec = myveccumsum(vec)
% This function imitates cumsum for a vector
outvec = [];
runsum = 0;
for i = 1:length(vec)
  runsum = runsum + vec(i);
  outvec = [outvec runsum];
end
```

Here is an example of calling the function:

```
>> myveccumsum([5 9 4])
ans =
     5  14  18
```

myveccumsumii.m

```
function outvec = myveccumsumii(vec)
% This function imitates cumsum for a vector
% It preallocates the output vector
outvec = zeros(size(vec));
runsum = 0;
for i = 1:length(vec)
  runsum = runsum + vec(i);
  outvec(i) = runsum;
end
```

# Combining for Loops with if Statements

Find the minimum value in a vector, the algorithm is

- The minimum so far is the first element in the vector
- Loop through the rest of the vector
  - If any element is less than the minimum found so far, then that element is the new minimum so far

myminvec.m

```
function outmin = myminvec(vec)
% Finds the minimum value in a vector
outmin = vec(1);
for i = 2:length(vec)
  if vec(i) < outmin
    outmin = vec(i);
  end
end
```

```
>> vec = [3 8 99 -1];
>> myminvec(vec)
ans =
     -1

>> vec = [3 8 99 11];
>> myminvec(vec)
ans =
      3
```

### The Efficient Method

MATLAB has functions **min** and **max**, which find the minimum and maximum values in a vector.

```
>> vec = [5 9 4];
>> min(vec)
ans =
       4
```

# Quick Question

Write a function to find and return the maximum value in a vector.

# For Loops that do not use the iterator variable in the action

```
for i = 1:3
    fprintf('I will not chew gum\n')
end
```

produces the output:

```
I will not chew gum
I will not chew gum
I will not chew gum
```

# Quick Question

What would be the result of the following **for** loop?

```
for i = 4:2:8
  fprintf('I will not chew gum\n')
end
```

# Input in a for loop (1)

The following script repeats the process of prompting the user for a number, and ***echo printing*** the number (which means simply printing it back out). A **for** loop specifies how many times this is to occur. This is another example in which the loop variable is not used in the action, but instead just specifies how many times to repeat the action.

forecho.m

```
% This script loops to repeat the action of
% prompting the user for a number and echo-printing it
for iv = 1:3
  inputnum = input('Enter a number: ');
  fprintf('You entered %.1f\n',inputnum)
end
```

```
>> forecho
Enter a number: 33
You entered 33.0
Enter a number: 1.1
You entered 1.1
Enter a number: 55
You entered 55.0
```

Modify the *forecho* script to sum the numbers that the user enters and print the result.

# Nested for loops (1)

The action of a loop can be any valid statement(s). When the action of a loop is another loop, this is called a ***nested loop***.

As an example, a nested **for** loop will be demonstrated in a script that will print a box of *'s. Variables in the script will specify how many rows and columns to print. For example, if *rows* has the value 3, and *columns* has the value 5, the output would be:

```
*****
*****
*****
```

- For every row of output,
  - Print the required number of *'s
  - Move the cursor down to the next line (print the '\n')

printstars.m

```
% Prints a box of stars
% How many will be specified by 2 variables
% for the number of rows and columns
rows = 3;
columns = 5;
% loop over the rows
for i=1:rows
  % for every row loop to print *'s and then one \n
  for j=1:columns
    fprintf('*')
  end
  fprintf('\n')
end
```

# Nested for loops (2)

The following function *multtable* calculates and returns a matrix that is a multiplication table. Two arguments are passed to the function, which are the number of rows and columns for this matrix.

multtable.m

```
function outmat = multtable (rows, columns)
% Creates a matrix which is a multiplication table

% Preallocate the matrix
outmat = zeros(rows,columns);
for i = 1:rows
    for j = 1:columns
        outmat(i,j) = i * j;
    end
end
```

In the following example, the matrix has three rows and five columns:

```
>> multtable(3,5)
ans =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
```

# Quick Question

How could this program be modified to print a triangle of *'s instead of a box? For example,

```
*
**
***
```

printtristars.m

```
% Prints a triangle of stars
% How many will be specified by a variable
% for the number of rows
rows = 3;
for i=1:rows
    % inner loop just iterates to the value of i
    for j=1:i
        fprintf('*')
    end
    fprintf('\n')
end
```

```
>> printtristars
*
**
***
```

# Nested loops and Matrices

Nested loops often are used when it is necessary to loop through all the elements of a matrix.

As an example, we will calculate the overall sum of the elements in a matrix.

mymatsum.m

```
function outsum = mymatsum(mat)
% Calculates the overall sum of the elements
% in a matrix
[row col] = size(mat);
outsum = 0;
```

```
% The outer loop is over the rows
for i = 1:row
    for j = 1:col
        outsum = outsum + mat(i,j);
    end
end
```

```
>> mat = [3:5; 2 5 7]
mat =
     3     4     5
     2     5     7
>> mymatsum(mat)
ans =
    26
```

### The Efficient Method

MATLAB has a built-in function **sum**, as we have seen. For matrices, like many built-in functions, the **sum** function operates columnwise, meaning that it will return the sum of each column.

