ELEC 342 Lab 3: Functions in MATLAB and the Sampling Theorem.

This lab will explore the use of MATLAB functions and investigate the effect of sampling rate on the Fourier transform of a signal.

User-defined functions in Matlab:

MATLAB allows for user-defined functions to be defined and to be invoked from within another M-file script. The following example shows how this is done:

Example 1: A function which returns the incremented value of its passed argument.

```
function inc_x = inc(x)
% inc(x) increments the value of x by 1


x = x + 1 ;  % since we are modifying the argument
             % within the function, it will be passed
             % by value and only a copy of the argument
             % will be modified

disp('The value of parameter x from within inc is : ');
disp(x);
% return the incremented value
inc_x = x ;
```

We define a MATLAB function by writing a M-file containing its definition. For example, the above code is saved in a file called `inc.m`. By convention, the name of the M-file is usually the same as the name of the function (`inc` in this particular example). The word `function` in MATLAB is a reserved keyword and is used to define a function. Let us examine the first line of this function in detail:

`function` **`inc_x`** = *`inc`*`(x)`

`function`: A reserved keyword in MATLAB used to define a function.

inc_x: The return value of the function. The function will return the last value assigned to this variable.

`inc`: The name of the function, usually the same name as the M-file containing the definition of the function, but it may be any syntactically correct name in MATLAB.

`(x)`: The argument(s) passed to the function. In this case there is only one argument called x. If a function has more than one argument, each would be separated by a comma.

The comment line immediately after the name of the function is referred to as the H1 line and is used to describe the behavior of the function. The H1 line and any subsequent comment lines up to the first non-comment line constitute the help section of the function and will be displayed when one types `help inc` from the MATLAB prompt[1]:

```
>> help inc
  inc(x) increments the value of x by 1
```

Example 2: A function may be invoked by simply typing its name (and passing an argument to it):

```
>> x = 5

x =

     5

>> ans = inc(5)
The value of parameter x from within x is :
     6


ans =

     6
```

The same may be performed from within a script file:

```
% Example script file which calls the inc function
% defined in the inc.m file

clear
x = 10;
disp('The value of this x is before calling inc is: ')
disp(x)
ans = inc(x) ;
disp('The value of this x after calling inc is:  ')
disp(x)
```

The output produced by this script is:

```
>> example2
The value of this x is before calling inc is:
    10

The value of parameter x from within inc is :
    11

The value of this x after calling inc is:
    10
```

This example illustrates what is known as the "call-by-value" parameter passing mechanism which MATLAB uses when the passed argument is modified by a function. The function receives a copy of the actual argument x and during the course of execution of the function, it modifies only the copy of the function. The argument x inside the function is said to be a local argument and furthermore the lifetime of any local arguments (and variables declared local to a function) is limited to the runtime of the function. Examine your workspace after running the script given in Example 2 and you will see there are only two variables: `x` with value of 10 and `ans` with value of 11.

Example 3: Pass-by-reference.

Consider the following function (defined in a file called array_square.m):

```
function array_squared  =  array_square(x)
% array_square(x) - returns an array containing the square of
% the elements
% in the  passed array x

array_squared = x .^ 2 ;
```

The function receives an array x as an argument, and it returns an array containing the square of the elements in the passed array. Is this a call-by-value? When a function only *reads* the value of an argument (and does not modify it), MATLAB employs what is known as a "pass-by-reference" parameter passing mechanism. Instead of copying the argument (as is done in pass-by-value), MATLAB passes only the memory address containing the argument. This is done in the interest of efficient use of memory and run-time. It would be very time-consuming and unnecessarily use lots of memory to copy a 100,000,000-element array into a local argument of a function. If a function modifies a passed argument (by assigning a value to it for example), then the pass-by-value mechanism is used. This has the benefit of avoiding any potential mistakes whereby a function inadvertently changes the value of an argument; the function will only be changing a copy of the argument and not the actual argument. Of course, if we wish to change the value of an argument (as a result of a function), then this must be done by assigning a new value to the argument (through the return value of the function) as shown in the next example.

