Adopted from Dr. Havlicek's assignment from OU.EDU

Project 3

What to Turn In:

Make one file that contains your solution for this assignment. For Problem 7, you must also include your final filtered signal in a wave file named `proj3_filteredsig.wav`. Make sure to include your name in your turn-in file. Number the problems and paste in the figures and graphs on a word file and make sure to include answers for the discussion questions. Also include matlab script file. Upload a zip folder.

Submit yelectronically on Canvas.

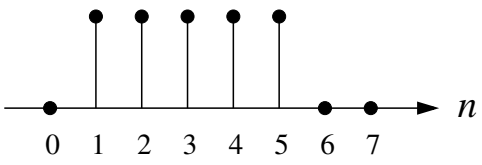
The Assignment:

In this project, we are going to work with discrete-time signals that have a finite length. To make the mathematics work out right, it is *very important* that our finite-length signals will always start at $n = 0$ and end at $n = N - 1$. So, an 8-point signal $x[n]$ will have length $N = 8$. It will be defined from $n = 0$ to $n = 7$. Outside of this range of n's, the signal is not defined. It's important for you to understand that $x[n]$ is *not* equal to zero for other values of n; it does not even exist for other values of n.

We are also going to work with digital audio signals. We will make some simple audio signals, play them through the sound card, and write them to wave files so that they can be played by a media player like *Windows Media Player*. Then you will design a high-performance digital filter to remove noise from a complex audio signal.

PART I: the DFT

Here is a simple example of an 8-point signal $x_1[n]$:



We can write the signal $x_1[n]$ like this:

$$x_1[n] = \delta[n-1] + \delta[n-2] + \delta[n-3] + \delta[n-4] + \delta[n-5], \quad 0 \le n \le 7. \tag{1}$$

In Matlab, you can represent this signal with a vector (array)

```
x1n = [0 1 1 1 1 1 0 0];
```

You must always remember that the first element of the Matlab array is `x1n(1)`, but in terms of the finite-length signal this is $x_1[0]$, which is for $n = 0$. Similarly, the last element of the Matlab array is `x1n(8)`, but this is really $x_1[7]$, corresponding to $n = 7$.

Because the signal $x_1[n]$ is defined only for $0 \le n \le 7$ and *not* for all $n \in \mathbb{Z}$, it does *not* have a discrete-time Fourier transform $X_1(e^{j\omega})$. But it does have an 8-point *discrete Fourier transform* (DFT) $X_1[k]$. Like $x_1[n]$, $X_1[k]$ also has length $N = 8$. It is defined for $0 \le k \le 7$ and it takes complex values. If you want to graph it, you can graph the real part and the imaginary part, or you can graph the magnitude and the angle (phase). Usually, we graph the magnitude and the phase.

The 8-point DFT $X_1[k]$ is given by

$$X_1[k] = \sum_{n=0}^{7} x_1[n] e^{-j2\pi kn/8}, \quad 0 \le k \le 7. \tag{2}$$

It gives us a way to write an 8-point signal like $x_1[n]$ as a sum of the eight DFT basis functions $e^{j0\frac{2\pi}{8}n}$, $e^{j1\frac{2\pi}{8}n}$, $e^{j2\frac{2\pi}{8}n}$, $e^{j3\frac{2\pi}{8}n}$, $e^{j4\frac{2\pi}{8}n}$, $e^{j5\frac{2\pi}{8}n}$, $e^{j6\frac{2\pi}{8}n}$, and $e^{j7\frac{2\pi}{8}n}$. Notice that each one of these basis functions is a complex sinusoid and their frequencies are given by $k(2\pi/8) = k\omega_0$, where $\omega_0 = 2\pi/N$ and $0 \le k \le 7$. As always, the transform (2) is the inner product (dot product) between the signal $x_1[n]$ and the basis functions.

To compute the DFT of $x_1[n]$, we have to compute the complex number $X_1[k]$ for each k from 0 to 7. So all together, we have to do the equation (2) eight times – once for each k. For each time, the sum (2) requires eight complex multiply-add operations. So the overall computational complexity is 64 complex multiply-add operations. More generally, for an N-point signal the computational complexity is N^2 complex multiply-add operations.

