MATLAB Loops (iterative statements)

This is Chapter 4 of the textbook

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2011 ; distribution outside this group of students is NOT allowed.*

G1. The `while` statement (loop).

G1.1 The `while` statement. The `while` statement allows us to repeat a statement until a specified expression becomes `false`. In brief, if the `expression` is evaluated to `true` (be reminded that any non-zero value is considered `true`), then `Statement 1,` is executed, and as soon as control reaches the `end`, control is back to the while statement where `expression` is to be evaluated again, for the next iteration. Any time `expression` evaluates to `false` control is passed to the statement following the `end` without executing `Statement 1` then; the same applies if `expression` initially evaluates to `false`.

```
while expression      % File : sub7p2awhile.m
   Statement 1        % one way to write a while  loop
end

while (expression)    % another way (with parentheses)
   Statement 1
end
```

G1.2 Example. Say we want to print the numbers from 1 to 10. Two ways to do that are given below. Pay attention to the placement of `a=a+1`, not the parenthesization of the `while` line.

```
1  %  One way to write          FILE:   sub7p2bwhileex.m
2  a=0;
3  while a< 10
4    a=a+1;
5    fprintf('%d \n',a);
6  end
7
8  % Another way to write
9  a=1;
10 while (a<=10) ;
11   fprintf('%d \n',a);
12   a=a+1;
13 end
14 % fprintf('%d \n',a) ;
```

(c) Copyright A. Gerbessiotis. All rights reserved.

G1. A `while` statement example.

G1.2a An explanation of `a=a+1`. The statement `a=a+1` might look strange to most of you. It means that the value of a is incremented by one, and the new value replaces the previous (old) value. When we have an assignment operator such as =, first the right-hand size is evaluated and the result stored in a temporary variable The value of a on the right-hand side is retrieved and to this value a one is added. (This step usually involves storing the result into a temporary variable.) The new value (of the sum) is then stored back into a by replacing its previous contents.

```
                 Value of a    Value of Temporary Variable          Step of evaluation
a=100;               ?                     ?                          100;
                                         100                         =100;
                    100                  100                        a=100;
a= a+1;             100                  101                         =a+1;
                    101                  101                        a=a+1;

Step 1:   Evaluate the right hand side  'a+1' using the old value of a (i.e. 100)
Step 2:   Store the result (101) into a temporary variable.
Step 3:   Set variable a to hold the contents (value) of temporary variable.
          Value of a is now 101.
```

G1.2b Infinite Loops: Bad programming (mostly). The following fragment shows how an infinite loop can run forever doing nothing useful. Programmers should be able to detect and avoid writing such code fragments.

```
while (1)
  x=1;
end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

G1. A `while` statement example.

G1.3 Example (Fibonacci) numbers. A Fibonacci sequence is a sequence that is recursive, that is, its value at a point n is given as a function of the value of the sequence at smaller value points. Let $f(n)$ be the n-th Fibonacci number. We are given $f(1) = 1$ and $f(2) = 1$, and for $n > 2$ we have that $f(n) = f(n-1) + f(n-2)$. The values $f(1), f(2)$ are given to us; for $f(3)$ we use the definition of $f(n)$ to find $f(3)$ i.e. $f(3) = f(3-1) + f(3-2) = 1 + 1 = 2$.

We would like to write a program that finds and prints the n-th Fibonacci number $f(n)$, given $f(1), f(2)$; those "first" values are sometimes called `initial` values and sometimes are called `base-case` values.

In the program that we will write, we will use a row vector(matrix) f to store Fibonacci sequence values through a MATLAB array f. Lines 1,2 below initialize the first two elements of an f array with the first two Fibonacci values. Line 3 reads from the input index n for which the $f(n)$ needs to be found. We use (**in fact, misuse**) variable i to iterate over all indices that will be generated. Note that i is set to 2 and the first thing that happens within the while-loop is to increment i by one i.e. i is now 3; we know $f(1)$ and $f(2)$ already, but not $f(3), f(4), \ldots$. In Line 7 the next $f(.)$ is computed using the previous two values already available. Thus for $f(3)$ we need to add $f(2)$ and $f(1)$ that are available from lines 1-2. The new value is set to a new element of the array f in line 8. Note that array f is dynamically extended to accommodate the new element, one element at a time. MATLAB is intelligent enough to realize that you need one more element in row-vector f and handles it gracefully for you.

