Basic Programming

Table of Contents

Introduction 1
Displaying Stuff - `disp` 1
Condition - `if` 1
Loop - `for` 2
More Examples of Loop - `for` 5
Using `for` Loops to Store Answer in Vectors 6
Loop - `while` 7
The General Idea when Comparing Successive Values 10
Closing 11

Introduction

If you've had any programming experience at all then most of this lesson will be straightforward. Here we'll introduce some basic programming structures which will show up in our M-files.

Displaying Stuff - `disp`

We know that if we do a calculation the output is printed unless we suppress this by ending the calculation with a semicolon. However what if we wish to display something like some text or the variable without the `x =` which Matlab automatically inserts? Well there are many ways to do this but the `disp` command (display) is the easiest. The format is `disp(X)` where `X` is either a string or a vector of strings. Keep in mind though that numbers can be converted to strings with `num2str` and symbolic expressions to strings with `char`. Here are some examples. Make sure you parse them carefully to understand what they do!

```
disp('Hello World!')
```

```
        Hello World!
```

```
x=2;
disp(['The value of x is ',num2str(x)])
```

```
        The value of x is 2
```

```
syms t;
disp(['The derivative of t^2 is ',char(diff(t^2,'t'))])
```

```
        The derivative of t^2 is 2*t
```

Condition - `if`

An `if` statement can be used to execute a statement if a certain condition is met. The general simplest syntax is:

```
if expression
statement
end
```

We should pause to note here that Matlab is situation-aware. By that we mean that if you type the line

into Matlab and press Enter nothing will happen. Matlab is waiting for you to finish your if-statement-end structure. Matlab will do nothing until you type `end` and press Enter. So type the following exactly as you see it:

```
x=5;
if (x>3)
  disp(['x was bigger than 3 so here is twice it: ',num2str(2*x)])
end
```

```
x was bigger than 3 so here is twice it: 10
```

Here we see that since `x` is in fact greater than `3` the code between `if` and `end` is executed. On the other hand the following gives no output.

```
x=2;
if (x>3)
  disp(['This disp will not execute, how sad!'])
end
```

Here since the condition is false the code is not executed. If we want to test multiple conditions and if we wish to have a catch-all which occurs if none of the conditions are met we can do that with the general form as follows:

```
if expression1
statement1
elseif expression2
statement2
...
else
catchallstatement
end
```

What happens here is that first `expression1` is tested and if it's true then `statement1` is executed. If not then `expression2` is tested and if it's true then `statement2` is executed and so on. If none of them are true then `catchallstatement` is executed. As in the previous example Matlab will do nothing until you `end`.

IMPORTANT: If you need the `if` statement to be a compound statement you can use `&` for *and* and `|` for *or*. For example

```
if (x>0 & x<10)
```

will check *both* conditions whereas

```
if (x<3 | x>5)
```

will require only one to be true.

Loop - `for`

Supose we wanted to assign the variable `x` to each of the numbers 1 through 5 and then print 2 times each. We could do this:

```
x=1;
disp(num2str(2*x))
x=2;
disp(num2str(2*x))
x=3;
disp(num2str(2*x))
x=4;
disp(num2str(2*x))
x=5;
disp(num2str(2*x))
```

```
2
4
6
8
10
```

This is pretty tedious. If we needed to do something complicated with each of the `x` values it would be even worse and if we needed to assign `x` to each of the numbers 1 through 100 even worse still. The `for` loop allows us to do this in a much more straightfoward manner. Abstractly it works like this:

```
for x=(tell it which x values to do)
what to do with each x
end
```

The way we tell Matlab to go through all the `x` values is with a vector. So now the commands above with the variable `x` can be taken care of with either

```
for x=[1 2 3 4 5]
  disp(num2str(2*x))
end
```

```
2
4
6
8
10
```

or

```
for x=[1:5]
  disp(num2str(2*x))
end
```

```
2
4
6
8
10
```

Like with `if` nothing will happen until you `end` so go ahead and type this straight into Matlab as above. Here is another example with vector notation:

```
for x=[0:pi/6:pi/2]
  disp(['The sine of ',num2str(x),' is ',num2str(sin(x))])
end
```

```
The sine of 0 is 0
The sine of 0.5236 is 0.5
The sine of 1.0472 is 0.86603
The sine of 1.5708 is 1
```

The variable `x` runs from `0` to `pi/2` in steps of `pi/6` and `sin(x)` is taken at each step.