The *fast Fourier Transform* (FFT) is a family of fast algorithms that rearrange the terms of the sum (2) in a tricky way that makes maximum re-use of partial products. This reduces the computational complexity from N^2 complex multiply-add operations to $N \log(N)$ complex multiply-adds. For an 8-point signal, it reduces the complexity from 64 multiply-adds to 40 multiply adds. For $N = 4096$, it reduces the complexity from about 16.8 million multiply-adds to 49,152 multiply-adds.

Once you have used the DFT to compute the eight inner products $X_1[k]$ for $k = 0, 1, 2, \ldots, 7$, then you can write the signal $x_1[n]$ by adding up the inner products times the basis functions. This is called the *inverse discrete Fourier transform* (IDFT). It is given by

$$x_1[n] = \frac{1}{8} \sum_{k=0}^{7} X_1[k] e^{j2\pi kn/8}, \quad 0 \le n \le 7. \tag{3}$$

Matlab provides a built-in function `fft` that uses the FFT algorithm to compute the DFT in Eq. (2). Matlab also provides a built-in function `ifft` that uses the FFT algorithm to compute the IDFT (3).

1. Consider the following Matlab code which computes the DFT of the signal $x_1[n]$ in (1) and plots the DFT magnitude and phase as functions of k. The program also plots the DFT magnitude as a function of the Matlab array index and as a function of the radian digital frequency $k(2\pi/8)$. Type in this code and run it. You can type it in line-by-line at the command prompt or you can create an m-file. You can download a copy of this code as an m-file from the "Files for Matlab 04" section of the course page on `canvas.ou.edu`.

```
%-------------------------------------------------------
% P1
%
% - Create and plot the signal x_1[n] as a function of n.
% - Compute the DFT X_1[k].  Plot the magnitude and phase
%   as functions of k.
% - Plot the DFT magnitude as a function of the matlab
%   array index.
% - Plot the DFT magnitude as a function of the discrete
%   radian frequency w.
% - Compute and plot the IDFT.
%
n = 0:7;                          % time variable
x1n = [0 1 1 1 1 1 0 0];          % our 8-point signal
X1k = fft(x1n);                   % compute the DFT
X1kmag = abs(X1k);                % magnitude of the DFT
X1karg = angle(X1k);              % phase of the DFT

% plot the signal
figure(1);
stem(n,x1n);
axis([0 7 0 1.5]);
title('Original Signal');
xlabel('n');
ylabel('x_1[n]');

% plot DFT magnitude and phase as functions of k
k = 0:7;                          % frequency index
figure(2);
stem(k,X1kmag); ylim([0 6]);
```

```
title('DFT Magnitude');
xlabel('k');
ylabel('|X_1[k]|');
figure(3);
stem(k,X1karg);
title('DFT Phase');
xlabel('k');
ylabel('arg(X_1[k])');

% plot DFT magnitude as a function of Matlab index
Matlab_idx = [1:8];               % Matlab index
figure(4);
stem(Matlab_idx,X1kmag); ylim([0 6]);
title('DFT Magnitude');
xlabel('Matlab index');
ylabel('|X_1[index]|');

% plot DFT magnitude as a function of discrete frequency
%  (radians per sample)
w = [0:2*pi/8:7*2*pi/8];          % discrete frequency
figure(5);
stem(w,X1kmag); ylim([0 6]);
title('DFT Magnitude'); ylim([0 6]);
xlabel('discrete radian frequency \omega');
ylabel('|X_1[\omega]|');

% Compute and plot the IDFT
x2n = ifft(X1k);
figure(6);
stem(n,x2n);
axis([0 7 0 1.5]);
title('IDFT');
xlabel('n');
ylabel('IDFT');
```

People often refer to the eight numbers $X_1[k]$ as "the DFT coefficients" of the signal $x_1[n]$. Here is a table that shows, for each DFT coefficient $X_1[k]$, the Matlab array index, the DFT frequency index k, the digital frequency ω in radians per sample, and the digital frequency f in cycles per sample:

Matlab array index	1	2	3	4	5	6	7	8
DFT freq index k	0	1	2	3	4	5	6	7
ω, rad/sample	$0(2\pi/8)$	$1(2\pi/8)$	$2(2\pi/8)$	$3(2\pi/8)$	$4(2\pi/8)$	$5(2\pi/8)$	$6(2\pi/8)$	$7(2\pi/8)$
f, cycles/sample	0/8	1/8	2/8	3/8	4/8	5/8	6/8	7/8

Now we need to remember two important things about discrete-time complex sinusoids. First, we only need frequencies from $-\pi$ to π to make all the possible graphs. Second, subtracting any integer multiple of 2π from the frequency does not change the graph.

