14:440:127– Introduction to Computers for Engineers

Notes for Lecture 06

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Loop Examples

1.1 Example- Sum Primes

Let's say we wanted to **sum all 1, 2, and 3 digit prime numbers**. To accomplish this, we could loop through all 1, 2, and 3 digit integers, testing if each is a prime number (using the *isprime* function). If and only if a particular value is prime, then we'll add it to our running total. Note that if a particular number is not prime, we don't do anything other than advancing to the following number.

```
total = 0;
for k = 1:999
    if(isprime(k))
        total = total + k;
    end
end
disp(total)
```

One interesting difference between Matlab and other programming languages is that it uses a vector to indicate what values a loop variable should take. Thus, if you simply write that x equals an arbitrary vector rather than a statement like *x = 1:100*, your program will work fine. Here, we rewrite the previous example for **summing all 1, 2, and 3 digit prime numbers** by first creating a vector of all the prime numbers from 1 to 999, and simply looping through those values:

```
total = 0;
for k = primes(999)
    total = total + k;
end
disp(total)
```

1.2 Example- Duplicate each element

Now, let's look at writing a loop to do something we can't already do in Matlab: duplicating each element of a vector. In other words, given the vector [1 4 3], we want to end up with [1 1 4 4 3 3]. Here are three of **many** ways we could write this:

```
V = 1:5;  % sample vector

%%% method 1
for j = 1:length(V)
    V2((2*j-1):(2*j)) = V(j);
end
disp(V2)

%%% method 2
for j = 1:(2*length(V))
    V2(j) = V(ceil(j/2));
end
disp(V2)

%%% method 3
counter = 0.5;
for j = 1:(2*length(V))
    V2(j) = V(ceil(counter));
    counter = counter + 0.5;
end
disp(V2)
```

1.3 Example- Converting A For Loop to a While

Essentially every for loop can be written as a while loop. This is a three step process:

- Notice that we need to initialize a loop variable (a *while loop* does not do this automatically). If our *for loop* began *for x = 1:2:15*, we must state that $x = 1$ initially, before our *while loop* begins.
- You must create a condition that is true while you want the loop to keep looping, and that becomes false when you want the loop to stop. Usually, this is the upper (or lower) bound on your loop variable. In our example, we'd want to keep looping while x is less than or equal to 15: $x <= 15$.
- Finally, before each iteration of our *while loop* ends, we must increment our loop variable. Notice that a *for loop* did that for us. Right before your *end* statement, you'll likely place something like $x = x+2$ if you were converting the above example.

Here's this simple example, written as both a *for loop* and an equivalent *while loop*.

```
for x = 1:2:15
    disp(x)
end
%%%%%%%%%%%%%%%%%%%%
x = 1;  % STEP 1
while(x<=15)   % STEP 2
     disp(x)
     x = x + 2;  % STEP 3
end
```

1.4 Example- Summing Integers

Of course, the power of a *while loop* isn't evident from just converting a *for loop*. Rather, a *while loop* should be used whenever you want to continue looping until some condition changes from true to false. For instance, let's say we wanted to **find the smallest number N for which the sum of the integers 1 through N is a 4 digit number**. In this case, it makes sense to use a *while loop*.

Essentially, we want to keep a running total of the sum, and keep adding successively larger integers to it as long as we haven't gotten a 4 digit number yet. Thus, while our running total is less than 1000, we should continue looping

(this is our condition). Once the total passes 1000, our loop will stop because the condition is false.

Our process could be as follows:

- Initialize our running total (*total*) as 0.
- Start at 1. Since this is a *while loop*, we need to make our own variable to do this. Let's call this variable n.
- While our total is still less than 1000, keep increasing n (our current number) and adding it on to the total.

We can thus write our loop as follows:

```
total = 0;
n=1;
while(total<1000)
     total = total + n;
     n = n + 1;
end
disp(n-1)
```

Note that we do something kind of funny in the last line. We display *n-1* as our answer, which is because once we've added a number to our total to push our total over 1000, we then increase n by 1 before checking the condition. Thus, we've gone one place too far!

There's an alternate way to write this loop that avoids that problem by switching the order of the statements in the loop, but then we have to start at 0 instead:

```
total = 0;
n=0;
while(total<1000)
     n = n+1;
     total = total + n;
end
disp(n)
```

We also could have written this example in a for loop:

```
total = 0;
for n = 1:inf
     total = total + n;
     if(total>1000)
          break;
     end
end
disp(n)
```

Note that it's often helpful to run through loops by hand to understand how they work. Try making columns for all of your variables, and run through the code one line at a time. Each time the value of a variable changes, mark this on your paper in the correct column.

1.5 Example- User Input

The *while* loop is very useful when getting input from the user. Let's say we wanted to, **over and over again, allow the user to input numbers as long as they're entering positive numbers. We want to sum all of these positive numbers that they've entered. However, once they enter a non-positive number, we want to stop the loop (and not include that number in the sum)**.