```
>> mat
mat =
     3     4     5
     2     5     7
>> sum(mat)
ans =
     5     9    12
```

So, to get the overall sum, it is necessary to sum the column sums!

```
>> sum(sum(mat))
ans =
     26
```

# Quick Question

How would we sum each individual column, rather than just getting an overall sum?

matcolsum.m

```
function outsum = matcolsum(mat)
% Calculates the sum of every column in a matrix
% Returns a vector of the column sums
[row col] = size(mat);
% Preallocate the vector to the number of columns
outsum = zeros(1,col);
% Every column is being summed so the outer loop
% has to be over the columns
for i = 1:col
    % Initialize the running sum to 0 for every column
    runsum = 0;
    for j = 1:row
        runsum = runsum + mat(j,i);
    end
    outsum(i) = runsum;
end
```

Modify this function; create a function *matrowsum* to calculate and return a vector of all the row sums, instead of column. For example, calling it and passing the previous *mat* variable would result in the following:

# Combining Nested for loops and if statements

The statements inside of a nested loop can be any valid statement, including any selection statement. For example, there could be an **if** or **if-else** statement as the action, or part of the action, in a loop.

Write a function *mymatmin* that finds the minimum value in each column of a matrix argument and returns a vector of the column minimums. Here is an example of calling the function.

```
>> mat = randint(3,4,[1 20])
mat =
     15     19     17      5
      6     14     13     13
      9      5      3     13

>> mymatmin(mat)
ans =
     6  5  3  5
```

Would it matter if the order of the loops was reversed in this example, so the outer loop iterates over the columns and the inner loop over the rows?

# Vectorizing (1)

In most programming languages, when performing an operation on a vector, a **for** loop is used to loop through the entire vector. For example, in MATLAB assuming there is a vector variable *vec*:

```
for i = 1:length(vec)
    % do something with vec(i)
end
```

Similarly, for an operation on a matrix, a nested loop would be required; for example, assuming a matrix variable *mat*:

```
[r c] = size(mat);
for row = 1:r
    for col = 1:c
        % do something with mat(row,col)
    end
end
```

Usually in MATLAB, this is not necessary.

# Vectorizing (2)

Numerical operations can be done on entire vectors or matrices. For example, let's say that we want to multiply every element of a vector $v$ by 3, and store the result back in $v$, where $v$ is initialized as follows:

```
>> v = [3 7 2 1];
```

## The Programming Concept

To accomplish this, we can loop through all the elements in the vector and multiply each element by 3. In the following, the output is suppressed in the loop, and then the resulting vector is shown:

```
>> for i = 1:length(v)
    v(i) = v(i) * 3;
end
>> v
v =
    9   21   6   3
```

## The Efficient Method

In MATLAB, we can simply multiply $v$ by 3 and store the result back in $v$ in an assignment statement:

```
>> v = v*3
v =
    9   21   6   3
```

# Vectorizing (3)

As another example, we can divide every element by 2:

```
>> v= [3 7 2 1];
>> v/2
ans =
    1.5000  3.5000  1.0000  0.5000
```

For a matrix, numerical operations can also be performed on every element. For example, to multiply every element in a matrix by 2 with most languages would involve a nested loop, but in MATLAB it is automatic.

```
>> mat = [4:6; 3: -1:1]
mat =
     4     5     6
     3     2     1
>> mat * 2
ans =
     8    10    12
     6     4     2
```

# Logical Vectors

- Create a vector variable and add 2 to every element in it.
- Create a matrix variable and divide every element by 3.
- Create a matrix variable and square every element.

## The Programming Concept

To accomplish this using the programming method, we would have to loop through all the elements of the vector and compare each element with 5 to determine whether the corresponding value in the result would be logical true or false.

## The Efficient Method

In MATLAB, this can be accomplished automatically by simply using the relational operator >.

```
>> isg = vec > 5
isg =
    0  1  0  0  1  1
```

Notice that this creates a vector consisting of all logical true or false values. Although this is a vector of ones and zeros, and numerical operations can be done on the vector *isg*, its type is **logical** rather than **double**.