In the table above, notice that the radian frequencies for the DFT coefficients with $k = 5$, 6, and 7 are all $\geq \pi$. Subtracting 2π from the frequency for the $k = 5$ DFT coefficient, we get $5(2\pi/8) - 8(2\pi/8) = -3(2\pi/8)$. This does not change the graph of the basis function at all. So we can think of the $k = 5$ DFT coefficient as being for frequency $+5(2\pi/8)$, or, equivalently, as being for frequency $-3(2\pi/8)$.

Similarly, we can think of the $k = 6$ DFT coefficient as being for frequency $+6(2\pi/8)$ or for frequency $6(2\pi/8) - 8(2\pi/8) = -2(2\pi/8)$. And we can think of the $k = 7$ DFT coefficient as being for frequency $+7(2\pi/8)$ or for frequency $-1(2\pi/8)$.

Finally, we can think of the $k = 4$ DFT coefficient as being for frequency $+4(2\pi/8)$ or for frequency $4(2\pi/8) - 8(2\pi/8) = -4(2\pi/8)$. Notice that this is the "$N/2$" coefficient, where $N = 8$ is the length of the signal. People often refer to this DFT coefficient as being for both of the frequencies $\pm 4(2\pi/8)$.

So let's draw the table again, but this time we will subtract 2π from the frequencies of the $k = 4$, 5, 6, and 7 DFT coefficients. It's important for you to remember that this does not change anything. The signals $e^{j5\frac{2\pi}{8}n}$ and $e^{-j3\frac{2\pi}{8}n}$ have exactly the same graph. They are just two different ways of writing the same DFT basis signal. So here's the new table:

Matlab array index	1	2	3	4	5	6	7	8
DFT freq index k	0	1	2	3	4	5	6	7
ω, rad/sample	$0(2\pi/8)$	$1(2\pi/8)$	$2(2\pi/8)$	$3(2\pi/8)$	$\pm 4(2\pi/8)$	$-3(2\pi/8)$	$-2(2\pi/8)$	$-1(2\pi/8)$
f, cycles/sample	0/8	1/8	2/8	3/8	$\pm 4/8$	$-3/8$	$-2/8$	$-1/8$

From this new table, you can see that the DFT coefficients in the first half of the array are for the positive frequencies, but the coefficients in the second half of the array are actually for the *negative* frequencies.

So, when we look at a DFT array in practice, we usually *swap* the left and right sides of the array so that the negative frequency coefficients are on the left, the zero frequency (DC) coefficient is in the center, and the positive frequency coefficients are on the right.

Here is what the table looks like *after* we swap the left and right halves of the array:

NEW Matlab index	1	2	3	4	5	6	7	8
OLD Matlab index	5	6	7	8	1	2	3	4
DFT freq index k	4	5	6	7	0	1	2	3
ω, rad/sample	$\pm 4(2\pi/8)$	$-3(2\pi/8)$	$-2(2\pi/8)$	$-1(2\pi/8)$	$0(2\pi/8)$	$1(2\pi/8)$	$2(2\pi/8)$	$3(2\pi/8)$
f, cycles/sample	$\pm 4/8$	$-3/8$	$-2/8$	$-1/8$	$0/8$	$1/8$	$2/8$	$3/8$

This is called the *centered DFT*. Matlab provides a built-in function `fftshift` to center the DFT array for you. Matlab also provides a built-in function `ifftshift` to un-center it. If you center a DFT, then you must **always** un-center it before you try to invert!

So, for example, if you wanted to compute the magnitude and phase of the centered DFT and then invert, you could do it like this:

```
X1kshift = fftshift(fft(x1n));
X1kmag = abs(X1kshift);
X1karg = angle(X1kshift);
x2n = ifft(ifftshift(X1kshift));
```

2. Modify the Matlab code in Problem 1 to compute and plot the magnitude and phase of the centered DFT for the signal $x_1[n]$ in (1). Plot the centered magnitude and phase as functions of the radian frequency ω and of the Hertzian frequency f. Also compute and plot the inverse DFT.

   Hint: you may find the following Matlab statement helpful for making the "x-axis" quantities to use in the `plot` command:

   ```
   w = [-4*2*pi/8:2*pi/8:3*2*pi/8];
   ```

3. As you saw in Problems 1 and 2, plotting the DFT magnitude and phase as functions of radian frequency ω is a little bit inconvenient. In Problem 2, we wanted the x-axis of these plots to go from $-\pi$ to $3(2\pi/8)$. But what we got were plots from -4 to 3. Matlab doesn't like it when the first and last ticks on the x-axis are irrational numbers like π.

