ENGR 121: Computation Lab I

Handout for Lab 6: Loop Statements and Vectorizing Code

Practice Exercises

1. The arithmetic mean μ is calculated by summing all of the elements of a given vector and dividing the answer by the number of elements. Generate a vector x of 10 random numbers picked from a uniform distribution. Use a `for` loop to calculate the sum of these numbers and once you have the sum, divide it by the total number of elements in the vector to find the arithmetic mean. Do not use the built-in **sum** and **mean** functions in MATLAB. Now, the standard deviation σ is a measure of how spread out the data is from the mean value. The higher this value, the more the variation or spread in the data, and vice versa. Implement the following formula, also using a `for` loop, to obtain the standard deviation of the generated data:

$$\sigma = \sqrt{\frac{1}{N-1} * \sum_{i=1}^{N} \big(x(i) - \mu\big)^2}$$

Here μ denotes the arithmetic mean and N denotes the number of elements in your vector x. Once you have the answer, check for correctness: find the standard deviation of the vector using MATLAB's in-built function **std** and check to see if the answers match.

2. Create a vector x of 10 random integers, each in the inclusive range from -10 to 10. Perform each of the following operations using `for` loops along with `if` statements if necessary:

- Subtract 5 from each element in x.
- Count how many elements are positive (greater than zero) in x.
- Get the absolute value of each element in x.
- Find the maximum and minimum value in x.

Do not use in-built MATLAB functions.

3. Create an x vector with integers 1 through 10 and set a y vector equal to x. Plot this straight line. Now, add noise to each of the data points in y by creating a new $y2$ vector that stores the values of $y(i) \pm 0.25$, where $y(i)$ denotes the i^{th} element of the vector y. Plot these noisy points in a new figure. Label the axes and title the graphs appropriately.

4. Write a script called `beautyofmath` that produces the following output. The script should iterate from 1 to 9 to produce the expressions on the left, perform the specified operation to get the results shown on the right, and print exactly in the format shown below.

```
>> beautyofmath
1 x 8 + 1 = 9
12 x 8 + 2 = 98
```

```
123 x 8 + 3 = 987
1234 x 8 + 4 = 9876
12345 x 8 + 5 = 98765
123456 x 8 + 6 = 987654
1234567 x 8 + 7 = 9876543
12345678 x 8 + 8 = 98765432
123456789 x 8 + 9 = 987654321
```

5. Write a function called `prodby2` that will receive a value of a positive integer n and will calculate and return the product of the odd integers from 1 to n. Use a `for` loop.

6. This problem illustrates the use of nested `for` loops to operate on elements within a matrix. The following code shows how to use nested loops to access every element of a given matrix. The outer loop iterates over each row and the inner loop visits each column element along a given row.

```
%% Accessing elements in a matrix via nested for loops

mymat = rand(5, 5); % Create a 5 x 5 matrix
[rows cols] = size(mymat); % Obtain the number of rows and columns in ...
    the matrix

for i = 1:rows % Iterate over rows
    for j = 1:cols % Iterate over columns
        fprintf('Element (%d, %d) of the matrix is %.4f. \n', i, j, ...
            mymat(i, j));
    end
end
```

Using the above code snippet as a starting point, create a 3 x 5 matrix and perform each of the following operations using `for` loops with `if` statements if necessary:

- Find the maximum value in each column of the matrix.
- Find the maximum value in each row of the matrix.
- Find the maximum value in the entire matrix.

7. Repeat the above problem using `while` statements.

8. Let us now see how to use nested `for` loops to perform matrix addition and multiplication. Matrix addition is quite straightforward as shown by the function below: if $\mathbf{A}$ and $\mathbf{B}$ are the input matrices, and $\mathbf{C}$ is the desired result matrix, then $C(i,j) = A(i,j) + B(i,j), \forall i, j$.

```
function result = matrixadd(A, B)
% We assume square matrices

[rows cols] = size(A); % Obtain the dimensions of the input matrix
C = zeros(rows, cols); % Allocate memory for the output matrix

for i = 1:rows % Iterate over rows
    for j = 1:cols %Iterate over column elements
        C(i, j) = A(i, j) + B(i, j); % Obtain the (i, j) element of the ...
           result matrix
    end
end

result = C; % Return the result matix
end
```

```
>> A = [2 3 4; 5 6 7; 2 1 4];
>> B = [1 2 5; 6 4 5; 9 8 7];
>> C = matrixadd(A, B)
C =
     3     5     9
    11    10    12
    11     9    11
```

Now, write your own function `matrixmult` to perform matrix multiplication. The function should accept two $n \times n$ square matrices $\mathbf{A}$ and $\mathbf{B}$ as input arguments, and return the resulting $n \times n$ matrix $\mathbf{C} = \mathbf{AB}$. Check your answer using MATLAB's in-built matrix multiplication operation. Every element in the resulting matrix $\mathbf{C}$ is obtained as

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j).$$

So, three nested `for` loops are required in your code.

9. Trace this to figure out what the result will be, and then type it into MATLAB to verify the results.

```
count = 0;
number = 8;
while number > 3
    number = number - 2;
    fprintf('Number is %d \n', number);
    count = count + 1;
end
fprintf('Count is %d \n', count);
```

10. Given the following loop:

```
while x < 10
action
end
```

- For what values of the variable x would the action of the loop be skipped entirely?
- If the variable x is initialized to have the value 5 before entering the loop, what would the action have to include in order for this not to result in an infinite loop?