```
1  f(1)=1;                                % sub7p4fib.m
2  f(2)=1;
3  n=input('␣index␣of␣Fibonacci␣number␣=␣');
4  i=2;
5  while (i< n);
6   i=i+1;
7   c = f(i-1)+f(i-2);
8   f(i)=c;
9  end
10 fprintf('Fibonacci(%d)␣=␣%d␣\n',i,f(i));
```

(c) Copyright A. Gerbessiotis. All rights reserved.

G1.4 Dynamic arrays in MATLAB. A dynamic array is an array that can be extended at will and transparently to the user (programmer) to accommodate new elements. In Lines 1,2,8 array f extends in size to accommodate one more element, $f(i)$. With reference to the program `prog4fib.m` of the previous page, the value of i the first time it enters the while loops is incremented by one and set to 3 in line 6. Thus the first time line 8 is executed, i's value is already 3. When the assignment `f(i) =c` of line 8 is to be executed for the first time, an element c is to be stored as the third element of array f (i.e. $f(3)$). However through lines 1 and 2, a row vector of only two elements $f(1)$ and $f(2)$ has been defined. There seems to be no space to accommodate a third element. MATLAB however treats f not as a fixed size mathematical object that a matrix is, but as an extensible dynamically-sized array. Then MATLAB extends the 2-element vector by one more element, i.e. it does the following:

(a) allocates space for a new 3-element vector,

(b) copies the contents of $f(1)$ and $f(2)$ to the new 3-element vector,

(c) releases/clears the memory of the 2-element vector, and renames the new vector into f, and

(d) finally assigns to its third element the value computed in line 7.

Dynamic arrays IS A VERY BAD PRACTICE OF USING ARRAYS/VECTORS IN MATLAB.
Although dynamic arrays look useful and convenient, their reckless use is highly inefficient. Every time an element is created through a statement similar to that of line 8, MATLAB repeats steps (a),(b),(c), and (d). This is too time and also space consuming. For an array with few (say, 3) elements this might not make much difference, but if you have thousands or millions of elements, it will slow down the computation.

Note of caution. In the previous page we mentioned that we **misused** variable `i`. This is because i is a reserved word in MATLAB for the square root of `-1`. One also needs to avoid for the same reason using a variable named `j`. A better choice would have been `ii` or `jj` for that matter.

G1. Dynamic arrays in MATLAB .

The code below dynamically allocates elements of vector x. It is a very bad programming practice. Based on the previous discussion in order to allocate element $x(k)$ the program will spend in the order of k steps to allocate a new copy of x with k elements, and copy the old copy with $k-1$ elements to the new. Thus the number of steps in all iterations $1, \ldots, k$ will be proportional to $1 + 2 + \ldots + k$, a quadratic number (i.e. in the order of k^2). Thus for a loop that runs 10000 or so times this will cost $100,000,000$.

```
1 x(1)=1;
2 k=1;
3 while (k < 10000)
4    x(k)=k;
5    k=k+1;
6 end
```

There is an easier and more efficient way to do this that will cost you only $10,000$. Preallocate vector x. Of course this assumes that you know how much x will grow. This is being done in the fragment below.

```
1 x=zeros(10000,1);
2 x(1)=1;
3 k=1;
4 while (k < 10000)
5    x(k)=k;
6    k=k+1;
7 end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
1 ii=1;                                          % sub7p6whileex.m
2 sum=0;
3 n=input('enter n= ');
4 while (ii <=n)
5   sum = sum +1;
6    ii = ii *2;
7 end
8 fprintf('n= %d sum=%d ii = %d\n',n,sum,ii);
```

In this artificial program, the index variable `ii` of the `while` statement expression is being changed within the loop through line 6. It is not very clear how many times the `while` loop will be executed unless we know how large n will be. It is interesting to note that the value of `sum` is the number of times the `body` of the while-loop is executed (i.e. lines 5 and 6) and it is also worth-noting that the value of sum is equal to the number of bits in the binary representation of integral value n. (We say integral value rather than integer n, if only because MATLAB stores n as a `double` floating-point number.)

(c) Copyright A. Gerbessiotis. All rights reserved.

G2. The `for` statement (loop)

G2.1 `for` statement. The MATLAB `for` statement executes a sequence of statements (the body of the loop) a specified number of times. The structure of a `for` loop is shown below.

```
for index = expr                % FILE: sub7p7for.m
   Statement 1
end
```

Variable `index` is the loop variable (index) and `expr` is the loop control expression whose result is an array. The columns of the array generated by `expr` are assigned one by one to `index` and then the body of the loop, i.e. `Statement 1` is executed once per column of the array generated by `expr`. One natural form for `expr` is the shortcut array notation `first:increment:last`.

```
1. expr is evaluated and an array is generated.
2. It is checked whether there is a column to be considered in the generated array
   (i.e whether array is empty, or it has one column that was not used in previous
        iterations)
  2a. If the array is empty or there is no additional column to be considered  go to line 3 below.
  2b. Otherwise, there is a new column to be considered, and  the next column of the array generated
      is assigned to index and the lines of Statement 1 are then executed all the way before the
      end statement
  2c. Go to line 2 above (again).
3. Continue with the execution past the end statement of the for loop.
```

G2.2 Example. For Case 1, the array generated by `1:1:0` is empty, and thus the body of the loop is never executed. For Case 2, the array generated has only one column with value `1` and thus an `1` will be printed by way of `disp(index)`.

