Looping Structures

Leigh J. Little SUNY Brockport September 25, 2020

Contents

1 Introduction

We have seen that it is possible to do more elaborate computation with MATLAB using `if-then` logic. Looping structures will permit even more flexibility in the programs that can be written.

2 Looping operations

Looping operations are used whenever a calculation needs to be performed multiple times.

2.1 `for` loops

Consider the MATLAB `sum` function when applied to a vector. This operation will produce a scalar that is the sum of the elements of a vector. The need for looping operations will be motivated by looking under the hood at what happens behind the scenes of the `sum` function.

Suppose `x` is a vector of length 5 whose element sum is desired. One way to do this would be to do

```
>> x = [5 3 -2 1 8];
>> total = x(1) + x(2) + x(3) + x(4) + x(5)
total =
    15
```

This will work, but it is not a good general approach. For one thing, it only works for a vector of length 5. Moreover, if the vector had 10,000 elements it would take forever to type this in (and you would probably make a few typo's along the way). hat is needed is a general framework that would permit the elements to be summed using the same number of programming statements regardless of how many elements are in the vector.

The solution to this is known as an *accumulation operation* and the framework is called a *loop*. Before writing a version of the `sum` function, it is necessary to examine how a loop works. Enter the following into a script file called `testloop1.m`

```
clear

n = input('Input a value for n ')
for i = 1:1:n
    i
end
```

If the script is run with an input a value of 5 for `n`, the following output is obtained:

```
>> testloop1
Input a value for n 5
i =
    1
i =
    2
i =
    3
i =
    4
i =
    5
```

Run the script for several other small values of `n`. What do you see? This is an example of a `for` loop. In general, a `for` loop has the syntax

```
for i = startvalue : stepsize : endvalue
   |
   |   Body of Loop
   |
end
```

Here,

- `i` is called the loop index variable (or loop index). This is a variable name that you choose.
- One of the most important uses of a loop index is as a subscript in a vector or matrix.
- `startvalue` is the starting value of the index `i`.
- `stepsize` is the step increment.
- `endvalue` is the terminal value of `i`.
- The statement after the `=` sign is an array index.
- The bodies of loops should be indented for readability.

When a loop executes, the flow of the program continues in the following way:

- MATLAB sets the index variable (`i`) to the `startvalue`.
- The statements inside the loop (called the loop body) are executed.
- When the end of the loop is reached (the `end` statement) MATLAB increments the value of the index by `stepsize`.
- MATLAB then performs a test:
  - If the new value of the index is greater than `endvalue`, the loop exits. This means that the program continues with the first executable statement past the `end` statement.
  - If the new value of the index is less than or equal to `endvalue`, the loop returns to the first statement in the loop body.

Note that if the `testloop1.m` script is run for a negative value of `n`, nothing is output. This is because there is no way to go from a starting value of 1 to a negative ending value using a step size of 1. The array index is empty, so this becomes an *empty loop*. An empty loop is one that never executes because the controlling array index is empty.

2.2 An Accumulation Loop to Sum the Elements of a Vector

Now that we know how a loop works, we can write a process for summing the elements of a vector. This capability is already built into MATLAB (via the `sum` function), but accumulation loops form the basis of a great many calculations for which there is not a built-in function. Examining how the calculation works in situations where we can easily check the answers is a good exercise.

2.2.1 Example 1: Accumulation Loop

Another name for an accumulation loop is a running total. Suppose we have some variable `total` and a vector

```
x = [1 5 7 -6 9 3 -1]
```

We want to store the sum of the vector elements in `total`. Before doing this, we need to set the value of `total` to zero. Think of `total` as a box. We need to empty the box before starting the process.

```
total = 0;
```

Now, we sweep over the elements of `x` using a loop. Each pass through the loop, we add the next element of `x` to the existing total. The step-by-step calculation is shown below:

```
total = 0;                          % empty the box
total = total + x(1) = 0 + 1 = 1    % throw the new term in the box and add
total = total + x(2) = 1 + 5 = 6
total = total + x(3) = 6 + 7 = 13
total = total + x(4) = 13 + -6 = 7
total = total + x(5) = 7 + 9 = 16
total = total + x(6) = 16 + 3 = 19
total = total + x(7) = 19 + -1 = 18
```

Note that the statement

```
total = total + x(1)
```

makes no sense algebraically unless `x(1)` is zero. Instead this statement should be read as

```
new value of total = current value of total + x(1);
```

This emphasizes that the `=` sign is used to assign a value to a variable and it not a mathematical statement of equality.

Most importantly, note that there is a pattern to the subscripts in the above calculation. We can create a `for` loop that runs from 1 to 7 in steps of 1 and use the index variable of the loop as the subscript of `x`. This will pick up each element of the vector.

The accumulation process then looks like

```
total = 0;                          % empty the box
for i = 1:1:7
    total = total + x(i);           % throw next term in the box and add
end
total
```

Notice that this process works for a vector of any length. If `x` has 10 or 10 million elements, this same code will work. A more general version of the algorithm would be

```
total = 0;                          % empty the box
for i = 1:length(x)
    total = total + x(i);           % throw next term in the box and add
end
total
```

Here we have used the `length(x)` function to determine the upper limit of the loop index `i` and simplified the loop control portion because the step size is 1.

2.3 Nesting of Loops

Loops can be nested, but each must have its own `end` statement.

```
for i =  start_i : step_i : end_i
   |
   | Body of i loop
   |
   for j = start_j : step_j : end_j
      |
      |   Body of j Loop
      |
   end   % end for j loop
   |
   | Body of i loop
   |
end    % end for i loop
```

Loops can be nested up to 7 levels deep. Also, as with `if-then` statements, nested loops must logically end in the block they begin in.