```
>> doubres = isg + 5
ans =
    5  6  5  5  6  6

>> whos
```

| Name | Size | Bytes | Class |
|---|---|---|---|
| doubres | 1x6 | 48 | double array |
| isg | 1x6 | 6 | logical array |
| vec | 1x6 | 48 | double array |

To determine how many of the elements in the vector *vec* were greater than 5, the **sum** function could be used on the resulting vector *isg*:

```
>> sum(isg)
ans =
    3
```

# Quick Question

```
>> vec([0 1 0 0 1 1])
??? Subscript indices must either be
real positive integers or logicals.
```

testvecgtn.m

```
function outvec = testvecgtn(vec,n)
% Compare each element in vec to see whether it
% is greater than n or not

% Preallocate the vector
outvec = zeros(size(vec));
for i = 1:length(vec)
  % Each element in the output vector stores 1 or 0
  if vec(i) > n
    outvec(i) = 1;
  else
    outvec(i) = 0;
  end
end
```

Calling the function appears to return the same vector as simply vec > 5, and summing the result still works to determine how many elements were greater than 5.

```
>> notlog = testvecgtn(vec,5)
notlog =
        0  1  0  0  1  1
```

# Quick Question

How could we remedy this? How can we assign logical values rather than doubles?

**Answer:** We can use the **logical** function to make the output argument vector the type **logical** rather than **double**. In the function, the only statement that needs to be modified is the statement that preallocates the vector; in this case, it also changes the type of the vector to **logical**.

```
outvec = logical(zeros(size(vec)));
```

# Logical Built-In Functions

There are built-in functions in MATLAB that are useful in conjunction with vectors or matrices of all logical true or false values; two of these are the functions **any** and **all**. The function **any** returns logical true if any element in a vector is logically true, and false if not. The function **all** returns logical true only if all elements are logically true. Here are some examples. For the variable *vec1*, all elements are logical true so both **any** and **all** return true.

```
>> vec1 = [1 3 1 1 2];
>> any(vec1)
ans =
     1
>> all(vec1)
ans =
     1
```

For *vec2*, some elements are logical true so **any** returns true but **all** returns false.

```
>> vec2 = [1 1 0 1]
vec2 =
     1     1     0     1
>> any(vec2)
ans =
     1
```

# Quick Question

How could we accomplish the same thing by using loops and **if** statements?

myany.m

```
function logresult = myany(vec)
% Simulates the built-in function any

% Assume 0 for the result
logresult = logical(0);
for i = 1:length(vec)
  % if any value is not false, the result will be 1
  if vec(i) ~= 0
    logresult = logical(1);
  end
end
```

```
>> vec2 = [1 1 0 1];
>> myany(vec2)
ans =
  1
>> vec3 = [0 0 0];
>> myany(vec3)
ans =
  0
```

# Quick Question

myall.m

```
function logresult = myall(vec)
% Simulates the built-in function all

% count how many values are true
count = 0;
for i = 1:length(vec)
  if vec(i) ~= 0
    count = count + 1;
  end
end
% if all were true, return 1 else return 0
if count == length(vec)
  logresult = logical(1);
else
  logresult = logical(0);
end
```

```
>> myall(vec1)
ans =
    1

>> myall(vec2)
ans =
    0

>> myall(vec3)
ans =
    0
```

# Vectors and Matrices as Function Arguments (1)

Using most programming languages, if it is desired to evaluate a function on every element in a vector or a matrix, a loop would be necessary to accomplish this. However, as we have already seen, in MATLAB an entire vector or matrix can be passed as an argument to a function; the function will be evaluated on every element. This means that the result will be the same size as the argument.

## The Programming Method

For example, let us find the sine in radians of every element of a vector *vec*. The algorithm would be to loop through the elements of the vector and get the sine of each one. To store the results in a new vector, the most efficient way would be to preallocate it.

```
>> vec = -2:1
vec =
      -2   -1    0    1
>> sinvec = zeros(size(vec));
```

```
>> for i = 1:length(vec)
     sinvec(i) = sin(vec(i));
   end
>> sinvec
sinvec =
   -0.9093  -0.8415   0   0.8415
```

# Vectors and Matrices as Function Arguments (2)

## The Efficient Method

The **sin** function will automatically return the sine of each individual element and the result will also be a vector with a length of four (in this case, in *ans*).

