## Contents



# REPETITIVE EXECUTION: LOOPS

Loops are useful when you want a certain set of commands executed repeatedly. You control the number of times the commands inside the loop are executed by either (A) specifying a starting value and ending value (and optionally, an increment size) for the loop; or (B) specifying a condition such that the loop will continue as long as the condition remains true.

There are two types of loops in MATLAB: `for` loops and `while` loops. These are discussed in detail in the following sections.

## `for` Loops

Using a `for` loops is useful when you need to perform a task a certain amount of times. The basic structure of a `for` loop is as follows:

```
for index = A:C:B
    commands/statements/calculations
end
```

Where the index is a variable that will take on the values given by `A:C:B` (one at a time) as it goes through the loop repeatedly, executing all statements inside the loops with each pass. The first value of the index will be `A`. The index variable does not need to be initialized before a `for` loop – the `for` loop will create the index variable if it does not exist, or overwrite it if it already has a value. Often, it is useful to think of `A`, `B`, and `C` in the following way:

```
for index = initial_value:increment_size:final_value
     statements/commands/calculations
end
```

Note that the notation for creating a `for` loop index is identical to that used to create an array using the colon operator as discussed previously. The increment is always optional – if you do not provide an increment the default step size is `1`. Notice that a `for` loop is always started with the keyword `for` and always terminated with an `end`. As was the case with `if` statements, the indentation of the commands inside the loop is optional but highly recommended.

Let's say that `A=1`, `C=2`, and `B=5` in our notation above. This gives `1:2:7`, which means that the loop index will take on the values `1 3 5 7`, one at a time, in that order. Note that `A`, `B`, and `C` can be existing variables stored in memory, or simply numbers that you hard-code when creating the loop. If the values of `A`, `B`, and `C` are parameters that are used throughout your code, it is a good idea to define them as variables in the beginning of your M-File, rather than hard-coding their values on the loops. That way, if the parameters change, you only need to update the values in one location, rather than on every loop throughout the M-File. In general, it is a good idea to define all parameters needed for your calculations as variables in the beginning of your code (with comments to provide a description, document units, etc.); avoid hard-coding values whenever possible.

When the `for` loop begins, the first value of index will be the initial value provided, which in this example is `1`. After each pass through the loop, the index changes value based on the provided step size, `C`. So, after the first pass through the loop, when all the commands between the `for` and the `end` have been executed, the index changes value to `3` and passes through the loop again starting at the beginning, meaning all the commands inside the loop are executed again. The loop would go through four passes in this case, and the index will have a different value each pass through the loop. All commands inside the loop are executed with each pass, regardless of the value of the index.

Let's look at a simple example where we create a loop and print out the loop index with each pass. Note that the loop index is ***not*** printed automatically by the loop – if we want it to be output, we must add the command(s) to do so inside the loop.

In `myfile.m`:

```
n = 5;
for i = 1:n     % The loop will go through 5 passes
    disp(i)
end
```

Sample Output:

```
     1

     2

     3

     4

     5
```

In this example, we did not include a step size, so the default increment of `1` was utilized. Note that the value of the index changed with each pass through the loop. Let use the same loop but change what we print inside of the loop.

In `myfile.m`:

```
n = 5;
for i = 1:n     % The loop will go through 5 passes
    disp(i^2)
end
```

Sample Output:

```
     1

     4

     9

    16

    25
```

Note that `i` still takes on the same values (`1  2  3  4  5`), but we are now printing $i^2$ instead of just `i`, so the output changes.

Let's now look at an example where we include an increment that is not the default value of 1. Again, we will print the loop index inside the loop to see what values it takes with each pass through the loop.

In `myfile.m`:

```
for i = 1:2:7 % The loop will go through 4 passes
    disp(i)
end
```

Sample Output:

```
1

3

5

7
```

The variable `i` is `1`, then `3`, then `5`, then `7`. Note that the loop index is **not** an array of values (although it appears to be written in array-like notation) – it takes on only one value at a time, and its value changes with each pass through the loop. When the loop finishes, the index's final value is still stored in memory. Let's look at an example to illustrate this point:

In `myfile.m`:

```
for i = 1:2:7  % The loop will go through 4 passes
    disp(['inside loop, i = ',num2str(i)])
end
disp(['after loop, i = ',num2str(i)])
```

Sample Output:

```
inside loop, i = 1
inside loop, i = 3
inside loop, i = 5
inside loop, i = 7
after loop, i = 7
```

Let's look at one more example that uses a step size other than `1`.