   Because of this, people often plot the DFT magnitude and phase using a *normalized* radian frequency axis. The normalized frequency is given by ω/π. Then, -1 on the normalized frequency axis corresponds to $\omega = -\pi$ and $+1$ on the normalized frequency axis corresponds to $\omega = +\pi$. This makes the Matlab plots turn out a little bit nicer. Here are some Matlab statements that show you how to do this:

   ```
   w = [-4*2*pi/8:2*pi/8:3*2*pi/8];
   stem(w/pi,X1kmag);
   xlabel('\omega/\pi');
   ```

Modify your Matlab code from Problem 2 to plot the centered DFT magnitude for $x_1[n]$ using a normalized frequency axis.

It's important for you to know about normalized frequency because the digital IIR filter design routines in the Matlab Signal Processing Toolbox require you to specify the passband and stopband edge frequencies in units of normalized frequency.

Now, as we said back near the top of page 2, the finite-length signal $x_1[n]$ in (1) does not have a DTFT $X_1(e^{j\omega})$. But suppose that we make a new signal $\hat{x}_1[n]$ by adding zeros to both sides of $x_1[n]$ so that the new signal is defined for all $n \in \mathbb{Z}$. In other words, we make the new signal

$$\hat{x}_1[n] = \begin{cases} x_1[n], & 0 \le n \le 7, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Then $\hat{x}_1[n]$ *does* have a DTFT $\widehat{X}_1(e^{j\omega})$. The relationship between the 8-point DFT $X_1[k]$ and the DTFT $\widehat{X}_1(e^{j\omega})$ is that $X_1[k]$ is given by eight equally spaced samples of $\widehat{X}_1(e^{j\omega})$ going from $\omega = 0$ to $\omega = 7(2\pi/8)$. The 8-point centered DFT of $x_1[n]$ is given by eight equally spaced samples of $\widehat{X}_1(e^{j\omega})$ going from $\omega = -\pi$ to $\omega = 3(2\pi/8)$.

To investigate this further, let's compute the DTFT $\widehat{X}_1(e^{j\omega})$. Unfortunately, $\hat{x}_1[n]$ is not in Table 5.2. However, the signal

$$\hat{x}_0[n] = \begin{cases} 1, & |n| \le 2, \\ 0, & |n| > 2, \end{cases} \tag{5}$$

is in Table 5.2. And $\hat{x}_1[n] = \hat{x}_0[n-3]$. According to the table,

$$\widehat{X}_0(e^{j\omega}) = \frac{\sin\left(\frac{5}{2}\omega\right)}{\sin(\omega/2)}. \tag{6}$$

Applying the DTFT time shifting property from Table 5.1, we get

$$\widehat{X}_1(e^{j\omega}) = e^{-j3\omega}\widehat{X}_0(e^{j\omega}) = \frac{\sin\left(\frac{5}{2}\omega\right)}{\sin(\omega/2)}e^{-j3\omega}. \tag{7}$$

4. Consider the following Matlab code which plots the magnitude and phase of the DTFT $\widehat{X}_1(e^{j\omega})$ together with the magnitude and phase of the centered DFT of $x_1[n]$:

```
%-------------------------------------------------------
% P4a
%
% Show that the DFT is given by samples of the DTFT.
% - plot the DTFT magnitude of x1hat from -pi to pi.
%   - plot the centered DFT magnitude of x_1[n] on the
%     same graph.
% -plot the DTFT phase of x1hat from -pi to pi.
%   - plot the centered DFT phase of x_1[n] on the same
%     graph.
%

% Frequency vector for plotting the DTFT.  Use 1000 points.
w = linspace(-pi,pi,1000);

% The DTFT was computed analytically
X1hat = sin(2.5*w)./sin(w/2) .* exp(-3*j*w);
X1hatmag = abs(X1hat);
X1hatarg = angle(X1hat);

% Now compute the 8-point DFT
x1n = [0 1 1 1 1 1 0 0];  % our 8-point signal
k =  -4:3;           % frequency index for the centered DFT
X1k = fftshift(fft(x1n));
X1kmag = abs(X1k);
X1karg = angle(X1k);

figure(1);
plot(w,X1hatmag,'-b');    % plot the DTFT magnitude
axis([-pi pi 0 6]);
hold on;                  % makes the next plot come out on the
                          %   same graph
plot(k*2*pi/8,X1kmag,'ro');  % plot the centered DFT magnitude
hold off;                    %   using a symbol, but no line
                             %   and no stem.
title('Magnitude of DTFT and centered 8-pt DFT');
xlabel('\omega','FontSize',14);
ylabel('$|\widehat X_1(e^{j\omega})|$, $|X_1[\omega]|$',...
       'Interpreter','latex','FontSize',14);
```

```
legend('DTFT','DFT');

figure(2);
plot(w,X1hatarg,'-b');    % plot the DTFT phase
axis([-pi pi -4 5]);
hold on;
plot(k*2*pi/8,X1karg,'ro');  % plot the centered DFT phase
hold off;
title('Phase of DTFT and centered 8-pt DFT');
xlabel('\omega','FontSize',14);
ylabel('$\arg\widehat X_1(e^{j\omega})$, $\arg X_1[\omega]$',...
       'Interpreter','latex','FontSize',14);
legend('DTFT','DFT');
```