11. Sales (in millions) from two different divisions of a company for the four quarters of 2013 are stored in vector variables as follows:

```
div1 = [4.2 3.8 3.7 3.8];
div2 = [2.5 2.7 3.1 3.3];
```

Using subplot, show side-by-side the sales figures for the two divisions. What kind of graph shows this in the best way? Why? In one graph, compare the two divisions. What kind of graph shows this in the best way? Why?

12. The following problems deal with vectoring code. Recall that MATLAB is optimized to perform vector, array, and matrix operations. So, if it is not necessary to use loops in MATLAB, don't.

- The following code was written by somebody who does not know how to use MATLAB efficiently. Rewrite this as a single statement that will accomplish exactly the same thing for a matrix variable *mat*, that is vectorize this code:

```
[r c] = size(mat);
for i = 1:r
    for j = 1:c
        mat(i,j) = mat(i,j) * 2;
    end
end
```

- Vectorize the following code. Write one assignment statement that will accomplish exactly

the same thing as the given code, assuming that the variable `vec` has been initialized prior to executing this code snippet:

```
newv = zeros(size(vec));
myprod = 1;
for i = 1:length(vec)
   myprod = myprod * vec(i);
   newv(i) = myprod;
end
newv % Note: this is just to display the value
```

13. MATLAB has built-in functions that determine how long it takes code to execute. One set of related functions is `tic/toc`. Use these functions to time the `matrixmult` function that you developed earlier and compare its performance to MATLAB's built-in matrix multiplication operation. Your script may look as follows:

```
%% Use of the tic/toc functions to time portions of the code
% Author: Naga Kandasamy
% Date created: 11/02/2014

clc;
clear;

rows = 100;
cols = rows;
A = rand(rows, cols); % Generate a 100 x 100 A matrix
B = rand(rows, cols); % Generate a 100 x 100 B matrix

% Time our implementation of matrix multiplication
fprintf('\nExecuting our implementation of matrix multiplication...\n');
tic
C = matrixmult(A, B);
toc

% Time MATLAB's multiplication operation
fprintf('\nExecuting MATLAB multiplication operation...\n');
tic
reference = A*B;
toc
```

14. Finally, let us focus on problem solving at its purest: solving puzzles.[1] Here, focus on solving the problem and not on the programming syntax. Once you figure out what you want to do, translating your thoughts into MATLAB code will be straightforward. Let us try a series of programs that produce patterned output in a regular shape. Programs like these develop loop-writing skills.

We will work through one such pattern in some detail. For example, let us write a program that uses only two output statements, `fprintf('#')` and `fprintf('\n')`, to produce a pattern

[1] Material derived from: V. Anton Spraul, *Think Like a Programmer,* No Starch Press, 2012.

of # symbols shaped like half of a perfect 5×5 square (or a right triangle):

```
#
##
###
####
#####
```

You may already see the solution in your head. If you don't, a good starting point is to reduce the problem to something easier; for example, a whole square pattern, which is easy to achieve using nested **for** loops:

```
1  %% Generate a single line of five hash symbols ##### using a for loop
2  for i = 1:5
3      fprintf('#');
4  end
5  fprintf('\n'); fprintf('\n');
6  %% Generate a full 5 x 5 square shape using a nested for loop
7  for i = 1:5
8      for j = 1:5
9          fprintf('#');
10     end % End j loop
11     fprintf('\n');
12 end % End i loop
13 fprintf('\n');
```

```
#####

#####
#####
#####
#####
#####
```

Looking at the desired pattern, we see that we need a single # in the first row, two #s in the second row, three #s in the third row, and so on. It is clear that we want an expression that increases as the row number increases, or as the loop index i increases. We can achieve this by reworking the j loop in line 10 to be an algebraic expression of the outer loop index i as follows.

```
1  %% Produce the left half (or a right triangle) of a perfect 5 x 5 ...
       square of #s.
2  % Generate:
3  % #
4  % ##
5  % ###
6  % ####
7  % #####
8
9  for i = 1:5
```

```
    for j = 1:i
        fprintf('#');
    end
    fprintf('\n');
end
fprintf('\n');
```

As another example, consider generating the pattern:

```
#####
####
###
##
#
```

Again by looking at the pattern, we need an expression that generates five #s in the first row, four #s in the second row, and so on. So, we want an expression that decreases the number of #s generated as the row number increases. So, count down the outer loop and make the inner loop dependent on the outer loop index.

```
%% Produce the right half (or a right triangle) of a perfect5 x 5 ...
    square of #s.
% Generate:
% #####
% ####
% ###
% ##
% #

for i = 5:-1:1
    for j = 1:i
        fprintf('#');
    end
    fprintf('\n');
end
fprintf('\n');
```

From the above examples, we know how to:

- Display a row of symbols of a particular length using a loop.
- Display a series of rows using nested loops.
- Create a varying number of symbols in each row using an algebraic expression instead of a fixed value for the loop bound.
- Discover the correct algebraic expression through some experimentation and analysis.

Apply the above skills to generate the following patterns:

- *A Sideways Triangle*. Write a MATLAB program that uses only two output statements, `fprintf('#')` and `fprintf('\n')`, to produce a pattern of hash symbols shaped like

a sideways triangle. (Hint: Think of how to exploit the symmetry in the pattern that is evident above and below row four.)

```
#
##
###
####
###
##
#
```

- Write a program that produces the following shapes:

```
########
 ######
  ####
   ##


   ##
  ####
 ######
########
########
 ######
  ####
   ##
```