In `myfile.m`:

```
for i = 0:3:10 % the final value can NOT be reached
    disp(['inside loop, i = ',num2str(i)])
end
disp(['after loop, i = ',num2str(i)])
```

Sample Output:

```
inside loop, i = 0
inside loop, i = 3
inside loop, i = 6
inside loop, i = 9
after loop, i = 9
```

Take a careful look at the output. You will notice that the loop index, `i`, never reached the final value that we provided on the loop (`10`). The same rules that apply to creating arrays with the colon operator apply to `for` loop indices. That is:

1. MATLAB will never use an increment/step size other than the one you provide (i.e., a full step must be taken).
2. MATLAB will never go beyond ("overshoot") the final value that you provide.

Thus, in the above example, the final value of the loop index is `9` because rule #2 prevents MATLAB from taking the next full step to `12`, and rule #1 prevents MATLAB from using any step size other than what was given (`3`).

You can also use the loop index variable to loop through elements of an array. This can be used to create an array one element at a time (example below) or access elements of an existing array that is already stored in memory. This is one of the most powerful applications of loops, and will be required throughout MAE10.

In `myfile.m`:

```
for i = 1:5
    a(i) = i^2;
end
a
```

Sample Output:

```
a =

     1     4     9    16    25
```

Example Practice Problem:

Calculate the average test score from an array of values without using the built-in `mean()` function. Assume that the scores are given by the following array: `scores = [76,100,90,99,91]`. Before proceeding with the notes, you should attempt to solve this problem on your own using a `for` loop.

Sample Solution:

```
scores = [76,100,90,99,91];
my_sum = 0; % Notice that we must initialize our sum variable to zero
for i = 1:numel(scores)
    my_sum = my_sum + scores(i);
end
my_avg = my_sum / numel(scores);
disp(['The average score is: ',num2str(my_avg)])
```

Sample Output:

```
The average score is: 91.2
```

In MATLAB, we also have the option of using the built-in `mean()` function to calculate the average score from an array of values. However, many programming languages do not have this handy function and you will need to use a loop to compute the average using your own algorithm like the one shown above. Be sure to read the instructions for any MAE10 problem carefully as you may be restricted on using certain built-in functions that are unique to MATLAB.

### `while` Loops

Although they have a different form and syntax, `while` loops can do essentially anything `for` loops can do. Rather than indicating the beginning/end of the loop with initial/final values, a `while` loop uses a test condition to determine when to start/stop. The test condition(s) that you use on a `while` loop are created using the same relational and logical operators used on `if` statements.
The basic structure of a `while` loop is as follows:

```
while (test condition)
     statements/commands/calculations
end
```

Let's look at a simple example, where we produce the same output as our first `for` loop example but this time use a `while` loop. This will help illustrate the similarities and differences between the two loop types, and show how either can typically be used to achieve the same result. Note the differences between the M-File used in this example and the one used in the previous `for` loop example.

In `myfile.m`:

```
x = 1;
while (x<=5)
    disp(x)
    x = x + 1;
end
```

Sample Output:

```
     1

     2

     3

     4

     5
```

Notice that when we use a `while` loop, we must initialize the variable that we plan to use with the test condition – it must have a value in memory or else an error will be produced when trying to check the test condition (e.g., if it is less than or equal to `5` in this example). In contrast, a `for` loop automatically creates the loop index variable even if it does not exist prior to the loop. A `for` loop is better suited to handle the task above since we know the initial value, final value, and step size ahead of time. However, you can use a `while` loop if you like. Let's look a sample problem that is best suited for a `while` loop.

Example Practice Problem:
Prompt the user to enter a test score on a 0-100 scale. Allow the user to continue entering test scores, one at a time, until they are finished. You do not know ahead of time how many scores the user needs to enter – they must be prompted to enter a score and should be able to continue entering scores until they choose to stop. Store each score value in a separate element of an array called `scores`. Solve this problem on your own before proceeding with the notes.

In `myfile.m`:

```
answer = 'y';
i = 0;
while (answer == 'y' | answer == 'Y')
    i = i + 1;
    scores(i) = input('Enter a score: ');
    answer = input('Enter another score (Y or N)? ','s');
end
disp('Here are the values you entered:')
disp(scores)
```

Sample Output:

```
Enter a score: 57
Do you want to enter another score (Y or N)? Y
Enter a score: 87
Do you want to enter another score (Y or N)? Y
Enter a score: 98
Do you want to enter another score (Y or N)? n
Here are the values you entered:
    57    87    98
```

Notice that in the above example we use two test conditions on our while loop just to be safe. Even if the user enters a lowercase `y` instead of a capital `Y` as prompted, the loop will continue. This is good practice – always write your codes to be as flexible and user friendly as possible.

**The `break` and `continue` Commands**

The `break` command will terminate a loop prematurely. It can be placed anywhere inside a `for` loop or a `while` loop and it will terminate the loop as soon as it is executed. Let's look at an example to illustrate.