(a) Type in this code and run it. You can type it in line-by-line at the command prompt or you can create an m-file. You can download a copy of this code as an m-file from the "Files for Matlab 04" section of the course page on `canvas.ou.edu`.

(b) Note that if we add zeros to the right side of $x_1[n]$, then it will make the finite-length signal and the DFT longer. But it will not change $\hat{x}_1[n]$ and $\widehat{X}_1(e^{j\omega})$. If, for example, we change $x_1[n]$ to

```
x1n = [0 1 1 1 1 1 0 0 0 0 0 0];
```

then the length of both $x_1[n]$ and $X_1[k]$ are increased to $N = 12$. But $\hat{x}_1[n]$ is still given by (4) and $\widehat{X}_1(e^{j\omega})$ is still given by (7).

This gives us a way to sample the DTFT $\widehat{X}_1(e^{j\omega})$ with arbitrary density by using zero padding to increase the length of $x_1[n]$.

Modify the Matlab code in Problem 4(a) to plot the magnitude and phase of the DTFT together with the magnitude and phase of the centered DFT of $x_1[n]$ for a length of $N = 16$. To do this, you must change three things in the program P4a. First, you must increase the length of `x1n` to $N = 16$ by appending zeros to the end of the signal. Second, you must change the DFT frequency index to go from $k = -8$ to $k = 7$ instead of $k = -4$ to $k = 3$. Third, in the plot commands for `X1kmag` and `X1karg`, you must change the DFT frequency vector from `k*2*pi/8` to `k*2*pi/16`. You should also change the titles of the plots to reflect the fact that it is now a 16-point DFT.

Now assume that N is an even positive integer (like 1024, for example). For an N-point finite-length signal $x[n]$ defined for $0 \le n \le N-1$, the N-point DFT is given by

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}, \quad 0 \le k \le N-1. \tag{8}$$

If the values of the N-point signal $x[n]$ are stored in the Matlab array `xn`, then the Matlab statement

```
Xk = fft(xn);
```

will place the N complex-valued DFT coefficients in the Matlab array `Xk`. As the Matlab index ranges from 1 to N, the DFT frequency index k ranges from 0 to $N-1$. The Matlab array elements `Xk(1)` through `Xk(N)` contain the DFT coefficients $X[0]$ through $X[N-1]$, which are for radian digital frequencies going from $\omega = 0$ to $\omega = (N-1)\frac{2\pi}{N}$ in steps of $\frac{2\pi}{N}$. If $N = 8$, then everything is exactly as shown in the table on page 5.

However, it's usually more intuitive to work with the centered DFT. The Matlab statement

```
Xk = fftshift(fft(xn));
```

will again place the N complex-valued DFT coefficients in the Matlab array `Xk`. But this time,

- the first half of the Matlab array, i.e., Matlab array elements `Xk(1)` through `Xk(N/2)`, will contain the DFT coefficients $X[N/2]$ through $X[N-1]$ which are for radian digital frequencies going from $\omega = -\pi$ to $\omega = -\frac{2\pi}{N}$ in steps of $\frac{2\pi}{N}$.
- the Matlab array element `Xk(N/2 + 1)` will contain the DFT DC coefficient $X[0]$, which is for radian digital frequency $\omega = 0$.
- the last half of the Matlab array, i.e., Matlab array elements `Xk(N/2 + 2)` through `Xk(N)`, will contain the DFT coefficients $X[1]$ through $X[N/2-1]$ which are for radian digital frequencies going from $\omega = \frac{2\pi}{N}$ to $\omega = \pi - \frac{2\pi}{N}$ in steps of $\frac{2\pi}{N}$.