Example 4: Assigning a new value to an argument when pass-by-reference is used.

```
function array =  clear_odd2(x)
% clear_odd(x) sets the odd elements of the array x to 0

for index = 1 : 2 : length(x)
 x(index) = 0;    % since we are modifying value of argument
end               % it will be a pass-by-value argument and
                  % we are working only with the argument
array = x;        % return the modified copy
```

Suppose we were to define an array called y in our workspace (after having cleared it):

```
>> clear
>> y = [ 1 : 5 ]

y =

     1     2     3     4     5
```

Next, we pass y (by value) to the function and assign the return value of the function back to variable y:

```
>> y = clear_odd2(y)

y =

     0     2     0     4     0
```

Example 5: Consider the following function:

```
function out = count_me_up
% adds 1 to a local variable initialized to 0 and
% returns this value
count = 0 ;
count = count + 1 ;
out = count;
```

What do you suppose would be the output if we were to invoke this function 5 times within a for loop as in:

```
for index = 1 : 5
  count_me_up;
end loop
```

The answer may come as a surprise to the casual reader:

```
ans =

     1


ans =

     1


ans =

     1


ans =

     1


ans =

     1
```

The reason for this behavior is that the variable `count` which is local to function `count_me_up` is initialized to 0 upon every function invocation. If we wish for a variable within a function to be initialized only once (the first time the function is called) and for its value to *persist* across different function invocations, then we must declare the variable to be `persistent` as in:

```
function out = count_me_up
% adds 1 to a persistent local variable initialized to 0
% and returns this value
% unfortunately this does not work...

persistent count = 0 ;
count = count + 1 ;
out = count;
```

If this file were defined in a script with filename count_me_up2.m, we would invoke it in a for loop as in:

```
>> for index = 1:5
count_me_up2
end
```

Doing so results in the following error:

```
Error: File: count_me_up2.m Line: 6 Column: 18
The expression to the left of the equals sign is not a valid tar-
get for an assignment.
```

The offending line is:

```
persistent count = 0 ;
```

Defining a variable to be persistent will result in an initially empty array being assigned to the variable [2]. The above statement is equivalent to doing:

```
>> count = [] = 0
 count = [] = 0
           |
Error: The expression to the left of the equals sign is not a
valid target for an assignment.
```

The solution is to leave the variable initialized to an empty array, then use the `isempty` function (which returns true if the argument is empty and false otherwise) to test for emptiness and if empty to set the starting value of `count` to 0 as in:

```
function out = count_me_up
% adds 1 to a  persistent local variable initialized to 0 and
% returns this value

persistent count ;

% check if count is the empty array and set it to 1 if yes
 if isempty(count)
  count = 0; % note this will be performed only 1 time
 end

count = count + 1 ;
out = count;
```

When invoked from our for loop, the value of count now lives across the 5 different times we call the function (assuming we saved the above function in a file called count_me_up3.m):

```
>> for index = 1:5
count_me_up3
end

ans =

     1

ans =

     2
ans =

     3

ans =

     4

ans =

     5
```

It should be readily apparent that MATLAB persistent variables are analogous the static variables in C++ functions (although the method of assigning a persistent variable in MATLAB is somewhat cumbersome compared to the initialization of static variables in C++).

Example 6: Global Variables

If a variable is to be used by many functions, then it is convenient to define the variable as a global variable. A global variable is visible to visible to all functions. The keyword `global` is placed in front of a variable name in order to declare it as a global variable. The following example defines a global variable called `counter` and two functions which modify the value of the global variable.

```
clear;
global counter ; % declare counter to be global
counter = 0 ;
disp('Value of counter = ');
disp(counter);
count_up;
count_up;
disp('Value of counter = ');
disp(counter);
count_down;
disp('Value of counter = ');
disp(counter);
```

The functions are defined as follows:

```
function [] = count_up
% add 1 to the global variable counter

global counter; % declare counter as a global variable
counter = counter + 1;
```

```
function [] = count_down
% subtracts 1 to the global variable counter

global counter;  % declare counter as a global variable
counter = counter -  1;
```

Sampling Theorem:

Consider a discrete time sinusoidal signal $x(n) = sin(\frac{2\pi}{N}n)$. The value of N defines the number of samples per period, and n defines the total number of data points stored. For example, with $N = 4$ and $n = 8$ we are sampling the signal 4 times per period over a total of 8 data points, which is equal to two periods. We can perform this in MATLAB with the following code:

```
>> n = [ 0 : 7 ];
>> N = 4 ;
>>  x = sin((2*pi)/N * n);
```

Figure 1 shows the stem plot produced with these parameters.