The general form is:

```
for v=vector
statement1
statement2
...
end
```

Here `v` is the vector which says which values `x` will run through and `statement1`, `statement2`,... are statements which will be executed.

Consider the following example which finds the derivative of `x^2` through `x^7`.

```
syms x;
for n=[2:7]
  diff(x^n,x)
end
```

```
ans =

2*x


ans =

3*x^2


ans =

4*x^3


ans =

5*x^4


ans =

6*x^5


ans =

7*x^6
```

Now then, `for` loops don't actually have to do anything with the looping variable, it can just be used as a counter. For example the following just says `Hi!` four times.

```
for n=[1:4]
  disp('Hi!')
end
```

```
        Hi!
        Hi!
        Hi!
        Hi!
```

Here is a `for` loop which does something more - it adjusts a value as it goes along. Imagine you wanted to start with the number `1.2` and square it five times. You can do this as follows. Notice how the code works. `mynum` is initially set to be `1.2` and is then adjusted within the `for` loop each time. Within the loop `mynum` is reassigned.

```
mynum=1.2;
for n=[1:5]
  mynum=mynum^2;
  disp(mynum)
end
```

```
           1.440000000000000

           2.073600000000000

           4.299816959999999

          18.488425889503635

             3.418218918716682e+02
```

If you just wish to print the final answer you can do:

```
mynum=1.2;
for n=[1:5]
  mynum=mynum^2;
end
disp(mynum)
```

```
             3.418218918716682e+02
```

More Examples of Loop - `for`

Here are some other examples:

Example 1: Adding the numbers 1 through 10.

How can we add the numbers 1 through 10 in Matlab? We first assign a total with the value 0 and then we successively add the numbers 1 through 10 to that total:

```
total=0;
for n=[1:10]
```

```
  total=total+n;
end
disp(total);
```

```
           55
```

Example 2: The function `f(x)=2.5x(1-x)` appears in certain population dynamics. The way it works is that if you plug in one years' population it outputs the next year's population. Here's a matlab loop which gives the next five years' populations if the initial population is 0.3 (think thousands):

```
f = @(x) 2.5.*x.*(1-x);
pop = 0.3;
disp(['Start at ',num2str(pop)]);
for n=[1:5]
  pop=f(pop);
  disp(['Next at ',num2str(pop)]);
end
```

```
        Start at 0.3
        Next at 0.525
        Next at 0.62344
        Next at 0.58691
        Next at 0.60612
        Next at 0.59685
```

Using `for` Loops to Store Answer in Vectors

Often we may do a calculation which requires us to store several different values. We can do this by storing those values in a vector. Mathematically a vector is just an ordered collection of numbers. We assign vectors using `[]` notation. Here is an example:

```
a = [1 -2 4 5]
```

```
        a =

             1    -2     4     5
```

Here `a` is a vector containing four numbers. If we want to access just one of those four we use `()` notation:

```
a(2)
```

```
        ans =

            -2
```

Or to change one of them we can assign this way. Notice that Matlab gives us the entire new vector back to see.

```
a(3)=17
```

```
a =

     1    -2    17     5
```

Suppose now we want to do a calcuation of something like `sin(2^1)`, `sin(2^2)`, ..., `sin(2^7)` and store them all in a vector. First we assign a vector with 7 entries and then we use a `for` loop to do the work:

```
v=[0 0 0 0 0 0 0];
for i=[1:7]
  v(i)=sin(2^i);
end
```

Then we can see it. Notice that because the vector stretched beyond the screen width matlab labeled the entries (`columns`) on each line.

```
v
```

```
v =

  Columns 1 through 3

   0.909297426825682  -0.756802495307928   0.989358246623382

  Columns 4 through 6

  -0.287903316665065   0.551426681241691   0.920026038196791

  Column 7

   0.721037710501732
```

