For Loops in MATLAB

Most of the code we've written so far in MATLAB is executed sequentially: one line after then next. In other words, the control of the program flows sequentially: we starts at the beginning of the code and works our way down, step by step until we reach the end of our code. In such a sequential execution, each line executes exactly one time: we don't skip any lines of code and we do not repeat any lines of code. This type of control flow is the default in MATLAB: unless we use special syntax designed to change the flow of the program, MATLAB will always process our code in sequential form (line by line).

We will encounter circumstances in which we want to depart from the default control flow. When solving programming problems, we may want to *jump* to other locations in our code. We can use such a jump to execute lines of code out of the default order. Instead of executing a single line of code and then immediately moving down to the next line, we can jump to some other place in our program.

One way to do this type of jumping is known as a `for` loop. As the name implies, this programming structure is designed to loop through a particular block of code multiple times. The syntax for using `for` loops in MATLAB is as follows

```
for loop_index = row_vector_of_index_values
    % Loop body
    %   Block of code to be executed multiple times
    %   Includes any executable programming statements
    %   Before we jump back to start of code, loop_index increments
end
```

In this syntax, we have the following values:

`loop_index`

This is a scalar-valued variable that stores the current value for index of the loop. You can think about this variable as serving two separate functions.First, we use the `loop_index` to define a finite, predetermined number of iterations. The number of times we run the code inside the loop body is equal to the number of columns in the row-vector `row_vector_of_index_values`. In other words, we run one iteration of the loop body for each element in `row_vector_of_index_values`. The `loop_index` automatically tracks our progress and ensures that we exit the loop as soon as we've run through all values stored in `row_vector_of_index_values`.

The second purpose of the `loop_index` is to progress through each individual entry stored in the `row_vector_of_index_values` vector. During the first iteration of the `for` loop, the stored value of the `loop_index` variable will be equal to the first entry in the `row_vector_of_index_values`. Once we reach the end of the loop body, the `loop_index` will increment to the next position and store the second entry in the `row_vector_of_index_values`. Once this happens, MATLAB will re-run all the code inside the loop body, step by step, using the updated value for the `loop_index` variable. We continue executing a single iteration of the code in the loop body for each entry of `row_vector_of_index_values`. In this way, the `loop_index` empowers us to move through each desired entry and utilize these in each iteration of the loop body.

`row_vector_of_index_values`

The number of columns of this vector indicates the number of iterations of our loop that we want to run. The individual entries of this vector store our desired values for the `loop_index` variable during each iteration. So, for the first iteration of our `for` loop, the `loop_index` will take the value of the first entry stored in `row_vector_of_index_values`. During the second iteration, the will become the second entry in `row_vector_of_index_values`. Unless we purposely interrupt the `for` loop using some other control structure, this process continues until we work through all entries of this vector and then exist the loop.

loop body

This is the block of code that we'd like to run for a pre-defined number of times. Usually we specifically use the value of the `loop_index` variable during each iteration to accomplish specific tasks, though this is not always necessary. In the absence of other control statements, the code in this loop body executes sequentially (line by line) during each iteration of the `for` loop. At the end of each loop, the value of the `loop_index` variable increments to the next stored value in `row_vector_of_index_values` and MATLAB re-runs the code in the loop body again with the new index variable value.

Let's take a look at this basic `for` loop structure in action.

EXAMPLE 1

Let's consider our first example of a `for` loop in action. In this case, we will simply display the various values of the `loop_index` during each iteration of our `for` loop. To do so, we use the following code:

```
for i = 1:6
    i
end
```

A few features of this code to notice. First, the values stored in the vector `row_vector_of_index_values` are defined using colon notation. We saw in previous work that the code `1:6` produces a 1×6 row vector given by

$$\texttt{1:6} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

Given that the `row_vector_of_index_values` variable is 1×6, the code in our loop body will execute 6 separate times. During the ith iteration, the value of the `loop_index` variable (in this case, we call it `i`) will take on the value stored in the ith entry of `1:6`. We could have achieved the smae results using different syntax

```
for i = [ 1 , 2 , 3 , 4 , 5 , 6 ]
    i
end
```