```
%  Case 1                       % Case 2
for index = 1:1:0               for index=1:1:1
  disp(index)                     disp(index)
end                             end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

G2.3 Example (Fibonacci) numbers. We rewrite the previous Fibonacci sequence `while`-based implementation to work with a for loop. In Line 1 we accept from the keyboard the index n for which $f(n)$ is to be computed. Based on this, an array f of n elements is generated in one step rather than dynamically one element at a time and all its elements are set to zero. Its first two elements are then set, i.e $f(1)$ and $f(2)$ are set to 1 in line 3. Then, an array [3 4 5 6 ... $n-1$ n] is generated through the statement `3:1:n`, i.e. the indexes (or indices) of the Fibonacci numbers that will be computed in line 5. f(index) is then computed in turn in Line 5 for all generated indexes. The final result, the n-th Fibonacci number, is printed through line 7. Note that because we avoid the dynamic reallocation of f, our code is more efficient than the previous `while`-based implementation.

```
1 n=input('␣Index␣of␣Fibonacci␣number␣=␣');          % FILE: sub7p8fibfor.m
2 f=zeros(1,n);
3 f(1)=1; f(2)=1;
4 for index = 3:1:n ;
5  f(index)= f(index-1)+f(index-2);
6 end
7  fprintf('Fibonacci(%d)␣=␣%d␣\n',n,f(n));
```

G2.4 Food for thought. Can you find the n-th Fibonacci number, without using an n-element f array? Explain.

G2.5 Example : Date into Numeric conversion. Assume for the sake of this example that February has 28 days. Given an input consisting of two numbers, one for `Month` stored in `m` and one for `Date` stored in `d`, where `Month` is between 1 and 12 and `Date` between 1 and 31 inclusive, and the latter range depends on the value of `Month`, write a MATLAB program that determines the index of the date within a Year. For example for input `1 1` the index is 1, because `1 1` means January 1st. For `1 31` it is `31` since this is January 31st, and similarly `3 2` is `61`, because this is March 2nd (and January has 31 days and February 28). Your code should be able to identify as incorrect input a pair `2 29` or say `2 0` or `3 32` and it must print an error message; however it should be able to recover and continue by resetting the date to the last day of the month given i.e. `2 28`, `2 28` and `3 31` respectively for the three cases given, notify the caller of the function at the end that a replacement pair has been used, and computer the correct index, i.e. `59,59,90` respectively. The two input values can be read by

```
m = input(' month = ');
d = input(' date  = ');
```

G3. The `break` and `continue` statements in loops

G3.1 Premature permanent exit with a `break`. When a `break` statement is encountered in the body of a loop (`while` or `for`), the execution of the loop immediately terminates and control is passed to the first statement after the `end` of the loop.

G3.2 Exit from current iteration with a `continue`. When a `continue` is encountered in the body of a loop, the execution of the statement immediately terminates and control is passed to the `end` of the loop, and the next iteration of the loop is prepared by going back to the top of the loop (i.e. the `for` or `while` line).

G3.3 Example. Consider the two examples below that use a `break` and a `continue` within a for loop. The code is supposed to add the integers from 1 to 10 (inclusive) giving a sum equal to 55. In the `break` variant execution terminates at $k = 5$, i.e. sum only computes `sum = 1 + 2 + 3 + 4`; in the `continue` variant Line 6 is not executed only for $k = 5$. In the former case we get a `sum` equal to 10 when Line 9 is executed, and in the other, `sum` equals to `50`.

```
1 sum=0;                          % FILE: sub7p9break.m
2 for k= 1:1:10
3     if (k==5) ;
4           break;
5     else
6        sum = sum+k;
7     end
8 end
9 fprintf('␣%d␣\n',sum);
```

```
1 sum=0;                          % FILE : sub7p9continue.m
2 for k= 1:1:10
3     if (k==5) ;
4           continue;
5     else
6        sum = sum+k;
7     end
8 end
9 fprintf('␣%d␣\n',sum);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

G4.1 Nested Loops. What does the following program fragment `Nest1` print for `sum`? What does `Nest2` do? In either case, when we have two nested loops, for every iteration of the outer loop, the inner loop is fully iterated. The programming language, i.e. MATLAB, knows that the `end` of line 6 is that of the previous `for` of line 4 and not that of line 3. Similarly the line 7 `end` matches the line 3 `for`. Thus for `ii=1`, the inner loop of lines 4-6 runs for $jj = 1\ 2\ 3\ \ldots\ 10$, before `ii` is assigned its next value 2. This is clear if one runs `Nest2` in which the values of `ii,jj` are printed for every iteration. Thus line 5 or line 14 is executed a total of `10*10` times since the loop of lines 4-6 is executed 10 times and the loop of line 5 is executed 10 times for each execution of the outermost loop. For nested for loops use different index variables (e.g `ii,jj`). Otherwise, an inner loop might change the index variable of an outer loop, with the end result being unpredictable. A `break` or `continue` statement will affect the innermost loop in which they (`break`/`continue`) reside.