Or just one entry

```
v(2)
```

```
ans =

  -0.756802495307928
```

Loop - `while`

The trouble with a `for` loop is that it executes a statement a certain fixed number of times. Often in mathematics we wish to execute a statement an unknown number of times until a criteria is met. For example when we do the bisection method to find a root we iterate until we're *close enough*, which could mean until our interval is some certain size. If we do a Riemann sum we might wish to increase the number of subintervals until our sum settles down, meaning it doesn't change by more than say 0.1 from iteration to iteration. In Matlab this is done most easily with a `while` loop. The general form of this loop is:

```
while (some conditions are met)
statement1
```

```
        statement2
        ...
        end
```

What happens in this case is that Matlab enters the while statement and tests the conditions. If they're true it proceeds through the statements. At the end it goes back to the beginning and tests again. If the statements (any of them) are ever false it ends the loop. Here is a simple example with its output:

```
x=3;
while (x<10)
  disp(['The value of x is ',num2str(x)]);
  x=x+1;
end
disp(['At the final exit the value of x is ',num2str(x)]);
```

```
        The value of x is 3
        The value of x is 4
        The value of x is 5
        The value of x is 6
        The value of x is 7
        The value of x is 8
        The value of x is 9
        At the final exit the value of x is 10
```

What this code does is starts by assigning the value `x=3`. The `while` statement checks the value, finds it's less than `10` and goes into the loop. It prints the text and value and then increments the value by `1` (this is what the `x=x+1` does). Then it goes back and tests `x<10` again. The final time the statements in the loop will be executed is when `x=9`. At the end of this iteration Matlab does `x=x+1` at which point `x=10` and then Matlab goes back to the `while`, sees it fails and drops out. Keep in mind that `x=10` upon exit but the code does not execute with `x=10`.

This next example is related to one of the earlier `for` loops. Again we start with the number `1.2` but now we repeatedly square the number until we get above `1000000` and then we stop. This happens pretty quickly! Notice what the `while` does - it checks if `mynum` is `<=1000000` and if so, it keeps going, squaring over and over. We've made it neater using `disp`.

```
mynum=1.2;
while (mynum <= 1000000)
  disp(['Now the number is ',num2str(mynum)])
  mynum=mynum^2;
end
disp(['Dropping out of the loop the number is ',num2str(mynum)])
```

```
        Now the number is 1.2
        Now the number is 1.44
        Now the number is 2.0736
        Now the number is 4.2998
        Now the number is 18.4884
        Now the number is 341.8219
        Now the number is 116842.2058
        Dropping out of the loop the number is 13652101047.4993
```

Here is a `while` loop which divides a number by `2` over and over until the result is less than or equal to `1`. Think " `while` the number is greater than `1`, keep dividing." Furthermore this code only prints the final result.

```
mynum=2753;
while (mynum > 1)
  mynum=mynum/2;
end
disp(mynum)
```

```
           0.672119140625000
```

Here's a more sophisticated example. It calculates successive partial sums `1/1+1/2+...+1/n` starting with `n=1` until successive sums are within `0.01` of each other. All the while it prints out a summary. Note the use of a new variable within the loop so that the old variable is not immediately lost. Also note that `oldvalue` is initially set to be large enough to get us into the loop.

```
oldvalue = 1/1;
newvalue = 1/1+1/2;
n = 2;
while (abs(newvalue-oldvalue) >= 0.1)
  disp(['Adding up to 1/',num2str(n),' yields ',num2str(newvalue)]);
  oldvalue = newvalue;
  n = n+1;
  newvalue = newvalue+1/n;
end
```

```
        Adding up to 1/2 yields 1.5
        Adding up to 1/3 yields 1.8333
        Adding up to 1/4 yields 2.0833
        Adding up to 1/5 yields 2.2833
        Adding up to 1/6 yields 2.45
        Adding up to 1/7 yields 2.5929
        Adding up to 1/8 yields 2.7179
        Adding up to 1/9 yields 2.829
        Adding up to 1/10 yields 2.929
```

Here's the same with `1/0!+1/1!+1/2!+...+1/n!` instead, the tolerance set to `0.000000000000001` and `format long` for more decimals. We've had to use `num2str(sum,20)` too which forces `num2str` to display up to 20 decimal digits rather than the usual four. If you remember something about series it may be familiar to you!

