14:440:127– Introduction to Computers for Engineers

Notes for Lecture 05

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 (Pseudo)random Number Generation

In previous lectures, you've probably seen me create 'random' matrices whenever I want to demonstrate a concept on the fly. How do you create "random" numbers in Matlab? There's an ~~app~~ *rand()* function for that.

In short, Matlab lets you create matrices of pseudorandom numbers between 0 and 1. However, from this primitive, you can generate numbers in any interval using some arithmetic.

rand(m,n) creates an m x n matrix of 'random' values and *rand(s)* creates an s x s matrix of 'random' values. In technical terms, the values in this matrix are decimals uniformly distributed on the unit interval [0,1]. That means the probability of getting 0.1 more or less equals the probability of getting 0.2, and so on.

Of course, sometimes we'll want random values in an interval [0,b], where b is not 1. Well, let's say we have random numbers on the unit interval [0,1]. If we multiply those numbers by b, we'll effectively stretch the interval and now have random values randomly distributed on the interval [0,b]. Thus, *b*rand(m,n)* creates an m x n matrix of decimals on the interval [0,b].

What if you instead wanted an m x n matrix of values uniformly distributed on the interval [a,b]? Well, first notice that the size of this interval is (b-a), and thus we use the trick above. *(b-a)*rand(m,n)* creates a random value on the interval [0,b-a]. Now, add *a* to the number you created, shifting the interval to the right on the number line by *a*. Therefore, *r = a + (b-a).*rand(m,n)* returns an m x n matrix of decimals on the interval [a,b].

What if you instead wanted an m x n matrix of 'random' **integers** from 1 to x? Create an m x n matrix of random decimals on the interval [0,x] and apply the ceiling function: *r = ceil(x.*rand(m,n));*. Pay close attention to the fact that the original interval started at 0, but applying the *ceil* function rounds all of the numbers like 0.001 up to 1, so the integers range from 1 to x.

Similarly, to create 'random' **integers** from A to B, we notice that *r = ceil(x.*rand(m,n))* creates integers in the range [1,x], noting that the size of this range is (x-1). Thus, to create a range of integers of size (B-A), we do *r = ceil((B-A+1).*rand(m,n))*. Since we want to shift the beginning of this range from 1 to A, we shift by (A-1). Thus, *r = ceil((B-A+1).*rand(m,n))+(A-1)* generates an m x n matrix of random integers between A and B, inclusive.

1.1 PSEUDOrandom

Note that this section is for students who are interested; I won't test you on it.

So why have I been putting the word 'random' in quotes? Because these numbers aren't actually random. They're generated with a Pseudorandom Number Generator (PRNG), deterministically (which means you'll get the same result each time you run the process). *Pseudorandom* means that they 'look' or 'seem' random (i.e. they meet some statistical tests about the uniformity of their distribution, or perhaps it's difficult, when given all of the numbers generated so far, to guess the next number with more than chance probability, or perhaps the numbers are computationally indistinguishable from random numbers in a reasonable amount of time).

PRNGs begin with some 'seed' number, which tells the generator where to start. Of course, starting with the same seed always gives the same sequence of "pseudorandom" numbers. In Matlab, you can specify the seed (and the particular algorithm used) by typing *rand('seed',x)* (this uses a multiplicative congruential algorithm), or better, *rand('state',x)* (which uses a superior algorithm), for integers x.

How does a PRNG work? One of the more famous and more flawed (the patterns are easy to find) algorithms is a linear congruential generator. We'll use the notation where x_{n+1} represents the n+1'th number in the sequence, while x_n represents the x'th number in the sequence. $x_{n+1} = (a * x_n + c) mod n$, (recall that mod is *similar to the rem() function*. If you're interested in learning about better techniques, please feel free to come by office hours!

2 Loops

One of the biggest advantages of using a computer program to perform some task is that computer programs don't mind doing things over and over again. "Loops" are the concept in programming that allow you to repeat similar actions (or the same action) over and over. Matlab has two main kinds: *for* and *while* loops. Note, however, that if you can perform some task without using loops, for instance by using matrix functions, use the alternate way. In Matlab, built-in functions and matrix tasks are pre-compiled and optimized, which means they generally run faster than if you had written them as loops. We'll see this in action next week. However, sometimes, loops are unavoidable.

Loops implement *iterative* processes, which means processes that repeat over and over. Each repetition, or loop, through some task is called an *iteration*.