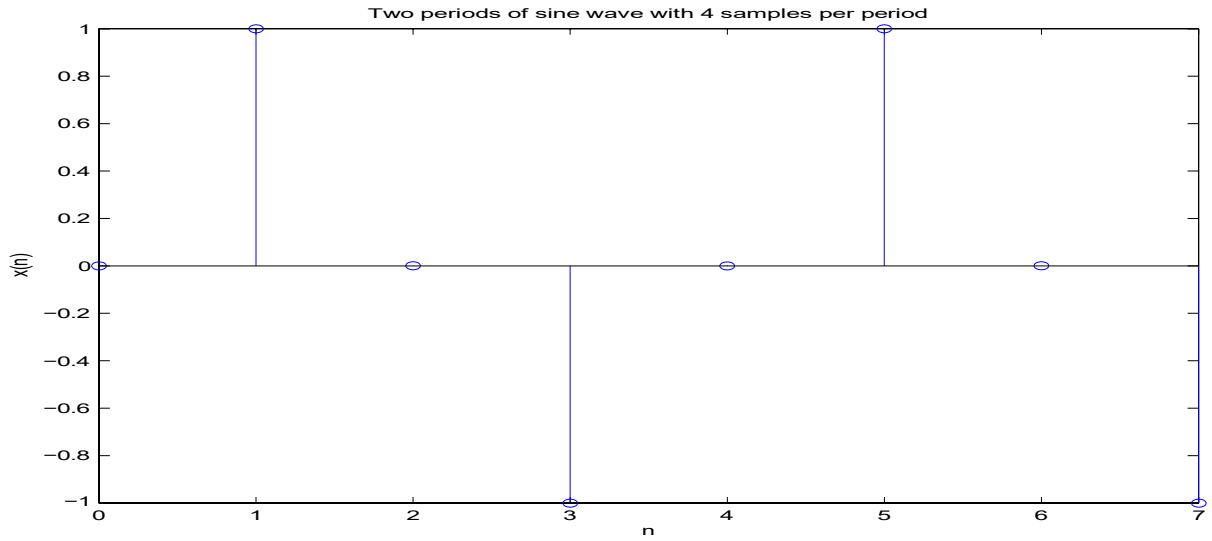

Figure 1: Sampling a sine wave with 4 samples per period over 2 periods.

With 8 samples per period, the sampled signal more closely resembles a sine wave as shown in Figure 2:

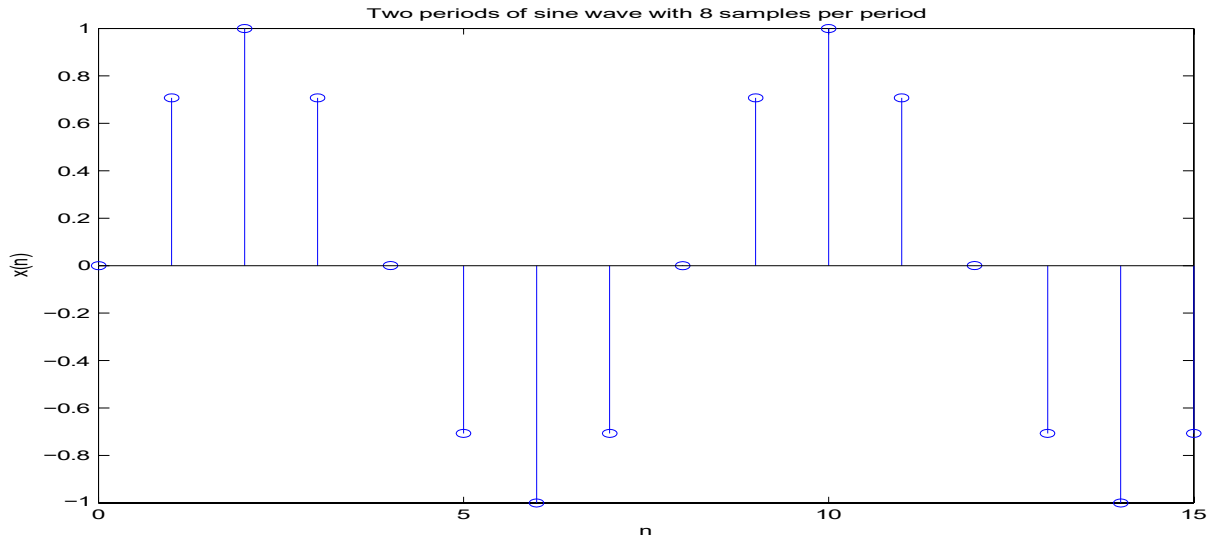

Figure 2: Sampled sine wave with 8 samples per period over two periods.

The rate at which a signal is sampled has an effect on its Fourier transform as shown by Figure 3 and 4; the higher the sampling rate the more the replicas of $X(jw)$ spread apart. The transforms are plotted over the interval -3π to 3π.

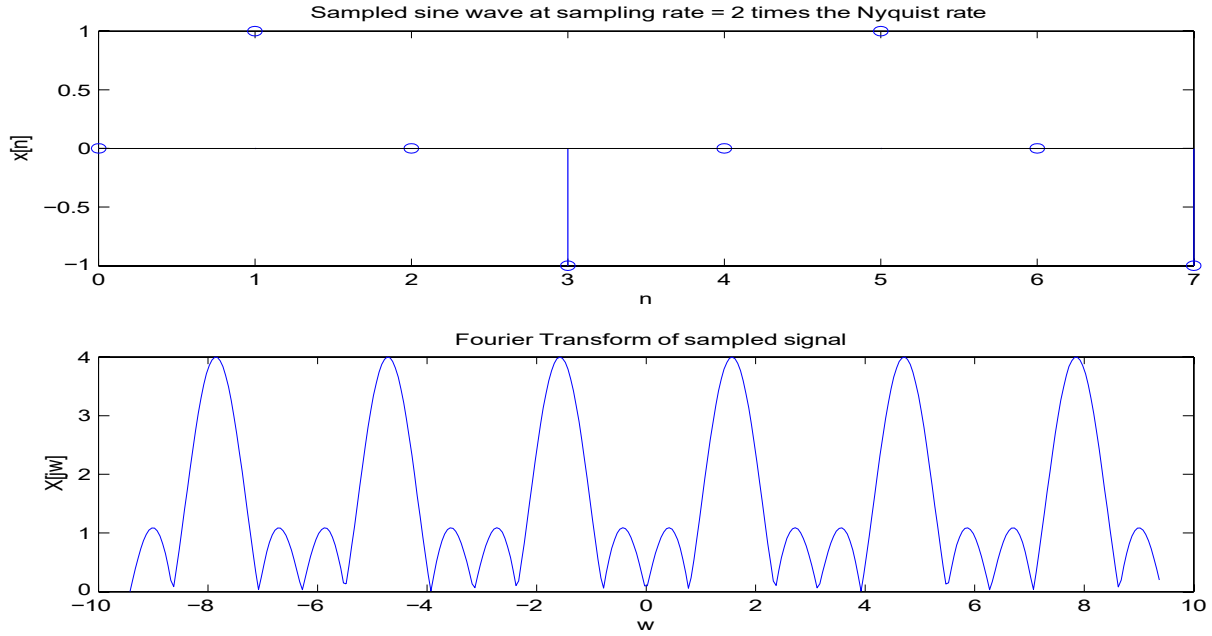

Figure 3: Sampled signal and its transform (4 samples per period).