Hint: What does `1/0!+1/1!+1/2!+...` converge to?

```
format long;
tol = 0.0000000000000001;
oldvalue = 1/factorial(0);
newvalue = 1/factorial(0)+1/factorial(1);
n = 1;
while (abs(newvalue-oldvalue) >= tol)
  disp(['Adding up to 1/',num2str(n),'! yields ',num2str(newvalue,20)]);
  oldvalue = newvalue;
  n = n+1;
  newvalue = newvalue+1/factorial(n);
end
```

```
        Adding up to 1/1! yields 2
        Adding up to 1/2! yields 2.5
```

```
Adding up to 1/3! yields 2.6666666666666665186
Adding up to 1/4! yields 2.7083333333333330373
Adding up to 1/5! yields 2.716666666666666341
Adding up to 1/6! yields 2.718055555555555447
Adding up to 1/7! yields 2.7182539682539683668
Adding up to 1/8! yields 2.7182787698412700372
Adding up to 1/9! yields 2.7182815255731922477
Adding up to 1/10! yields 2.7182818011463845131
Adding up to 1/11! yields 2.7182818261984929009
Adding up to 1/12! yields 2.7182818282861687109
Adding up to 1/13! yields 2.7182818284467593628
Adding up to 1/14! yields 2.7182818284582301871
Adding up to 1/15! yields 2.7182818284589949087
Adding up to 1/16! yields 2.7182818284590428703
Adding up to 1/17! yields 2.7182818284590455349
```

The General Idea when Comparing Successive Values

It's extremely common to continue a calculation until some particular tolerance is reached. To help you see what the code should look like here's a code template. The idea for this template is that we're finding successive values of a variable for n=1, n=2, ... until they differ by less than tol.

```
Set oldvalue = the first value, corresponding to n=1.
Set newvalue = the second value, corresponding to n=2.
Set n = 2.
while (abs(oldvalue-newvalue)>=tol)
  oldvalue = newvalue;
  n=n+1;
  newvalue = find the new value for this new n.
end;
At this point, newvalue has the final calculated value.
```

Please make sure you see how this works. First we find the first two values, `oldvalue` and `newvalue` we're interested in. We set `n=2` because that's how far we've gone. We then start the loop, comparing these values (note we compare `while` the difference is too big!) Inside the loop we make `oldvalue` equal to `newvalue`, add `1` to `n` and then find a new `newvalue`. The `while` loop check then happens again and we continue.

In general the only things that need to be done to turn this into workable code is:

1. Set the oldvalue correctly for whatever you're finding.

2. Set the newvalue correctly for whatever you're finding.

3. Inside the loop, set the newvalue correctly for an unknown n.

Make sure to look at the two examples given earlier and see how they fit this template. The only additional thing they have is the `disp` statement.

Just to finish it off, here's one more example. It finds values of the integral from `1/x^2` from `1/n` to `2` as `n` gets larger and larger, starting at n=1 until successive values differ by less than `0.001`. The `disp` at the end just displays it.

```
syms x;
tol = 0.001;
oldvalue = int(x^2,1/1,2);
newvalue = int(x^2,1/2,2);
n=2;
while (abs(oldvalue-newvalue) >= tol)
  oldvalue = newvalue;
  n = n+1;
  newvalue = int(x^2,1/n,2);
end;
disp(double(newvalue));
```

```
           2.665694849368319
```

Closing

In closing we should mention that Matlab is a full-fledged programming language and there are many commands other than these. We recommend searching the Web and the help files to see what else can be done. We will explore other commands as we venture into M-files.

Published with MATLAB® 8.0