Our strategy for a *while loop* could be that while the number they input is greater than 0 (our condition), to add it to the running total and then ask for another number. Of course, before the loop starts, we'll need to ask

them to input the first number. Otherwise, when we check our condition for the first time, x won't exist:

```
total = 0;
x = input('Enter a number');
while(x>0)
     total = total + x;
     x = input('Enter a number')
end
disp(total)
```

1.6 Example- Twin Primes

In many cases, it's possible to use either a *while loop* or a *for loop* (often adding counting variables to *while loop* or adding *break* to a *for loop*) to solve a problem. Writing a loop that **finds all 4 digit twin primes** (numbers separated by two that are both prime) is such a case.

```
for x = 1001:2:9997
    if(isprime(x) & isprime(x+2))
        fprintf('%.0f and %.0f are both prime\n',x,x+2)
    end
end
```

```
x = 1001;
while(x<=9997)
    if(isprime(x) & isprime(x+2))
        fprintf('%.0f and %.0f are both prime\n',x,x+2)
    end
    x = x + 2;
end
```

Also note that you can solve this example without loops in Matlab:

```
x = 1001:2:9997;
twins = x(isprime(x) & isprime(x+2));
printout = [twins; twins+2];   % fprintf gets two elements from each column
fprintf('%.0f and %.0f are both prime\n',printout)
```

1.7 Example- Powers

In cases where a number changes, but not in an arithmetic sequence (i.e. **finding the smallest integer N for which 2^N is greater than one million)**, *while loops* are key, unless you can write the problem arithmetically:

```
count = 0;
total = 1;
while(total<=1000000)
     total = total * 2;
     count = count+1;
end
disp(count)
```

2 Nested Loops

You can also put loops (*for* or *while*) inside of each other, in what are called *nested loops*. These are very powerful, especially when working with matrices. Here's a first example of a nested loop so that you can see how the numbers change:

```
for x = 1:3
    for y = 1:2
       fprintf('x= %.0f and y= %.0f\n',x,y)
    end
end
```

This creates:

```
x= 1 and y= 1
x= 1 and y= 2
x= 2 and y= 1
x= 2 and y= 2
x= 3 and y= 1
x= 3 and y= 2
```

Note that the outer loop changes slowly, while the inner loop changes quickly.

2.1 Nested Loops- Convert a Matrix into a Vector

Having two variables, one changing more quickly than the other, is extremely useful when working with matrices. Let's say we wanted to create a vector V from a matrix M without using the colon operator. We could take the following approach:

- Determine how many rows and columns are in the matrix using *size*.
- Create an empty vector V.
- Start on column 1 of the matrix. Loop through each row, adding that element onto the end of the vector.
- Then, move to column 2, 3, 4,... and repeat the process.

```
% Assume matrix M exists
[r c] = size(M);
V = [ ];
for col = 1:c
    for row = 1:r
         V(end+1) = M(row,col);
    end
end
disp(V)
```

2.2 Nested Loops- Printing Out Stars

Let's say we wanted to print out the following pattern of stars, allowing the user to specify how many rows:

```
*
**
***
****
*****
******
*******
...
```

This situation lends itself perfectly to nested *for loops*. This is because we need to loop through one thing slowly (which row we're on), and then inside of that slow loop, repeat something else (if we're on row 1, print out 1 star; if we're on row 2, print out 2 stars; if we're on row 3, print out 3 stars).

Here's our plan:

- Ask the user how many rows they want.
- Loop through each row, beginning with 1. Keep a variable R, containing which row we're on.

- In each row, print out R stars. To do this, create a loop that prints out a single star, repeating this operation R times.
- After we've printed out R stars, go to the next line using `\n`.

```
rows = input('How many rows do you want?');
for R = 1:rows
    for s = 1:R
        fprintf('*');
    end
    fprintf('\n');
end
```

3 Loops- Efficiency Concerns

In Matlab, you'll often hear that it's bad to use loops when you could have instead used vector or matrix functions. This is because Matlab's built-in functions and arithmetic operations are optimized to use data stored as vectors or matrices in memory. Thus, whenever possible, use vectors, matrices, and their associated functions. Many of the examples I've given you in this lecture and the previous lecture could be written more efficiently using built-in functions and by "vectorizing" (replacing loops with vector operations). I've given you many examples using loops for two reasons. First of all, they're often some of the simplest loops to write, and thus they serve a pedagogical purpose. As you start writing more complicated programs, you'll often need to write loops, but these loops will be implementing much more complex algorithms. Second, most computer programming languages use loops much more extensively than Matlab, so being good at loops is a skill-set that you can port to other languages you learn. As with natural (human) languages, learning more programming languages becomes easier after your first one or two!