3 For Loops

Sometimes, you'll know exactly how many times you need to perform some action. Take the following example: **"Display HELLO 15 times."** This would be equivalent to writing:

```
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
disp('HELLO')
```

It would be pretty lame to type out (or even cut and paste) the same thing that many times. Instead, you can use what's called a *FOR LOOP*, which lets you repeat an action a specified number of times. A *for loop* takes the following form:

```
for VARIABLE_NAME = VECTOR_OF_VALUES
     STATEMENTS
     ....
end
```

For our first example, let's write the following:

```
for x = 1:15
     disp('HELLO')
end
```

In this example, the variable x will be our counter. Normally in Matlab, you'd think that the variable x now contains a vector of 15 elements. However, this is NOT the case since we're writing this statement as *for x = 1:15*. What a *for loop* does is first set the variable x to be 1, and 1 only. It will then execute the statement *disp('Hello')*, get to the *end* statement, and LOOP back up to the top of this loop. Now, x will be set equal to 2, and Matlab will repeat the loop. This means that it will execute *disp('Hello')*, get to the *end* statement, and LOOP back up to the

top of this loop yet again. This process will continue until x is set equal to 15, executes *disp('Hello')*, hits the *end* statement, and moves on with the program. Phew!

Now, what if we instead wanted to **print out the numbers 1 through 15 in the form "The number is X."** Notice that this is equivalent to writing:

```
fprintf('The number is %.0f\n',1)
fprintf('The number is %.0f\n',2)
fprintf('The number is %.0f\n',3)
...
fprintf('The number is %.0f\n',15)
```

When you're first starting to learn loops, I strongly advise you to first think about the process you want to implement and then try writing Matlab code that would implement the first few iterations (steps) of that process. Likely, these first few lines will be very similar. Identify what, if anything, changes from line to line, and replace that part with a variable.

In this case, the key is to notice that everything in the loop stays the same *except for* the last number. So, let's replace it by a variable, say X. Thus, we can write `fprintf('The number is %.0f\n',X)` . Now, we have to repeat this line, but change the value of X each time– and that's exactly what a *for loop* does! Thus, to write the above example as a loop, write:

```
for X = 1:15
     fprintf('The number is %.0f\n',X)
end
```

The key points here are to notice that the variable we define for the loop, X, takes on a single value at a time, starting at 1 and going to 15 (incrementing by 1 each time). For each of these different values of X, the *fprintf* statement is repeated. This *loop variable* is sometimes referred to as the *loop index*. The *for* loop must begin with the word *for*, and end with the word *end*.

In this next section, we'll go through a bunch of examples of *for loops*. Of course, note that these are just a few of the many, many types of loops you can write.

3.1 For Loop- Copying A Vector

For this example, we'll assume that there **exists a vector V1, and you want to make a copy of this vector and save it in V2**. We'll also assume that we forgot how to just type *V2 = V1*, so therefore we'd need to write this as a loop. First, let's come up with our algorithm (process) for doing this.

First, we'll look up the first element of V1, and save it as the first element of V2. Then, we'll look up the second element of V1, and save it as the second element of V2. We'll continue this until we copy the last element of V1 into the last element of V2. But wait– how do we know when we've reached the end? Well, before we begin this process, let's determine how many elements there are in V1 using the *length* function. If *length* returns 125, well, then we'll be repeating this process 125 times. Of course, we can just save the *length* as a variable. Ok, now let's try writing this process line by line:

```
howMany = length(V1);
V2(1) = V1(1);
V2(2) = V1(2);
V2(3) = V1(3);
V2(4) = V1(4);
...
V2(howMany-1) = V1(howMany-1);
V2(howMany) = V1(howMany);
```

Notice that each of these lines follows the pattern *V2(X) = V1(X)*, and we're doing this starting at X = 1, then X = 2, and continuing all the way until X = *howMany* (the variable storing the number of elements). Well, we've just defined our loop:

```
howMany = length(V1);
for X = 1:howMany   % OR for X = 1:length(V1)
     V2(X) = V1(X);
end
```

3.2 For Loop- Summing A Vector

For many loops, we'll want to keep a "running total" that sums a process. For instance, we could write Matlab code that works exactly the same as the *sum* function for a vector V. First, we'd need to determine how many elements are contained in a vector. Then, we'd want to start keeping track of our total, initially saying that it's 0 since we need to know from where to start counting. Each time we look at an element in the vector V, we'll add it on to our running total. Trying to write this process line by line, we could do:

```
% assume the vector V has already been defined
howMany = length(V);
total = 0;
total = total + V(1);
total = total + V(2);
total = total + V(3);
...
total = total + V(howMany-1);
total = total + V(howMany);
disp(total)
```

Hopefully, you see the pattern. Therefore, let's write our loop as:

```
% assume the vector V has already been defined
total = 0;
for k = 1:length(V)
    total = total + V(k);
end
disp(total)
```

Note that if you hadn't initialized total, Matlab wouldn't have known where to start counting. Even worse, *total* may have been defined earlier in the program, leading you to some pretty crazy and unexpected results.

