Looping Structures

Leigh J. Little SUNY Brockport September 28, 2018

Contents

1 Introduction

We have seen that it is possible to do elaborate computation with MATLAB using basic linear algebra and whole array operations. However, sooner or later you will encounter a need to do calculations that can't be performed using just these capabilites. You will need to be able to perform more fundamental programming operations. Over the next few lectures, we will examine two basic programming constructs (loops and conditional branching) that will allow you to perform nearly any calculation.

2 Looping operations

Looping operations are used whenever you have a calculation that 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. We will motivate the need for looping operations by looking under the hood at what happens behind the scenes when you use the `sum` function.

Suppose we have a vector of length 5 and we want to add up all the elements. 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. What we need 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 we write our own version of the `sum` function, we will examine how a loop works. Enter the following into a script file called `testloop.m`

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

If you run your script and input a value of 5 for `n`, you see the following output

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

Run your 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)
- `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 contiues 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 `testloop.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 contolling array index is empty.

As in other languages, 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
```

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

Important! You should *never* change the value of the loop index variable yourself. This means that your should never have a statement like

```
for i =  start_i : step_i : end_i

   i = .....

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 allw you to do this. Fortunately, I will not. If I see you doing this in your programs I will mark lots of points off.

2.2 Example 1: Accumulation Loop

Returning to the oiginal motivation for the looping operation; pretend for the time being that we need to sum of the elements of a vector, but we don't have a `sum` function. This can be accomplished using an *accumulation process*. Accumulation processes are very common in programming.

```
x = some vector

total = 0;
for i = 1:length(x)
    total = total + x(i);
end

ts = ['Total of vector x = ' num2str(total)];
disp(ts)
```

The loop in the middle is the accumulation loop and `total` is called the accumulation variable. Prior to entering the loop, `total` is set to some known value. This is usually zero (but not always). In the body of the loop, each individual element of the vector `x` (`x(i)`) is added to the existing value of `total`. When the end of the vector is reached (*i.e.*, when the loop terminates) `total` will contain the sum of all elements of `x`.

We have used the idea of overwriting some variable with a new value at several points in the semester already, however this is a good place for a reminder that assignment statements are not equations. The statement

```
total = total + x(i);
```

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

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

2.3 Example 2: 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)';
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 immediately start storing the answers 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. This means that as coded, `y` will be a row vector while `x` could be a row or a column vector.
- For this type of calculation, 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`.

2.4 `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 situation 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` statment is encountered, MATLAB performs the conditional test.
- If the test is false, the loop is empty and will not execute.
- If the test is true, the statements in the body of the loop are executed.
- At the end of the loop, MATLAB performs the conditional test again. If the test is still true, the loop will execute 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 infinte (*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

```
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` statments 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, leaarning how to make use of `if-then` and looping structures will comprise the remainder of the semester.

2.5 Example: 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`, raher than include its value in total the `if-then` test will set `flag` to 1. This will cause the loop to exit.

3 Summary so Far

At this point in the semester, we have reviewed most of the basic functionality of MATLAB, however we have also covered what would be the essential elements of any programming language. It turns out, all programming languges do more or less the same things. What differs between them is the programming environment, the built-in command set and the command syntax. For example, if you wanted to learn another language, you would need to learn the following:

- The programming environment.
- The types of variables and how to use them.

- How to define and work with scalars, vectors and matrices.
- The syntax for `if-then` structures.
- The syntax for looping structures.
- The syntax for creating your own functions.

Once these are known, you can write programs in many different languages.

There is one important topic we have not discussed and that is character variables. Character variables are used in situations where your data is not numerical but rather consists of values such as names, addresses, phone numbers, *etc.*. We will look at these later in the semester.