```
>> sin(vec)
ans =
   -0.9093 -0.8415 0 0.8415
```

For a matrix, the resulting matrix will have the same size as the input argument matrix. For example, the **sign** function will find the sign of each element in a matrix:

```
>> mat = [0 4 -3; -1 0 2]
mat =
   0  4  -3
  -1  0   2
>> sign(mat)
ans =
   0  1  -1
  -1  0   1
```

# The Programming Method

## The Programming Method

To write our own *signum* function that accomplishes exactly the same thing as the built-in **sign** function, nested loops would be required.

signum.m

```
function outmat = signum(mat)
% This function imitates the sign function
[r c] = size(mat);
for i = 1:r
  for j = 1:c
    if mat(i,j) > 0
      outmat(i,j) = 1;
    elseif mat(i,j) == 0
```

*(Continued)*

```
        outmat(i,j) = 0;
    else
        outmat(i,j) = -1;
    end
  end
end
```

To test this function, we will create a matrix of random integers in a range from –8 to 8, and pass this to the function. Here are some examples of using this function:

```
>> mat = randint(2,4,[-8 8])
mat =
    -2     7    -3     2
     0     2     0    -6

>> signum(mat)
ans =
    -1     1    -1     1
     0     1     0    -1

>> signmat = signum(mat)
signmat =
    -1     1    -1     1
     0     1     0    -1

>> help signum
  This function imitates the sign function
```

# The Programming Method

Vectors or matrices can be passed to user-defined functions as well, as long as the operators used in the function are correct. For example, we previously defined a function that calculates the area of a circle:

```
>> type calcarea
   function area = calcarea(rad)
   % This function calculates the area of a circle
   area = pi * rad * rad;
```

This function was written assuming that the argument was a scalar, so calling it with a vector instead would produce an error message:

```
>> calcarea(1:3)
??? Error using ==> mtimes
Inner matrix dimensions must agree.
Error in ==> calcarea at 3
   area = pi * rad * rad;
```

This is because the * was used for multiplication in the function, but .* must be used when multiplying vectors term-by-term. Changing this in the function would allow either scalars or vectors to be passed to this function:

calcareaii.m

```
function area = calcareaii(rad)
% This function calculates the area of a circle
% The input argument can be a vector of radii
area = pi * rad .* rad;
```

```
>> calcareaii(1:3)
ans =
  3.1416  12.5664  28.2743
>> calcareaii(4)
ans =
  50.2655
```

# While loops

The **while** statement is used as the conditional loop in MATLAB; it is used to repeat an action when ahead of time it is *not* known *how many* times the action will be repeated. The general form of the **while** statement is:

```
while condition
    action
end
```

The action, which consists of any number of statement(s), is executed as long as the condition is true. The condition must eventually become false to avoid an ***infinite loop***. (If this happens, Ctrl-C will exit the loop.)

factgthigh.m

```
function facgt = factgthigh(high)
% Finds the first factorial > high
i=0;
fac=1;
while fac <= high
  i=i+1;
  fac = fac * i;
end
facgt = fac;
```

Here is an example of calling the function, passing 5000 for the value of the input argument *high*.

```
>> factgthigh(5000)
ans =
    5040
```

# Multiple Conditions in a While loop (1)

In the previous section, we wrote a function *myany* that imitated the built-in **any** function by returning logical true if any value in the input vector was logical true, and logical false otherwise. The function was inefficient because it looped through all the elements in the input vector, even though once one logical true value is found it is no longer necessary to examine any other elements. A **while** loop will improve on this. Instead of looping through all the elements, what we really want to do is to loop until either a logical true value is found, or until we've gone through the entire vector. Thus, we have two parts to the condition in the **while** loop. In the following function, we initialize the output argument to logical false, and an iterator variable *i* to 1. The action of the loop is to examine an element from the input vector: if it is logical true, we change the output argument to be logical true. Also in the action the iterator variable is incremented. The action of the loop is continued as long as the index has not yet reached the end of the vector, and as long as the output argument is still logical false.

myanywhile.m

```
function logresult = myanywhile(vec)
% Simulates the built-in function any
% Uses a while loop so that the action halts
% as soon as any true value is found
logresult = logical(0);
i = 1;
while i <= length(vec) && logresult == 0
  if vec(i) ~= 0
    logresult = logical(1);
  end
  i = i + 1;
end
```

The output produced by this function is the same as the *myany* function, but it is more efficient because now as soon as the output argument is set to logical true, the loop ends.

# Multiple Conditions in a While loop (2)

`while` loops are best if you aren't sure how many loops will be needed.