```
1  % Code Nest1                    FILE: sub7p10nested.m
2  sum =0;
3  for ii= 1:1:10
4    for jj= 1:1:10
5      sum = sum + 1;
6    end
7  end
8  disp(sum);
9
10 % Code Nest2
11 sum =0;
12 for ii= 1:1:10
13   for jj= 1:1:10
14     fprintf('ii= %3d jj= %3d \n',ii,jj);
15   end
16 end
```

G5.1 Logical operators operating on vectors/matrices. If a is a matrix and you somehow perform operation $b = a > 5$, the result might be expectedly unexpected. In $b = a > 5$ the operator $>$ is a logical operator, the left operand a is a matrix (in the example below a is a 3×3 matrix) and the right operand 5 is a scalar. Yet this mismatch of operators, and operands generates a result, in the form of a matrix of dimension those of a whose elements are `true` or `false` (in fact their corresponding values of `1,0`) depending on whether the corresponding element of a is greater than 5 or not. When we perform `a(b) = a(b) .* a(b)`, the behavior is more unexpected. MATLAB leaves untouched those elements of a for which the corresponding b is 0; those elements for which the corresponding b is 1 get multiplied to each other. The net effect if the first five elements of a (in row-major order) are unchanged, and the remaining four get squared. The effect can be reversed by taking `a(b) = sqrt(a(b))` square-roots.

```
  >> a = [ 1 2 3; 4  5 6 ; 7 8 9]
  >> b = a > 5
b =
     0     0     0
     0     0     1
     1     1     1
  >> a(b) = a(b).* a(b)
a =
     1     2     3
     4     5    36
    49    64    81
  >> a(b) = sqrt(a(b))
a =
     1     2     3
     4     5     6
     7     8     9
  >> a(b) = sqrt(a(b))
a =
    1.0000    2.0000    3.0000
    4.0000    5.0000    2.4495
    2.6458    2.8284    3.0000
  >> a(b) = a(b).* a(b)
a =
    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    8.0000    9.0000
```

(c) Copyright A. Gerbessiotis. All rights reserved.

G5.2 Array programming. The same effect as `a(b) = sqrt(a(b))` could have been realized by writing a small program instead. In the program below `ii` iterates overall all rows of a and `jj` overall columns of a. `size(a,1)` retrieves the number of rows and `size(a,2)` the number of columns of a. The if condition $a(ii, jj) > 5$ retrieves only those elements greater than 5 and their values are changed and replaced by the corresponding square root. Note that the innermost `end` is that of the `if` statement.

```
% Matrix a is set at the end of the previous page
>> for ii=1:1:size(a,1)
    for jj=1:1:size(a,2)
      if (a(ii,jj) > 5)
           a(ii,jj) = sqrt(a(ii,jj));
      end
    end
   end
>> a
a =
    1.0000    2.0000    3.0000
    4.0000    5.0000    2.4495
    2.6458    2.8284    3.0000
```

(c) Copyright A. Gerbessiotis. All rights reserved.

For a matrix originally defined as follows, we would like to change all elements x that are greater than 10 to $x/10$. We can accomplish this in three different ways based on what we have learned so far.

```
a=[1   20  3; 40  5  60; 7  80  9];
a
```

Method 1. The first method involves two nested for loops that inspect every element in turn and change them accordingly.

```
for ii=1:size(a,1)
 for jj=1:size(a,2)
    if a(ii,jj) > 10 a(ii,jj)=a(ii,jj)/10;
    end
 end
end
```

Method 2. The second method involves only array (or matrix operations). First we use the relational operator $>$ as an array operator to determine the elements of a greater than 10. Matrix b would be 1 in those positions (i, j) for which $a(i, j) > 10$ and 0 otherwise. What we want to then then is set up b so that instead of holding 1s in those positions to hold 10, and 1 everywhere else. This is accomplished in two steps. First `b = b .* 9` generates 9s in place of the 1s, and then `b+1` adds 1 to every element of b, to obtain the desired result. The next step involves an array operation: dividing a by b element-wise.

```
b= a > 10;
b= b .* 9;
b= b + 1;
b
a= a ./b
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Method 3. The steps of Method 2 can be done in a one-liner.

```
a = a ./ ((9 .* (a>10))+1) ;
```

Method 4. Finally, the approach of section G5 can work with logical operators.

```
b=a>10;
a(b)=a(b) ./ 10;
```

(c) Copyright A. Gerbessiotis. All rights reserved.