In `myfile.m`:

```
n = 5;
for i=1:n
    if (i==3) % when i is 3, the break is executed
        break
    end
    disp(['inside loop, i = ',num2str(i)])
end
disp(['after loop, i = ',num2str(i)])
```

Sample Output:

```
inside loop, i = 1
inside loop, i = 2
after loop, i = 3
```

Look carefully at the output. You should notice a couple things right away:

- When `i=3`, the `if` condition becomes true and the `break` is executed. The loop is terminated *immediately*, meaning that the `disp()` command inside the loop is *not* executed when `i=3`.
- Notice that the final value of the loop index is `3`, even though technically we had a final value of `5` given on the loop. Once again, the `break` terminated the loop early and the loop index retains its last value in memory.

The `continue` command jumps back to the beginning of a loop and increments the loop index by the given step size. Let's look at an example that is identical to the one above but replaces `break` with `continue`.

In `myfile.m`:

```
n = 5;
for i=1:n
    if (i==3) % when i is 3, the continue is executed
        continue
    end
    disp(['inside loop, i = ',num2str(i)])
end
disp(['after loop, i = ',num2str(i)])
```

Sample Output:

```
inside loop, i = 1
inside loop, i = 2
inside loop, i = 4
inside loop, i = 5
after loop, i = 5
```

Look carefully at the output again and compare it with the output produced by the previous example that uses `break`.

- When `i=3`, the `if` condition becomes true and the `continue` is executed. The loop index *immediately* takes the next step (an increment of 1 in this case) to `i=4`, meaning that the `disp()` command inside the loop is *not* executed when `i=3` since it appears after the

`continue`. Notice that a `continue` can cause certain statements inside the loop to be skipped on that pass, since it immediately jumps back to the beginning of the loop, essentially skipping any commands between the `continue` and the `end`.
    - In this example, the loop resumes with `i=4` and the loop continues, since the `continue` command does not terminate the loop entirely.
- Notice that the final value of the loop index is `5` – the `continue` did not change the final value of the loop index, it just "skipped" some statements inside the loop when the loop index had a certain value (`3` in this example).

You might be wondering what some potential applications are for `break` and `continue`. Here are a couple things to consider:

- A `break` command inside an `if` statement can serve as a fail-safe when you are performing calculations.
    - For example, if you are calculating values inside of a loop that you know should not be negative (e.g., concentration, temperature in kelvin, etc.), it might be a good idea to include an `if/break` inside of the loop to terminate the loop if the values become negative.
        - In practice, codes can take hours or even days to execute. Including such a fail-safe could save you precious time by stopping your code early if something goes wrong. Otherwise, you may end up waiting for your code to run only to find out that a mistake caused the results to become incorrect early on during the calculations.
- A `continue` command inside an `if` statement can perform a similar function, but allows the loop to resume instead of terminating altogether.
    - For example, you may ask the user to enter a value at the beginning of your loop, which is then used in some calculations. If the user enters the value incorrectly (e.g., it must be positive and it is not), it is desirable to simply restart the loop and ask for the value again (before the calculations are performed), rather than terminating the loop entirely.
        - This is analogous to any prompt you see in an online form or elsewhere where you must enter some specific information. If you do not enter correctly, it simply asks you for the information again before proceeding.

**Nested Loops**

It is often necessary to use two or more loops together to solve a given problem. A nested loop refers to one loop inside of another loop. There are no restrictions on nesting loops; `for` loops can be placed inside of `while` loops, other `for` loops, etc. One of the main applications of using nested loops is to work with 2D (or larger) arrays. Since 2D arrays have both rows and columns, we use a nested loop

structure to go through all elements of the array – one loop for the rows and one loop for the columns. Let's look at a quick example to illustrate.

In `myfile.m`:

```
A = [1 2 3;4 5 6;7 8 9]
[m,n] = size(A);
for i = 1:m
    for j = 1:n
        disp(['A(',num2str(i),',',num2str(j),') = ',num2str(A(i,j))])
    end
end
```

Sample Output:

```
A =

     1     2     3
     4     5     6
     7     8     9

A(1,1) = 1
A(1,2) = 2
A(1,3) = 3
A(2,1) = 4
A(2,2) = 5
A(2,3) = 6
A(3,1) = 7
A(3,2) = 8
A(3,3) = 9
```

In this example, we create a `3x3` array `A`, then use a nested loop structure to go through all of the elements of the array one at a time. Using the `disp()` command, we produce some nicely formatted output that shows us not only the value in `A`, but also its position (given by row and column numbers). Remember that 2D arrays are indexed by `(row,col)` notation, so in the above example our `i` loop goes through the rows of `A` while the `j` loop goes through the columns.

**Notice that for each `i`, the `j` loop is executed entirely**. Once the inner `j` loop finishes, the outer `i` loop takes the next step, and the `j` loop starts over. Lastly, note that we used the `size()` command to determine the size of `A`, then used these variables (the number of rows and columns) as the final values for our loops. This is **strongly recommend** – do not simply look at `A` and hard-code the number of rows and columns on your loops. Always make your code as flexible as possible – if you use `size()`

and you happen to change the original array `A`, your code will automatically adapt to the changes and will not require any other modifications.