```
clear all;clc;
k = 0.5; %MUST initialize k to avoid error
while k <= 2 % MUST use k in while loop condition
    fprintf('k is now %.1f\n',k)
    k = k + 0.5; % MUST change k inside loop
                 % to avoid infinite loop
end
```

This code outputs:

```
k is now 0.0
k is now 0.5
k is now 1.0
k is now 1.5
```

Notes:

- Use ctrl-c to break out of infinite loops.
- Deleting the `k=k+0.5;` line (or changing it to `k=k-0.5;`) will result in an infinite loop

# Input in a While loop

The following script repeats the process of prompting the user, reading in a positive number, and echo-printing it, as long as the user correctly enters positive numbers when prompted. As soon as the user types in a negative number, the program will print OK and end.

whileposnum.m

```
% Prompts the user and echo prints the numbers entered
% until the user enters a negative number
inputnum=input('Enter a positive number: ');
while inputnum >= 0
  fprintf('You entered a %d.\n\n',inputnum)
  inputnum = input('Enter a positive number: ');
end
fprintf('OK!\n')
```

When the program is executed, the input/output might look like this:

```
>> whileposnum
Enter a positive number: 6
You entered a 6.

Enter a positive number: -2
OK!
```

If the user enters a negative number the first time, no values would be echo-printed:

```
>> whileposnum
Enter a positive number: -33
OK!
```

# Counting in a While loop (1)

When it is not known ahead of time how many values will be entered into a script, it is frequently necessary to ***count*** the number of values that are entered. For example, if numbers are read into a script, and then the average of the numbers is desired, the script must add them together, and keep track of how many there are, in order to calculate the average. The following variation on the previous script counts the number of numbers that the user successfully enters:

# Counting in a While loop

Write a script *aveposnum* that will repeat the process of prompting the user for positive numbers, until the user enters a negative number, as earlier. Instead of echo-printing them, however, the script will print the average (of just the positive numbers). If no positive numbers are entered, the script will print an error message instead of the average. Here are examples of executing this script:

```
>> aveposnum
Enter a positive number: -5
No positive numbers to
average.

>> aveposnum
Enter a positive number: 8
Enter a positive number: 3
Enter a positive number: 4
Enter a positive number: -6
The average was 5.00
```

countposnum.m

```
% Prompts the user for positive numbers and echo prints as
% long as the user enters positive numbers

% Counts the positive numbers entered by the user
counter=0;
inputnum=input('Enter a positive number: ');
while inputnum >= 0
  fprintf('You entered a %d.\n\n',inputnum)
  counter = counter + 1;
  inputnum = input('Enter a positive number: ');
end
fprintf('Thanks, you entered %d positive numbers\n',counter)
```

# A guessing game

The problem is easy to state. MATLAB "thinks" of an integer between 1 and 10 (i.e., generates one at random). You have to guess it. If your guess is too high or too low, the script must say so. If your guess is correct, a message of congratulations must be displayed.

A little more thought is required here, so a structure plan might be helpful:

1. Generate random integer
2. Ask user for guess
3. While guess is wrong:

   If guess is too low
      Tell her it is too low

   Otherwise
      Tell her it is too high
   Ask user for new guess

4. Polite congratulations
5. Stop.

```
matnum = floor(10 * rand + 1);
guess = input( 'Your guess please: ' );
load splat

while guess ~= matnum
sound(y, Fs)

  if guess > matnum
    disp( 'Too high' )
  else
    disp( 'Too low' )
  end;

  guess = input( 'Your next guess please: ' );
end

disp( 'At last!' )
load handel
sound(y, Fs)              % hallelujah!
```

Try it out a few times. Note that the `while` loop repeats as long as `matnum` is not equal to `guess`. There is no way in principle of knowing how many loops will be needed before the user guesses correctly. The problem is truly indeterminate.

# Error-Checking User Input in a While loop

In most applications, when the user is prompted to enter something, there is a valid range of values. If the user enters an incorrect value, rather than having the program carry on with an incorrect value, or just printing an error message, the program should repeat the prompt. The program should keep prompting the user, reading the value, and checking it, until the user enters a value that is in the correct range. This is a very common application of a conditional loop, looping until the user correctly enters a value in a program. This is called ***error-checking***.

For example, the following script prompts the user to enter a positive number, and loops to print an error message and repeat the prompt until the user finally enters a positive number.

readonenum.m

```
% Loop until the user enters a positive number
inputnum=input('Enter a positive number: ');
while inputnum < 0
  inputnum = input('Invalid! Enter a positive number: ');
end
fprintf('Thanks, you entered a %.1f \n',inputnum)
```

Here is an example of running this script:

```
>> readonenum
Enter a positive number: -5
Invalid! Enter a positive number: -2.2
Invalid! Enter a positive number: c
??? Error using ==> input
Undefined function or variable 'c'.