In each case, the loop body simply displays the value of i during each iteration. One way to make this code a little more fancy is to use MATLAB's built-in `fprintf` function. This function enables us to print out text and also numerical values.

```
for i = 1:1:6
    fprintf('Value of the index variable i is %d. \n', i)
end
```

EXAMPLE 2

Let's see how we can use `for` loops accomplish some fun arithmetic. Suppose we want to create code that counts up from 0 all the way to 8, in increment sizes of 1. To accomplish this type of work, we will use a slightly modified syntax structure:

```
% Initialize variable
for loop_index = row_vector_of_index_values
    % Loop body
    %   Block of code to be executed multiple times
    %   Update initialized variable using loop_index value
    %   Before we jump back to start of code, loop_index increments
end
```

In this case, we want to count starting at 0 all the way up to 8. The code below accomplishes that goal:

```
count = 0;
for i = 1:8
    count = count + 1
end
```

We can do some work to analyze each iteration of this `for` loop. To do so, let's add some text to the loop body and also a `pause statement`

```
count = 0;
for i = 1:1:8
    count = count + 1;
    fprintf('Count value: %d. \n', count)
end
```

We could use the same programming structure to count up from an initial value of 5 to an ending value of 25 by fours:

```
count = 5;
for i = 1:5
    count = count + 4;
    fprintf('Count value: %d. \n', count)
end
```

EXAMPLE 3

Let's use this programming structure to do some calculations for famous sums in mathematics. We'll start with one of the most famous sums from discrete mathematics, the sum of the numbers from 1 to n:

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + (n-1) + n.$$

Let's take a look at the values of this summation for various values of n:

n	Summation	Total sum
1	1	1
2	$1+2$	3
3	$1+2+3$	6
4	$1+2+3+4$	10
5	$1+2+3+4+5$	15
6	$1+2+3+4+5+6$	21
7	$1+2+3+4+5+6+7$	28
8	$1+2+3+4+5+6+7+8$	36
9	$1+2+3+4+5+6+7+8+9$	45
10	$1+2+3+4+5+6+7+8+9+10$	55
n	$\sum_{k=1}^{n} k = 1+2+3+\cdots+(n-1)+n$	$\frac{n(n+1)}{2}$

We can achieve these same results using `for` loops in MATLAB. To do so, we'll use the same syntax structure as in our last example:

```matlab
n = 7;
sum = 0;
for k = 1:n
    sum = sum + k;
end
```

We can now change the value of n and confirm any of the entries on this table. More powerfully, we can change n to a value not on this table and run this summation.

EXAMPLE 4

Let's level up and use this tool to calculate more difficult sums. Specifically, let's calculate the sum of the first n squares:

$$\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + 4^2 + \cdots + (n-1)^2 + n^2 = 1 + 4 + 9 + 16 + \cdots + n^2.$$

Let's take a look at the values of this summation for various values of n:

n	Summation	Total sum
1	1	1
2	$1+4$	5
3	$1+4+9$	14
4	$1+4+9+16$	30
5	$1+4+9+16+25$	55
6	$1+4+9+16+25+36$	91
7	$1+4+9+16+25+36+49$	140
8	$1+4+9+16+25+36+49+64$	204
9	$1+4+9+16+25+36+49+64+81$	285
10	$1+4+9+16+25+36+49+64+81+100$	385
n	$\sum_{k=1}^{n} k^2 = 1^2+2^2+3^2+\cdots+n^2$	$\frac{n(n+1)(2n+1)}{6}$

We can achieve these same results using `for` loops in MATLAB. To do so, we'll use the same syntax structure as in our last example:

```matlab
n = 8;
sum = 0;
for k = 1:n
    sum = sum + k^2;
end
```

We can now change the value of n and confirm any of the entries on this table. More powerfully, we can change n to a value not on this table and run this summation.