3.3 For Loop- Counting

Now let's say that we instead wanted to **count the number of 3 digit prime numbers**. In this case, we'll keep a running count of how many primes we've seen.

Let's use the following process. Loop through each 3 digit number (100 to 999). For each of these numbers, check if the number is prime (using the *isprime* function). If it is indeed a prime number, add 1 to our count. If it's not, don't change our count. Of course, we need to start off by setting the count equal to 0. Here's how this loop would work:

```
% start off our count at 0
% go through each 3 digit number
%    if the current number is prime
%         increase our count by 1
%    end
% end
% display the final count
```

```
count = 0;
for x = 100:999
    if(isprime(x))
        count = count + 1;
    end
end
disp(count)
```

3.4 For Loop- Sum Primes Method 1

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, and test if each is a prime number (using the *isprime* function). If (and only if) a particular value is prime, then we'll add it on 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)
```

3.5 For Loop- Sum Primes Method 2

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)
```

3.6 For Loops- Finding The Maximum in a Vector

In the previous few examples, we've seen cases where we've kept a running total or running count as we've gone through our loop. However, sometimes, you'll instead want to keep a "running maximum," or something along those lines. For instance, let's say we wanted to **find the largest value in some vector V**.

Let's first define our process. At all times, we'll keep track of the "maximum so far," which we'll save in the variable *maxvalue*. We'll loop through each element of the vector, and for each of these elements, compare it to our *maxvalue so far*. If the number we're currently looking at is bigger than *maxvalue*, then that should replace *maxvalue* with the current element of V since that's the new largest number we've seen so far. If it's not bigger than *maxvalue*, then don't do anything. Thus, at the end of our loop, the variable *maxvalue* will contain the overall maximum value, since that will be the largest value we've seen so far, and we'll have seen every element of the vector.

There's one complication: when we try and compare the first element of the matrix to *maxvalue*, the variable *maxvalue* won't have a value yet and we'll thus get an error message. To fix this, let's initially set *maxvalue* to be *-inf* (negative infinity) since every value is bigger than negative infinity. Similarly, if we were trying to find the minimum of a vector, we'd want to set our initial value to *+inf*, since every value is smaller than positive infinity.

```
% assume the vector V has already been defined
% set maxvalue to be -inf
% loop through each element of V
%       if the current element is bigger than maxvalue
%               replace maxvalue with the current element
%       end
% end
% display the maximum value
```

```
% assume the vector V has already been defined
maxvalue = -inf;
for j = 1:length(V)
    if( V(j) > maxvalue )
         maxvalue = V(j);
    end
end
disp(maxvalue)
```

3.7 For Loop- Flip A Vector Left-Right

It's important to realize that sometimes you have to do tricky computations with the loop variable. For instance, what if you wanted to flip a vector *V1* horizontally (from left to right) and save it as *V2*... and the *fliplr* function did not exist! Well, for a 5 element vector, we'd want to write:

```
% assume V1 exists and has 5 elements
V2(5) = V1(1);
V2(4) = V1(2);
V2(3) = V1(3);
V2(2) = V1(4);
V2(1) = V1(5);
```

Notice that we usually find the pattern in the first few statements we write, and replace whatever is changing by a variable. Here, there are two things changing, and they are changing in opposite directions. Who you gonna call?