Invalid! Enter a positive number: 44
Thanks, you entered a 44.0
```

Note that MATLAB itself catches the character input and prints an error message and repeats the prompt when the c was entered.

# Exiting Loops Early

Any loop (e.g. **for** or **while**) can be exited at any point by using the **break** command. For example,

```
clear all;clc;
for k = 2:2:10
    disp(['k = ' num2str(k)])
    if k == 6
        break; %exit loop
    end
end
```

only loops 3 times instead of 5 and outputs:

```
k = 2
k = 4
k = 6
```

# Quick Question

How could we vary this example, so that the script asks the user to enter positive numbers *n* times, where *n* is an integer defined to be 3?

**Answer:** Every time the user enters a value, the script checks and in a **while** loop keeps telling the user that it's invalid until a valid positive number is read. By putting the error-check in a **for** loop that repeats n times, the user is forced eventually to enter three positive numbers.

readnnums.m

```
% Loop until the user enters n positive numbers
n=3;
fprintf('Please enter %d positive numbers\n\n',n)
for i=1:n
  inputnum=input('Enter a positive number: ');
  while inputnum < 0
    inputnum = input('Invalid! Enter a positive number: ');
  end
  fprintf('Thanks, you entered a %.1f \n',inputnum)
end
```

```
>> readnnums
Please enter 3 positive numbers

Enter a positive number: 5.2
Thanks, you entered a 5.2
Enter a positive number: 6
Thanks, you entered a 6.0
Enter a positive number: -7.7
Invalid! Enter a positive number: 5
Thanks, you entered a 5.0
```

# Error-Checking for Integers

Since MATLAB uses the type **double** by default for all values, to check to make sure that the user has entered an integer, the program has to convert the input value to an integer type (e.g., **int32**) and then check to see whether that is equal to the original input. The following examples illustrate the concept.

If the value of the variable *num* is a real number, converting it to the type **int32** will round it, so the result is not the same as the original value.

```
>> num = 3.3;
>> inum = int32(num)
inum =
         3

>> num == inum
ans =
     0
```

If, on the other hand, the value of the variable *num* is an integer, converting it to an integer type will not change the value.

```
>> num = 4;
>> inum = int32(num)
inum =
         4
>> num == inum
ans =
     1
```

The following script uses this idea to error-check for integer data; it loops until the user correctly enters an integer.

readoneint.m

```
% Error-check until the user enters an integer
inputnum = input('Enter an integer: ');
num2 = int32(inputnum);
while num2 ~= inputnum
  inputnum = input('Invalid! Enter an integer: ');
  num2 = int32(inputnum);
end
fprintf('Thanks, you entered a %d \n',inputnum)
```

Here are examples of running this script:

```
>> readoneint
Enter an integer: 9.5
```

# Error-Checking for Integers

```
Invalid! Enter an integer: 3.6
Invalid! Enter an integer: -11
Thanks, you entered a -11
```

```
>> readoneint
Enter an integer: 5
Thanks, you entered a 5
```

Putting these ideas together, the following script loops until the user correctly enters a positive integer. There are two parts to the condition, since the value must be positive and must be an integer.

readoneposint.m

```
% Error checks until the user enters a positive integer
inputnum = input('Enter a positive integer: ');
num2 = int32(inputnum);
while num2 ~= inputnum || num2 < 0
  inputnum = input('Invalid! Enter a positive integer: ');
  num2 = int32(inputnum);
end
fprintf('Thanks, you entered a %d \n',inputnum)
```

# Quick Question

Modify this script to read $n$ positive integers, instead of just one.