Let's look at one more example of nested loops that may be helpful when you are working on homework 4. You should attempt this problem on your own before looking at the sample solution.

Two quizzes are given in a particular class to five students. The scores for quiz #1 are stored in the first row of a table and the scores for quiz #2 are stored in the second row. Compute the overall average quiz score (of both quizzes) using loops. Check your answer with the built-in `mean()` function.

| | Student 1 | Student 2 | Student 3 | Student 4 | Student 5 |
|---|---|---|---|---|---|
| Quiz #1 | 99 | 66 | 99 | 83 | 98 |
| Quiz #2 | 93 | 87 | 100 | 89 | 100 |

In `myfile.m`:

```
scores = [90, 66, 99, 83, 98 ; 93, 87, 100, 89, 100]
[nrow,ncol] = size(scores);
my_sum = 0;
for i=1:nrow
    for j=1:ncol
        my_sum = my_sum + scores(i,j);
    end
end
average = my_sum/(nrow*ncol);
disp('Average using nested loops:')
disp(average)
disp('Average using mean() function:')
disp( mean( scores(:) ) )
```

Sample Output:

```
scores =

    90    66    99    83    98
    93    87   100    89   100

Average using nested loops:
   90.5000

Average using mean() function:
   90.5000
```

**Distinguishing Characteristics of `for` and `while` Loops – Things to Remember**

Using `for` loops is a better choice when you need to do something `N` times. In other words, if you know the final value and/or you know exactly how many iterations you need, a `for` loop is easier to use than a `while` loop.

Using `while` loops are better when you have a test condition for breaking the loop. In this case, you may not know the final value or how many passes through the loop are necessary to solve the problem. A `while` loop will continue executing as long as the condition on the loop remains true. Although it may be easier to use one type of loop in a given situation, in almost all instances either type of loop can be used.

Here is a summary of some key points to remember when using loops:

- "`for`" and "`while`" are case sensitive and must be all lowercase (same with "`end`").
- If you are using the loop index to access elements of an array, be sure that the loop index is a positive integer.
- When a `for` loop finishes, the value of the loop index on the final pass through the loop will be stored in memory.
    - Remember that the loop index variable is not an array, but a scalar that takes on one value at a time with each pass through the loop.
- When using a `for` loop, the loop index does not need to exist prior to the loop – the `for` loop creates the loop index variable if it does not exist (i.e., no need to initialize a value for the loop index variable prior to the loop).
- When using a `while` loop, you must initialize the variable that is used for your test condition.
    - The variable should be initialized such that the condition on the loop is initially true. Otherwise, the loop will be skipped entirely.
    - You must have some commands/statements/calculations inside of the `while` loop that eventually cause the test condition to become false. This means that the value of the variable that is used for the test condition must change inside the loop. Otherwise, you will end up with an infinite loop.
- A `for` loop automatically changes the value of the loop index with each pass through the loop, but a `while` loop does not.
- It is good practice to write loops as generally as possible; avoid hard-coding numbers on loops whenever possible.
    - For example, if you are looping through elements of a 1D array (as in the above `for` loop problem), you should use `numel(array)` as the final value of your loop, rather than counting the number of elements of the array and hard-coding that value. In this

way, if the array is changed, all loops automatically adapt and no other changes in your code are required.
        - For 2D arrays, you can use `size()` to determine the number of rows and columns, then used a nested loop structure to go through all elements of the array (by row/column).
- The loop index is not printed automatically when using a loop. Thus, a `for` loop with no statements/commands inside will produce no output (although the loop index will still be stored in memory as described above).
- The `break` and `continue` commands can be used in either type of loop, but are most commonly used in `for` loops.
    - The `break` command will terminate a loop, skipping any commands between the `break` and the end of the loop.
        - If you are using nested loops, it terminates only the inner most loop (in which the `break` is located).
    - The `continue` command will cause the code to jump back to the beginning of the loop and start another pass, taking the next step with the loop index if it is a `for` loop.
        - All commands between the `continue` and the end of the loop will be skipped once the `continue` is reached – it immediately goes back to the beginning of the loop.
        - If you are using nested loops, it applies to only the inner most loop (in which the `continue` is located).
- When using nested loops, loops are executed from the inside out. That is, the inner most loop executes in its entirety before the next step is taken on any outer loop(s). After the next step is taken on the outer loop(s), the inner loop restarts at its initial value and executes in its entirety once again.
    - This means that for each pass of the outer loop(s), the inner loops(s) will make several passes (determine by their initial value, final value, and step size). This is best illustrated in the example that uses the 2D `A(i,j)` array.