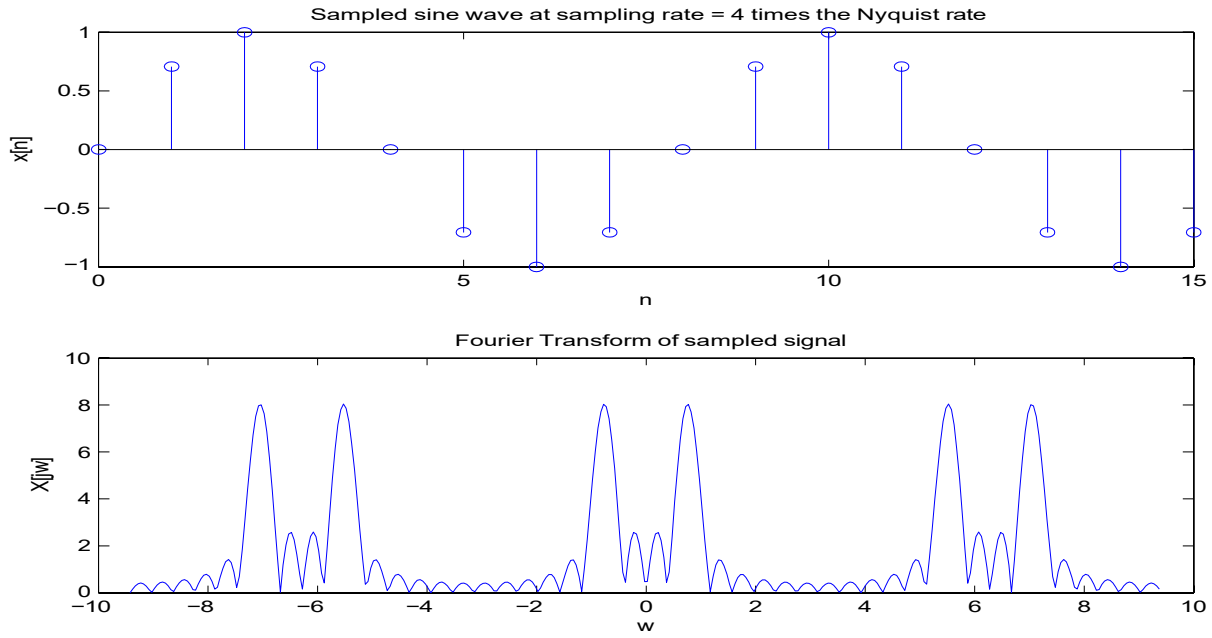

Figure 4: Sampled signal and its transform (8 samples per period).

In order to have no overlap (referred to aliasing), it is necessary to sample the signal at a rate at least twice that of the highest frequency component in the signal. This is known as the Nyquist rate. In terms of samples per period, the Nyquist rate corresponds to 2 samples per period.

Questions

Question 1: (30/80)

(a) This question will investigate the effect of varying the sampling rate on the transform of the sampled signal. Write a MATLAB script which will prompt the user to specify the number of periods and the step size over which the transform is to be computed. This will define the `w` array as
`[ -n *pi : step_size : n*pi ]` where n is the number of periods. The array `w` should be declared to be global.

Re-write your code used in Lab #2 as a function which receives a signal `x(n)` and returns the Discrete Time Fourier Transform of the signal. Make use of a function to compute the value of `N` for a given value of sampling rate.

The program is to ask the user to input the value of the sampling rate (in terms of the number of times of the Nyquist rate, i.e. 2 times, 3.6 times, etc). The program then plots in one figure window the sampled signal (using stem) and the Fourier transform of the sampled signal. This is to be repeated 5 times.

The pseudocode is as follows:

```
input the number of periods ;
input the step size of the frequency interval
w = [ - (number of periods ) * pi : step_size : (number of periods ) * pi ]
for loop = 1  : 5
   input the sampling rate ;
  determine N (number of samples in one period) based upon sampling rate;
  determine n (total number of samples needed to store two complete periods);
  compute x[n] = sin ( 2*pi /N * n);
  plot the signal over two complete periods;
  compute the fourier transform of the signal (by passing x as an argument to
  your function);
  plot the transform of the signal;
end
```

The following MATLAB functions will be helpful:

```
>> help floor
 floor  Round towards minus infinity.
    floor(X) rounds the elements of X to the nearest integers
    towards minus infinity.

>> floor ( 1.2 )

ans =

     1
```

Make use of the `floor` function when computing N to handle the case of non-integer values of sampling rate. For example, if we want to sample at 2.8 times the Nyquist rate, we have

N = number of samples in one period = 2.8 x 2 = 5.6
floor(5.6) = 5

```
>> help figure
 figure Create figure window.
    figure, by itself, creates a new figure window, and returns
    its handle.
```

The `figure` function is used to plot multiple graphs in separate figure windows.