```
>> readoneposint
Enter a positive integer: 5.5
Invalid! Enter a positive integer: -4
Invalid! Enter a positive integer: 11
Thanks, you entered a 11
```

# Summary

- A `for` statement should be used to program a determinate loop, where the number of repeats is known (in principle) *before* the loop is encountered. This situation is characterized by the general structure plan:

  Repeat *N* times:
  Block of statements to be repeated,

  where *N* is known or computed *before* the loop is encountered for the first time, and is not changed by the block.

- A `while` statement should be used to program an indeterminate repeat structure, where the exact number of repeats is *not* known in advance. Another way of saying this is that these statements should be used whenever the truth value of the condition for repeating is changed in the body of the loop. This situation is characterized by the following structure plan:

  While *condition* is true repeat:

  statements to be repeated (reset truth value of *condition*).

  Note that *condition* is the condition to repeat.

- The statements in a `while` construct may sometimes never be executed.
- Loops may be nested to any depth.

# Common Pitfalls

- Forgetting to initialize a running sum or count variable to 0.
- Forgetting to initialize a running product variable to 1.
- In cases where loops are necessary, not realizing that if an action is required for every row in a matrix, the outer loop must be over the rows (and if an action is required for every column, the outer loop must be over the columns).
- Forgetting that for array operations based on multiplication, the dot must be used in the operator. In other words, for multiplying, dividing, or raising to an exponent term-by-term, the operators are .*, ./, and .^.
- Attempting to use || or && with arrays; always use | and & when working with arrays; || and && are used only with scalars.
- Not realizing that it is possible that the action of a **while** loop will never be executed.
- Not error-checking input into a program.

# Programming Style Guidelines

- Use loops for repetition only when necessary:
  - **for** statements as counted loops
  - **while** statements as conditional loops
- Do not use *i* or *j* for iterator variable names if the use of the built-in constants **i** and **j** is desired.
- Indent the action of a loop.
- If the loop variable is being used just to specify how many times the action of the loop is to be executed, use the colon operator 1:*n* where *n* is the number of times the action is to be executed.
- Preallocate vectors and matrices whenever possible (when the size is known ahead of time).
- Vectorize code whenever possible. If it is not necessary to use loops in MATLAB, don't!
- Use the array operators .*, ./, .\, and .^ in functions so that the input arguments can be arrays and not just scalars.

# Practice Problem

Write a function ***sumsteps2*** that calculates and returns the sum of 1 to n in steps of 2, where n is an argument passed to the function. For example, if 11 is passed, it will return 1 + 3 + 5 + 7 + 9 + 11. Do this using a **for** loop. Calling the function will look like this:

```
>> sumsteps2(11)
ans =
  36
```

# Practice Problem

Write a function called ***geomser*** that will receive values of r and n, and will calculate and return the sum of the geometric series:

$$1 + r + r^2 + r^3 + r^4 + \ldots + r^n$$

The following examples of calls to this function illustrate what the result should be:

```
>> geomser(1,5)
ans =
     6
>> disp(geomser(2,4))
     31
```

# Practice Problem

Create a $1 \times 6$ vector of random integers, each in the range from 1 to 20. Use built-in functions to find the minimum and maximum values in the vector. Also create a vector of cumulative sums using **cumsum**.

# Practice Problem

Write a program to compute the sum of the series $1^2 + 2^2 + 3^2 \ldots$ such that the sum is as large as possible without exceeding 1000. The program should display how many terms are used in the sum.

# Practice Problem

Use the Taylor series:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

to write a program to compute $\cos x$ correct to four decimal places (x is in radians). See how many terms are needed to get four-figure agreement with the MATLAB function cos. Don't make x too large; that could cause rounding error.

# Practice Problem

A person deposits $1000 in a bank. Interest is compounded monthly at the rate of 1% per month. Write a program which will compute the monthly balance, but write it only *annually* for 10 years (use nested `for` loops, with the outer loop for 10 years, and the inner loop for 12 months). Note that after 10 years, the balance is $3300.39, whereas if interest had been compounded annually at the rate of 12% per year the balance would only have been $3105.85. See if you can vectorize your solution.

# Practice Problem

A student borrows $10,000 to buy a used car. Interest on her loan is compounded at the rate of 2% per month while the outstanding balance of the loan is more than $5000, and at 1% per month otherwise. She pays back $300 every month, except for the last month, when the repay ment must be less than $300. She pays at the end of the month, *after* the interest on the balance has been compounded. The first repayment is made 1 month after the loan is paid out. Write a program which displays a monthly statement of the balance (after the monthly payment has been made), the final payment, and the month of the final payment.

# Practice Problem

When a resistor ($R$), capacitor ($C$), and battery ($V$) are connected in series, a charge $Q$ builds up on the capacitor according to the formula:

$$Q(t) = CV(1 - e^{-t/RC})$$

if there is no charge on the capacitor at time $t=0$. The problem is to monitor the charge on the capacitor every 0.1 s in order to detect when it reaches a level of 8 units of charge, given that $V=9$, $R=4$, and $C=1$. Write a program which displays the time and charge every 0.1 s until the charge first exceeds 8 units (i.e., the last charge displayed must exceed 8). Once you have done this, rewrite the program to display the charge only while it is strictly less than 8 units.

# Practice Problem

A sound engineer has recorded a sound signal from a microphone. The sound signal was sampled, meaning that values at discrete intervals were recorded (rather than a continuous sound signal). The units of each data sample are volts. The microphone was not on at all times, however, so that data samples below a certain threshold are considered to be data values that were samples when the microphone was not on, and therefore not valid data samples. The sound engineer would like to know the average voltage of the sound signal. Write a script that will ask the user for the threshold and the number of data samples, and then for the individual data samples. The program will then print the average and a count of the valid data samples, or an error message if there were no valid data samples. An example of what the input and output would look like in the Command Window is shown:

```
Please enter the threshold below which samples will be
considered to be invalid:
3.0
Please enter the number of data samples to be entered:
7

Please enter a data sample: 0.4
Please enter a data sample: 5.5
Please enter a data sample: 5.0
Please enter a data sample: 2.1
Please enter a data sample: 6.2
Please enter a data sample: 0.3
Please enter a data sample: 5.4

The average of the 4 valid data samples is 5.53 volts.
```

Note: If there had been no valid data samples, the program would print an error message instead of the last line shown.

# Practice Problem

Create a vector of five random integers, each in the range from –10 to 10. Perform each of the following two ways: using built-in functions, and also using loops (with **if** statements if necessary):

- Subtract 3 from each element.
- Count how many are positive.
- Get the absolute value of each element.
- Find the maximum.

# Practice Problem

Create a 3 × 5 matrix. Perform each of the following two ways: using built-in functions, and also using loops (with **if** statements if necessary):

- Find the maximum value in each column.
- Find the maximum value in each row.
- Find the maximum value in the entire matrix.

# Practice Problem

Write a script that will print the following multiplication table:

```
1
2   4
3   6   9
4   8  12  16
5  10  15  20  25
```

# Practice Problem

The Wind Chill Factor (WCF) measures how cold it feels with a given air temperature T (in degrees Fahrenheit) and wind speed (V, in miles per hour). One formula for it is

$$WCF = 35.7 + 0.6T - 35.7(V^{0.16}) + 0.43T(V^{0.16})$$

Write a function to receive the temperature and wind speed as input arguments, and return the WCF. Using loops, print a table showing wind chill factors for temperatures ranging from –20 to 55 in steps of 5, and wind speeds ranging from 0 to 55 in steps of 5. Call the function to calculate each wind speed.

# Practice Problem

A vector *v* stores for several employees of the Green Fuel Cells Corporation their hours worked one week followed for each by the hourly pay rate. For example, if the variable stores

```
>> v
v =
    33.0000 10.5000 40.0000 18.0000 20.0000 7.5000
```

that means the first employee worked 33 hours at $10.50 per hour, the second worked 40 hours at $18 an hour, and so on. Write code that will separate this into two vectors, one that stores the hours worked and another that stores the hourly rates. Then, use the array multiplication operator to create a vector, storing in the new vector the total pay for every employee.

# Practice Problem

The mathematician Euler proved the following:

$$\frac{\pi^2}{6} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \ldots$$

Rather than finding a mathematical proof for this, try to verify whether the conjecture seems to be true or not. Note: There are two basic ways to approach this: either choose a number of terms to add, or loop until the sum is close to $\pi^2/6$.

# Practice Problem

Write a script *echoletters* that will prompt the user for letters of the alphabet and echo-print them until the user enters a character that is not a letter of the alphabet. At that point, the script will print the nonletter, and a count of how many letters were entered. Here are examples of running this script:

```
>> echoletters
Enter a letter: T
Thanks, you entered a T
Enter a letter: a
Thanks, you entered a a
Enter a letter: 8
8 is not a letter
You entered 2 letters
```

```
>> echoletters
Enter a letter: !
! is not a letter
You entered 0 letters
```

# Practice Problem

Write a script called *prtemps* that will prompt the user for a maximum Celsius value in the range from –16 to 20; error-check to make sure it's in that range. Then, print a table showing degrees F and degrees C until this maximum is reached. The first value that exceeds the maximum should not be printed. The table should start at 0 degrees F, and increment by 5 degrees F until the max (in C) is reached. Both temperatures should be printed with a field width of 6 and one decimal place. The formula is C = 5/9 (F – 32).

```
>> prtemps
When prompted, enter a temp in degrees C in range –16
to 20.
Enter a maximum temp: 30
Error! Enter a maximum temp: 9

  F             C
 0.0         -17.8
 5.0         -15.0
10.0         -12.2
15.0          -9.4
20.0          -6.7
25.0          -3.9
30.0          -1.1
35.0           1.7
40.0           4.4
45.0           7.2
```