3.1 Timing Efficiency- Tic and Toc

Matlab has two convenient commands that let you time how long an operation takes to compute. To start (and reset) a timer, use the command *tic;* . To stop the timer and display the amount of time elapsed, use *toc;* . If you've brainstormed a few different methods for solving a problem and want to have an idea of which method runs most quickly in Matlab, use *tic* and *toc*!

3.2 Loops, Pre-allocated Memory Loops, Vector Operations

As an example of the huge differences in efficiency between methods in Matlab, let's compare three different methods of performing the same operation in Matlab. Particularly, let's create a 10,000 element vector filled with the number *5*. Indeed, 5 is against the law. (That's not actually true, that's just what happens when you type *"5 is"* into Google and look at the list of auto-complete possibilities.)

First, we can use the totally naive method and create this vector one element at a time:

```
% Allocate memory one element at a time
clear;clc
tic
for z = 1:10000
    x(z) = 5;
end
toc
```

This method takes **0.181933 seconds** on my desktop.

Now, let's pre-allocate memory for the vector we're creating. By this, we mean that we want to create a matrix full of zeros that's the eventual size we want. That way, Matlab will set aside space in memory for the entire vector, rather than having to mess around with allocating memory each time. If you're interested in this subject, I'd recommend you take more advanced classes in Computer Engineering or Computer Science such as Architecture, Operating Systems, and Compilers.

```
% Allocate memory in advance
clear;
tic
x = zeros(1,10000);
for z = 1:10000
    x(z) = 5;
end
toc
```

This method takes **0.000111 seconds** on my desktop.

Now, let's use the optimized, built-in functions (and arithmetic):

```
% Use built-in functions
clear;
tic
x = 5*ones(1,10000);
toc
```

This method takes **0.000059 seconds** on my desktop, or approximately half the amount of time of using a loop!

4 More Loop Examples

4.1 Example- Interest

Let's look at two different ways (using loops and not using loops) for calculating compound interest on a loan. In particular, we want to see, when given some amount of money (the principal) and an interest rate, how long it takes to have at least a million dollars if interest is compounded monthly. This is, of course, assuming that the bank in which you invest your money doesn't close, go bankrupt, need a bailout, or connect itself in any way with AIG.

Recall that when given the principal (p) and interest rate (r) as a decimal, the formula for the amount of money accumulated (A) after M months is

$$A = p * (1 + \frac{r}{12})^M \tag{1}$$

Let's look at a way of calculating this quantity using a while loop. *While* we don't yet have a million dollars, we'll keep looping and multiplying by $1 + r/12$, and keeping track of how many times we perform that multiplication.

```
p = input('Enter the principal');
r = input('Enter interest rate, as a decimal');
tic
months = 0;
currentp = p;
while(currentp < 1000000)
    months = months + 1;
    currentp = currentp * (1+r/12);
end
fprintf('It will take %.0f months. \n', months)
toc
```

Note that when p = 100 and r = 0.05, this calculation takes 0.007027 seconds on my computer.

Now, let's rewrite this example without using loops, of course noting that Equation 1 simplifies to:

$$m = \log_{1+\frac{r}{12}} \left(\frac{A}{p}\right) \tag{2}$$

$$m = \frac{\log\left(\frac{A}{p}\right)}{\log\left(1+\frac{r}{12}\right)} \tag{3}$$

```
p = input('Enter the principal');
r = input('Enter interest rate, as a decimal');
tic
months2 = ceil(log(1000000 / p) / log(1+r/12));
fprintf('It will take %.0f months. \n', months2)
toc
```

Don't forget to round up using *ceil* since it will take us to the end of a month to see our interest. Note that when p = 100 and r = 0.05, this calculation takes 0.000258 seconds on my computer... that's a pretty big improvement!

4.2 Fibonacci- Calculate First 100 Elements

Now, let's look at the Fibonacci Sequence, which is defined as $f_1 = 1$, $f_2 = 1$, and $f_n = f_{n-1} + f_{n-2}$.

As our first example, let's create a vector containing the first 100 elements of the Fibonacci Sequence. Since we know how many elements we want to calculate, we'll use a *for* loop. Also, let's use our trick of pre-allocating memory that we learned earlier. We'll first pre-allocate memory for a 100 element vector, and then set the first and second elements equal to 1. Then, starting at the third element, we'll set each element equal to the sum of the two previous elements.

We'll notice that:

- f(1) = 1;
- f(2) = 1;
- f(3) = f(2) + f(1);
- f(4) = f(3) + f(2);
- f(5) = f(4) + f(3);
- ...
- f(100) = f(99) + f(98);

See the pattern? Starting at the third element, *f(x) = f(x-1) + f(x-2)*. Let's write this as Matlab code:

```
f = zeros(1,100);
f(1) = 1;
f(2) = 1;
for j = 3:100
     f(j) = f(j-1) + f(j-2);
end
```

Now, we have a vector f with 100 elements, and these elements are the first 100 terms of the Fibonacci Sequence. If we thus wanted to know the 100'th element of the Fibonacci Sequence, or the sum of the first 100 elements, we could calculate those values trivially.