EXAMPLE 5

We can also use the same `for` loop structure to cycle through the individual elements of a vector. Let's suppose we wanted to access the individual elements of a 1×5 vector given by

$$\mathbf{x} = \begin{bmatrix} 10 & 5 & 0 & -5 & -10 \end{bmatrix}$$

Remember that each of these elements has a row index

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} = \begin{bmatrix} 10 & 5 & 0 & -5 & -10 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix} = \begin{bmatrix} 10 & 5 & 0 & -5 & -10 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$$

Suppose we wanted to define a new vector **y** such that

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} x_5 \\ x_4 \\ x_2 \\ x_6 \\ x_1 \\ x_3 \end{bmatrix}$$

To do this, we can use the following code:

```
x = 10:-5:-10
for j = 1:length(x)
    x(1, j)
end
```

We can make this a little more fancy using the `fprintf` command:

```
x = 10:-5:-10
for j = 1:length(x)
    fprintf('Entry number %d of vector x is %d, \n', j, x(1, j))
end
```

This would work equally well if we wanted to cycle through the entries of an equivalently define column vector. The only difference is that we have to be careful to track the address of each entry we want to display.

```
x = [10 ; 5 ; 0 ; -5; -10]
for j = 1:length(x)
    x(j, 1)
end
```

Notice if I use the exact same code as I did for the row vector MATLAB gets angry because I am not accessing the data in the form that I stored it.

EXAMPLE 6

This ability to cycle through vector elements is quite powerful. We can use this to store the output of each iteration of a `for` loop in an individual entry of a vector-valued variable. Let's return to our Example 3 in which we sum the first n numbers together. Recall our code to do this was

```
n = 7;
sum = 0;
for k = 1:n
    sum = sum + k;
end
```

Let's modify our code slightly to take advantage of MATLAB's array-based features.

```
n = 7;
sum = 0;
sum_vec = zeros(1, n);
for k = 1:n
    sum = sum + k;
    sum_vec(1,k) = sum;
end
```

After the kth iteration, we store the sum of the first k numbers in the kth entry of the `sum_vec` variable. We can quickly confirm all the values of our table using this construction.

EXAMPLE 7

We are now in a position to use MATLAB `for` loops to define operations between vectors. Suppose we want to add two vectors together

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

Notice that the ith sum is stored in entry $(i, 1)$ with $z_i = x_i + y_i$. We can use this structure to create the sum of two column vectors with equal sizes using a `for` loop

```
n = 10;
x = [1:n]';
y = [n:-1:1]';
sum = zeros(n, 1);
for k = 1:n
    sum(k, 1) = x(k, 1) + y(k, 1)
end
```

EXAMPLE 8

We can also use `for` loops to define scalar-vector multiplication. Suppose we want to multiply a scalar $a \in \mathbb{R}$ by a vector $\mathbf{x} \in \mathbb{R}^n$ with

$$a\mathbf{x} = a \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} ax_1 \\ ax_2 \\ \vdots \\ ax_n \end{bmatrix}$$

Notice that the ith entry stores the product of the scalar a with the ith entry of $\mathbf{x}$: ax_i. Using a `for` loop, we can create the product of a scalar and a vector multiple and store the result as a vector:

```
n = 10;
a = 4;
x = ones(n, 1)
product = zeros(n, 1);
for k = 1:n
    product(k, 1) = a*x(k, 1)
end
```

If Statements in MATLAB

In almost all our work so far, we've executed our code sequentially. We execute each line of code, move on to the next line of code, and so on. In this sequential flow, each line of code is executed exactly one time and we cannot jump to other places in our program nor skip any lines of code.