Well, we're actually lucky since the two parts of these lines that are changing are always changing by the same amount. Thus, if we change the right-hand side of these statements to be *V1(x)*, then we can write the left-hand side as *V2(6-x)*, which decreases at the same rate as the other side increases! Note that in general, this formula for determining the index of the left-hand side is *vectorLength + 1 - x*.

```
% assume the vector V1 exists and has an unknown number of elements
z = length(V1);
for x = 1:z
     V2(z+1-x) = V1(x);
end
```

3.8 For Loop- Making A Decision (isprime)

What if Matlab hadn't defined a function like *isprime*? This isn't unreasonable at all– many programming languages DON'T define a similar function. How could you test primality? Remember that a prime number is evenly divisible by only 1 and itself. Therefore, we can test whether some number X is evenly divisible by any number between 2 and X-1. If there does exist some number from 2 to X-1 that evenly divides the number, it's composite. If there does not exist such a "factor", then the number is prime. We'll keep track of whether or not the number is prime in a variable *IsItPrime*. At the end of our program, *IsItPrime* should be 1 (true) if the number is prime, and 0 (false) if it's not. (Of course, note that we only really have to test up to X/2 since there can't be any factors between X/2 and X-1).

A naive and TOTALLY INCORRECT approach to test whether some number X is prime would be to loop through

all numbers from 2 to X-1, and for each of these potential factors, to do the following:

- INCORRECT: If that potential factor evenly divides X, set IsItPrime to 0, or false
- INCORRECT: If that potential factor doesn't evenly divide X, set IsItPrime to 1, or true.

Notice that this approach crashes and burns since we just need to see if one number evenly divides X to determine if it's composite. In this naive approach, we may find a number that evenly divides X, and thus set *IsItPrime* to 0. Then, if the next potential factor we test does not evenly divide X, we'll reset *IsItPrime* to 1, which is not correct (we already know it's NOT prime). **Thus, let's assume it's prime at the beginning. As we continue through the loop, if we find any evidence to the contrary (i.e. a factor does evenly divide X), we'll permanently set IsItPrime to 0 since we've determined once and for all that X is not prime.**

```
x = input('Enter a number');
IsItPrime = 1;
for (potentialFactor = 2 : (x/2))
    if(rem(x,potentialFactor)==0)
        IsItPrime = 0;  % we've found a factor!
    end
end
disp(IsItPrime)
```

4 Break and Continue

In the previous example for determining whether or not a number X is prime, once we find a factor that's neither 1 nor X, we can stop searching since we now know that the number is composite. Testing any more potential factors is a waste of time! In cases like these, you should use a command called *break*, which "breaks" (or stops) the loop, but doesn't stop the program itself from running. To integrate *break* into the previous example, we could type:

```
x = input('Enter a number');
IsItPrime = 1;
for (potentialFactor = 2 : (x-1))
    if(rem(x,potentialFactor)==0)
        IsItPrime = 0;  % we've found a factor!
        break;   % stop the loop
    end
end
disp(IsItPrime)   % still displays even if break is used
```

Another command similar to *break* is *continue*. Whereas *break* stops the execution of a loop, *continue* stops the current iteration (the current pass through the loop) and jumps to the top of the loop to continue looping with the next value of the index. As a (kind of useless) example, the following loop will display 1,2,3, and 5, but not 4 (since it will jump to the top of the loop and move on to 5):

```
for x = 1:5
    if(x==4)
        continue;
    end
    disp(x)
end
```

5 While Loops

While a *for loop* lets you repeat some segment of code a specified number of times, sometimes you'll instead want to continue looping a section of code until some condition is met. In these situations, you use a *while loop*. A *while loop* repeats some segment of code until a particular condition is false. While the condition is true, Matlab will loop

through the code. Note that if the condition is initially false, a *while* loop will never loop through the code and will just continue on with the Matlab program.

Conversely, if the condition is always true, the loop will go on forever. This is called an "infinite loop." If you accidentally cause an infinite loop (or if Matlab is just taking too long to perform some operation), hit **Ctrl - C** on your keyboard (hit the two keys at the same time) to stop the current Matlab code!

Note that if the tested condition becomes false part way through an iteration (a pass through the loop), the particular iteration will finish and the loop will not stop until the condition is checked again when the loop returns to the top. Also note that, like *for* loops, the *while* loop requires that you type "end." Here's the syntax for a while loop:

```
while(CONDITION) % replace capitalized parts
     STATEMENTS  % replace capitalized parts
end
```

5.1 While Loop- Converting A For Loop

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
```

5.2 While Loop- 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.

5.3 While Loop- 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)
```

5.4 For / While Loop- 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)
```

5.5 For / While Loop- 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. Here's an example of each approach:

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

```
for x = 1:inf  % Note that this is not a clean way to solve this problem
     if(2^x>1000000)
          disp(x)
          break
     end
end
```

6 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.

6.1 Nested Loop- 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)
```

6.2 Nested Loop- 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
```