Now, let's look at the bounds of the loop, *j = 3:100*. Why 3 and why 100? Well, let's think about how we originally wrote this loop line by line. Our pattern was *f(x) = f(x-1) + f(x-2)*, and the first line fitting this pattern was *f(3) = f(2) + f(1)*. Similarly, the **last line** fitting this pattern would have been *f(100) = f(99) + f(98)*. Thus, in the first line of the pattern, x would be 3; in the last line of the pattern, x would be 100.

Of course, you might have seen the pattern as *f(x+2) = f(x+1) + f(x)*. The first line fitting this pattern would have remained *f(3) = f(2) + f(1)*, and the final line fitting this pattern would have remained *f(100) = f(99) + f(98)*. In this case, x would run from 1 (to create the first line) to 98 (to create the final line). Therefore, whenever you're deciding on the bounds to use in a loop, think of what the typical line's pattern will be, what the first line will be, and what the final line will be.

4.3 Example- Calculate the 100th Fibonacci Element

Now, let's say we only wanted to know the 100th element of the Fibonacci Sequence. It'd be quite a waste of our computer's resources to keep storing the previous 99 elements if we only cared about the 100th. Therefore, let's only keep track of the current element of the sequence (*the variable C*), the element one behind the current element (*the variable B*), and the element two behind the current element (*the variable A*). For instance, we can start by saying that A and B are both 1. Thus, $C = A + B$. Now, for our next trip through the loop, let's move B to A and C to B, in that order. This might be reminiscent of the lesson about switching variables from the first lecture. Let's repeat this process the necessary number of times, and we'll have our 100th element.

How many times should we repeat this? Well, the first time we execute our *for* loop, we'll be calculating the third element, and we want to go to the 100th element. Thus, we can loop from 3 to 100. Note that we never actually use the loop variable in our calculations; rather, we use it as a counter.

```
A = 1;
B = 1;
for j = 3:100
     C = A + B;
     %  move everything over one spot
     A = B;  % eliminate old A
     B = C;
end
disp(C)
```

Note that when calculating the 100th element, the first method took my computer 0.000819 seconds and our new A,B,C method took 0.000070 seconds, so the second method was about 10x as fast! Of course, note that these values are both heavily variable and heavily dependent on your system. A few times I ran these tests, the numbers came out almost identical, likely because I'm running many other things on my computer right now. Thus, take these sorts of estimates with a grain of salt.

4.4 Example- First 6 digit Fibonacci number

Now, what if I instead wanted to find the first element of the Fibonacci Sequence with at least 6 digits? In this case, we don't know how many times to repeat the process, so we use a *while* loop that repeats *while* our number is less than 6 digits.

```
A = 1;
B = 1;
C = A+B;
count = 3;  % we've just calculated element number 3
while(C<100000)
     %  move everything over one spot
     A = B;  % eliminate old A
     B = C;
     C = A + B;
     count = count+1;
end
fprintf('The first 6 digit Fibonacci number is %.0f, which is element %.0f \n',C,count)
```

Try writing this loop on your own. There are a lot of tricky details to worry about (getting the count correct, stopping at the right number, ordering the statements correctly inside the loop). Try and think about why I repeated the $C = A + B$ line before and inside the loop, why the count started at 3, why I had to manually increase the count, and all of the other subtleties of this example.

5 Debugging Loops- Part 1

One of the most difficult parts of computer programming is deciding what to do when your program doesn't work. You look at your code, and it seems right, but Matlab keeps giving you the wrong answer (or lots of error messages).

The error is called a "bug" in your program. The process of fixing the error is called "debugging."

In those cases, you need to figure out what's wrong, step by step. One of my favorite techniques to do this is to track the variables through each iteration of the loop by inserting a "dummy" *fprintf* statement inside the loop, as follows:

```
A = 1;
B = 1;
C = A+B;
count = 3;  % we've just calculated element number 3
while(C<100000)
     fprintf('Now A: %.0f, B: %.0f, C: %.0f, count: %.0f \n',A,B,C,count);  %HERE!!!
     %  move everything over one spot
     A = B;  % eliminate old A
     B = C;
     C = A + B;
     count = count+1;
end
fprintf('The first 6 digit Fibonacci number is %.0f, which is element %.0f \n',C,count)
```

The line that ends in the comment "HERE!!!" is our debugging line. Now, if our loop stops in an error, or if a value seems to be calculated incorrectly, we'll know the state of the variables when it happened so that we can diagnose our problem!