There are situations in which it can be helpful to break away from sequential flow. We might want to be able to make a choice whether to execute certain lines of code, jump to specific lines of code based on specific conditions, or jump to different locations in our code after making decisions about data stored in memory. We can use programming constructs known as *branching statements* or *selection statements* to do this type of jumping. As we will see, the name for these type of statements comes from the idea that we can create branches of (i.e. select different paths in) our flowchart diagram based on the different options we have in our code.

One basic tool for making choices in MATLAB is know as the `if` statement. We will study four variations of this statement, including:

A. The `if` - `end` structure

B. The `if` - `else` structure

C. The `if` - `elseif` - `else` - `end` structure

D. Nested `if`-`else`-`end` statements

Each of these expressions take advantage of MATLAB's logical data class. Specifically, using logical and relational operations we create logical scalars whose value we use to make decisions. Specifically, the entire idea of the `if` statement is to execute certain lines of code if the value of a particular logical scalar is `true` or jump to some other line of code if the value is `false`.

The simple if statement

Let's start with the most basic if statement. The syntax for this type of programming structure is as follows:

```
if logical_scalar
    % Body of if statement
    %   Block of code to be executed if logical_scalar is true
    %   Includes any executable programming statements
    %   If logical_scalar is false, we jump over (skip) this code
end
```

This construct has some interesting features:

logical_scalar

> In the single if statement, we use a logical scalar as a condition to make decisions. We might write conditional expressions using logical or relational operations. We might also detect the state of some variable.
>
> Recall from our previous work that a *logical scalar* is an expression in MATLAB that produces an output in the form of a 1-by-1 logical array (a scalar). A logical scalar can have only one assigned value: either true with logical value 1 or false with logical value 0. When the expression is evaluated, the single scalar-valued logical output is assigned based on the truth value of the expression.
>
> 1 : If the logical_scalar evaluates to 1, we execute all statements inside the body of the if statement.
>
> 0 : If the logical_scalar evaluates to 0, we jump over the entire body of the if statement and continue to the next lines of code after the end of our statement.

body of if statement

> This is the block of code between the if logical_scalar condition and the end of our statement. The body of the code can include any executable programming statements. The major point is that we only run the code in the body of the if statement when the logical_scalar condition achieves the value of 1 (or true).

Let's take a look at an example of this basic `if` statement in practice action.

EXAMPLE 1

More on this example here.

The if - else structure

We continue our exploration with if - else statements. The simple if statement chooses whether or not to execute one single block of code inside the body of the if statement. Sometimes we might want to choose between two different blocks of code: one block for true statements and a separate block for false statements. We can achieve this using an if - else statement. The syntax for if - else statements are follows:

```
if logical_scalar
    % Code block 1
    %   Block of code to be executed if logical_scalar is true
    %   Includes any executable programming statements
else
    %Code block 2
    %   Block of code to be executed if logical_scalar is false
    %   Includes any executable programming statements
end
```

This construct has some interesting features:

logical_scalar

In the single if statement, we use a logical scalar as a condition to make decisions. We might write conditional expressions using logical or relational operations. We might also detect the state of some variable.

Recall from our previous work that a *logical scalar* is an expression in MATLAB that produces an output in the form of a 1-by-1 logical array (a scalar). A logical scalar can have only one assigned value: either true with logical value 1 or false with logical value 0. When the expression is evaluated, the single scalar-valued logical output is assigned based on the truth value of the expression.

1 : If the logical_scalar evaluates to 1, we execute all statements inside code block 1.

0 : If the logical_scalar evaluates to 0, we execute all statements inside code block 2.

Code block 1

This is the block of code between the if logical_scalar condition and the else keyword in our statement. This code block can include any executable programming statements. The major point is that we only run code block 1 when the logical_scalar condition achieves the value of 1 (or true).

Code block 2

This is the block of code between the else and end keywords in our statement. Again, we can write any executable programming statements in code block 2. The lines of code in block 2 will only run if the logical_scalar condition achieves the value of 0 (or false).

1. Video 1: Simple `if` statements
2. Video 2: `if`, `else` statements
3. Video 3: `if`, `elseif`, `else` statements