(b) To investigate the effect of changing the window size (i.e. how many samples of the signal are obtained at some fixed sampling rate) we will rewrite the code in part (a) to ask the user to input the sampling rate (in terms of the Nyquist rate). This sampling rate will be kept constant and the program is to ask the user to enter the window size (in terms of the number periods of the signal). The sampled signal (over the total number of periods as defined by the value of the window

size) and its Fourier transform is to be plotted in one figure window. This is to be repeated 5 times with a different value for the window size in each loop iteration. The pseudocode is as follows:

```
input the sampling rate  (in terms of the Nyquist rate) ;
determine N (number of samples in one period) based upon the sampling rate;

for index = 1 : 5
   input the window size (in terms of the number of periods of the signal);
   determine n (total number of samples obtained based upon sampling rate and
                window size. n  = window size x N ) ;
   w = [ -(window size) * pi : 0.05 : pi*(window_size) ]; (for simplicity keep
                                                          step size fixed )
   compute x[n] = sin(2*pi/N * n) ;
   plot the signal over the window size ;
   compute the Fourier transform of the signal (by passing x as an argument to
                                                your function);
   plot the transform of the signal;
end
```

Question 2: (20/80)

Polar plots are useful for plotting signals which are periodic. For example, consider a "full-wave rectified'" sine wave (one in which the negative portions have been "inverted"). Such a signal can be readily created and plotted in MATLAB as follows:

```
%  Example script making use of a polar plot
clear
clf   % clear the current figure
% define the interval over three periods

n = [ 0 : 0.1 : 6*pi ] ;
x = sin(n);
x_rectified = abs(x) ; % obtain a "full-wave rectified" version
                       % of the sine wave
subplot(2,1,1)
stem(n,x_rectified)
xlabel('n')
ylabel('x[n]')
subplot(2,1,2)
polar(n, x_rectified);
```

Instead of using Cartesian coordinates, the `polar(angle,r)` command makes use of polar coordinates and plots the angle in radians against the radius r). A `help polar` gives the following:

```
>> help polar
 polar  Polar coordinate plot.
    polar(THETA, RHO) makes a plot using polar coordinates of
    the angle THETA, in radians, versus the radius RHO.
```

Note that negative values of `RHO` are reflected through the origin and are rotated by 180 degrees.[3] This is the reason why a polar plot of a sine wave consists of a single circle.

The plots produced by the sample script are shown in Figure 5 where we can clearly see the periodic nature of the signal in the polar plot representation.