2.4 Leave the Loop Index Variable Alone

Important! You should *never* change the value of the loop index variable yourself. You can use the value of the index variable in calculations, but you should never have a statement like

```
for i =  start_i : step_i : end_i

   i = sin(a/b)      %%     Don't change the value of i yourself.
                     %%     i should never appear on the left side of an = sign.
                     %%        in the body of the loop

end
```

inside a `for` loop. This leads to unpredictable behavior of your loop and is a terrible idea from the point of view of good programming practice. Unfortunately, MATLAB will allow you to do this. Fortunately, I will not. If I see you doing this in your programs I will mark lots of points off.

2.5 Example 2: Mathematical Sums

One attractive feature of the loop structure is that it makes evaluating sums very easy because the mathematical statement of the sum translates almost directly into programming statements. For example, consider the sum

$$\sum_{k=1}^{5} k^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2.$$

The limits on the sum are exactly the values we want to use for the starting and ending values of the loop index variable. We can translate this into MATLAB code using

```
total = 0;
for k = 1:5
    total = total + k^2;
end
t1 = ['Sum value = ' num2str(total)];
disp(t1)
```

This is another example of an *accumulation loop* and the variable `total` is the *accumulation variable*.

2.6 Example 3: Piecewise Function Evaluation

We have seen that to tabulate a simple function, we can use a whole array operation. For example,

```
x = linspace(0,2*pi,51)';
y = sin(x);
```

will generate a table of the sine function for 51 equally spaced `x` values from 0 to 2π. However, this approach will not work for a function like

$$y = \begin{cases} x^2 & x \geq 0 \\ x^3 - x + 5 & x < 0 \end{cases}$$

For this function, there are two possible values of y depending on the value of x. While this function cannot be tabulated using a whole array operation, you can use a loop to test each individual value of x and then compute the corresponding value of y.

```
x = linspace(-1,1,21)';
y = zeros(size(x));        % Create a vector of zeros the same size as x.
                           % This is not required, but is assures that
                           % y will have the same orientation as x.
                           % This will also make the code run faster
                           % if the vectors are large.
for i  = 1:length(x)
    if(x(i) < 0)
       y(i) = x(i)^3 - x(i) + 5;
    else
       y(i) = x(i)^2;
    end
end
```

Some comments on this program:

- This loop has an `if-then` statement in its body.
- Notice that in the formulas, we apply the necessary calculation to each individual element `x(i)` of the vector `x` rather than the entire vector.
- Also notice that we store the calculation in the vector `y` so that each `x(i)` has a corresponding `y(i)` that goes with it.
- Whenever you start storing elements in a vector, MATLAB will assume the vector is a row vector unless you tell it otherwise. Although not necessary, it's good programming practice for `x` and `y` to have the same orientation. One way to contend with this is to set aside space for `y` before entering the loop by adding the command

  ```
  y = zeros(size(x));
  ```

  This will set the vector `y` to a vector of all zeros that has the same size and orientation as `x`.

2.7 `while` loops

`for` loops are used whenever you know in advance (or have an upper bound on) the number of times that a loop must execute. There are situations when the number of times that a loop must execute is unknown (for example, asking the user to input numbers until a negative value is input). In these cases, a `while` loop is useful. A `while` loop has the following structure

```
while (conditional test)
   |
   | Body of while loop
   |
end
```

The program flow for a `while` loop is as follows:

- When the `while` statement is encountered, MATLAB performs the conditional test.
- If the test is false, the loop is empty and will not execute. Flow continues to the first executable statement past the `end` statement.

- If the test is true, the statements in the body of the loop are executed.
- When the bottom of the loop body is reached, MATLAB returns to the top of the loop and performs the conditional test again. If the test is still true, the loop will execute the body of the loop again. If the test is false, the loop terminates and flow continues to the first executable statement past the `end` statement.
- It is critical that the conditional statement become false at some point, otherwise the loop is infinite (*i.e.,* a loop that never terminates). A good example of an infinite loop is

  ```
  lather
  rinse
  repeat
  ```

  This illustrates an important notion about computers. As humans, we mentally append some implicit, but critical instructions to the last statement. Computers do exactly what you tell them to do and it is easy to either tell a computer to do the wrong thing or to make incorrect assumptions about the behavior of the programs we write.

Consider the example below. Enter the statements into a file called `testloop2.m` and execute them

```
clear

i = 0
while (i < 10)
   i = i + 1
end
```

As with `if-then` statements, you can nest `while` loops within `for` loops an vice-versa. However, you can nest all of theses structures together in nearly any way you wish. You can have `if-then` statements in the bodies of `for` loops within the bodies of `if-then` statements within the bodies of `while` loops. This is where the complexity of programming arises. While we now have all the tools we need to write almost any program we want, learning how to make use of `if-then` and looping structures will comprise the remainder of the semester.

2.8 Example 4: Add a Series of User-input Positive Values

The program below will sum the numbers that the user enters at the keyboard until a negative value is entered

```
total = 0;
flag = 0;
while (flag == 0)
    t = input('Input next value ');
    if(t < 0)
       flag = 1;
    else
       total = total + t;
    end
end

ts = ['Total of values entered = ' num2str(total)];
disp(ts)
```

In this example, the variable `flag` is used to terminate the loop. Prior to entering the loop, `flag` is to zero. The loop will continue to execute for a long as `flag` remains zero. If the user enters a negative value for `t`, rather than include its value in total the `if-then` test will set `flag` to 1. This will cause the loop to exit.