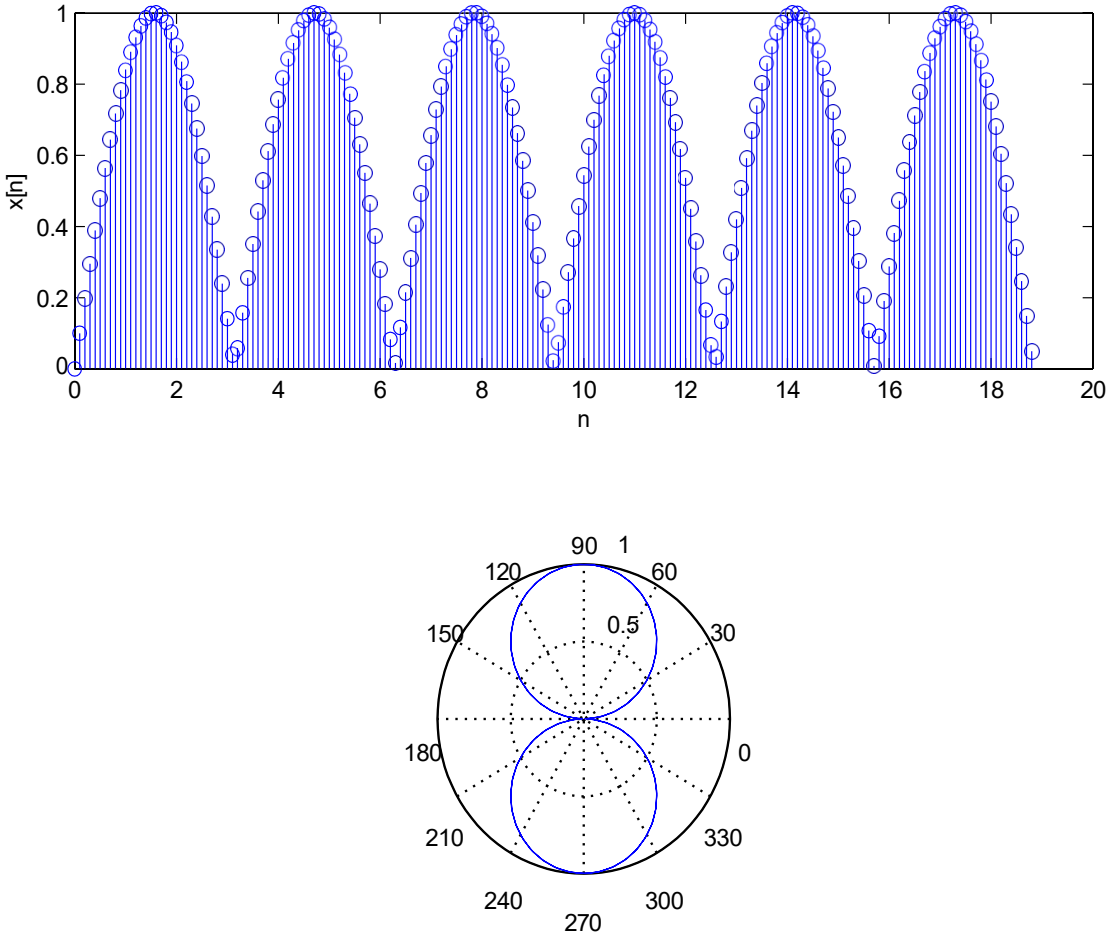

Figure 5: A rectified sine wave and it's polar plot.

Repeat Question 1(a) using polar plots for the transform instead of rectangular plots. Use the following for the input signal:

$$x[n] = 0.5sin\left(\frac{2\pi}{N}n\right) + 0.33sin\left(\frac{4\pi}{N}n\right)$$

Question 3: (30/80)

Sometimes a signal is a distorted version of another signal, for example when a signal passes through a communication channel or when a signal is compressed. In order to recover the original signal from the distorted one, we can use an LTI system. This system receives the distorted signal and returns the recovered one. This LTI system can be designed in frequency domain or time domain depending on the type of distortion.

To measure the difference between two signals we use mean square error (MSE) as a metric. The MSE for given signals $x[k]$ and $y[k]$ with length of L is defined as:

$$MSE = \frac{1}{L}\sum_{k=1}^{L}(x[k] - y[k])^2$$

- From Moodle site to find two data files: *Original.wav* and *Distorted.wav* provide the original signal with length of 2 seconds and its distorted version, respectively. The sampling frequency for these audio signals is Fs = 22050 Hz and number of bits per sample is 16. Thus, the number of samples in each signal is L = 44100. To load audio files into your programs you can use function *wavread* as:

```
[x, Fs, numBitsPerSample] = wavread('Original.wav', 44100);
```

- To play a sound signal x you can use function *sound* as:

```
sound(x, Fs , numBitsPerSample);
```

- To write an audio signal into a file you can use *wavwrite* function as:

```
wavwrite(x,Fs,bits,'Recovered.wav');
```

where `Fs` and `numBitsPerSample` here are 22050 and 16 respectively.

Tasks:

(a) Load two files *Original.wav* and *Distorted.wav* to your program and plot them in time domain.

(b) Use MSE to compare the original signal and the distorted.

(c) Design a system that recovers the original signal from the distorted signal and save it to a file called *Recovered.wav*. Which domain did you choose for the design: frequency domain or time domain? Why?

(d) Compute the MSE between the recovered signal and the original signal. Does your system improve the MSE?

(e) By playing and listening to recovered signal, does your system improve the quality of the sound?

References

1. *MATLAB Programming*, David C. Kuncicky, Pearson Education Inc., 2004, p.139.

2. *MATLAB Programming*, David C. Kuncicky, Pearson Education Inc., 2004, p. 145.

3. MATLAB online help documentation “doc polar”.