In other words, in an N-point centered DFT array, the radian digital frequency goes from $-\pi$ to $\pi - \frac{2\pi}{N}$ in steps of $\frac{2\pi}{N}$. If $N = 8$, then everything is exactly as shown in the second table on page 5.

Discrete Hertzian frequency (cycles per sample) is obtained by dividing the radian digital frequency ω (radians per sample) by 2π. In an N-point centered DFT array, the Hertzian digital frequency goes from $-\frac{1}{2}$ cycle per sample to $\frac{1}{2} - \frac{1}{N}$ cycles per sample in steps of $\frac{1}{N}$.

Normalized frequency is obtained by dividing the radian digital frequency ω by π. In an N-point centered DFT array, the normalized frequency goes from -1 to $1 - \frac{2}{N}$ in steps of $\frac{2}{N}$.

The N-point inverse DFT (IDFT) is given by

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j2\pi kn/N}, \quad 0 \le n \le N-1. \tag{9}$$

For an un-centered DFT array, the N-point IDFT can be computed using the Matlab statements

```
Xk = fft(xn);
xn = ifft(Xk);
```

The Matlab statements for a centered DFT array are:

```
Xk = fftshift(fft(xn));
xn = ifft(ifftshift(Xk));
```

Note: The DFT is usually written using the special symbol

$$W_N = e^{-j2\pi/N}.$$

For any *fixed* value of N, W_N is a *constant*. In terms of W_N, the DFT and IDFT equations (8) and (9) become

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \quad 0 \le k \le N-1, \tag{10}$$

and

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}, \quad 0 \le n \le N-1. \tag{11}$$

Although this is the way that you will usually see the DFT written, we are not going to use the "W_N notation" in this assignment. It would only make things more complicated.

PART II: Digital Audio

Professional compact disc digital audio is sampled with a sampling frequency of $F_s = 44.1$ kHz. This means that there are 44,100 samples per second. The time interval between samples is called the *sampling period*. It is given by $T_s = 1/F_s \approx 22.676$ μsec. Although professional compact disc audio signals are stereo and have two channels of audio data, in this assignment we will only consider single-channel (mono) audio signals.

The audio samples are stored as 16-bit two's complement integers. In high performance professional applications, they are stored without compression. For computer processing, the digital audio samples are usually stored in a *wave file* (`.wav`). Matlab provides a built-in function `audioread` that can read the digital audio data in a wave file into a Matlab array. It also provides a built-in function `audiowrite` that can write the digital audio data in a Matlab array out to a wave file.

The Matlab statement

```
[x,Fs] = audioread('test.wav');
```

reads the digital audio signal contained in the file `test.wav` into the Matlab array `x`. The sampling rate, which is stored in the wave file, is placed in the Matlab variable `Fs`. For professional audio, it is always 44.1 kHz. When Matlab reads in the 16-bit two's complement integer audio samples, it converts them to double precision floating point numbers in the range $[-1, 1]$.

The Matlab statement

```
audiowrite('test.wav',x,Fs);
```

writes the digital audio data stored in the double precision floating point array `x` out to the wave file `test.wav` in 16-bit two's complement integer format. It is important for you to make sure that the digital audio data in the array `x` is normalized to the range $[-1, 1]$ before you call `audiowrite`, since operations like filtering and adding signals together will generally change the range. This can be done by placing the Matlab statement

```
x = x / max(abs(x));
```

just before the call to `audiowrite`.

The Matlab statement

```
sound(x,Fs,16);
```

will play the digital audio data in the array `x` through the sound card as 16-bit two's complement integers with a sampling frequency of `Fs`. As with `audiowrite`, it is important for you to ensure that the data stored in `x` are normalized to the range $[-1, 1]$ before you call `sound`.

The Nyquist frequency is given by $F_s/2 = 22.05$ kHz. A/D and D/A conversion, i.e., sampling, maps the analog Nyquist frequency to the digital frequency $\omega = \pi$ radians per

sample. Thus, if the Matlab array `x` holds an N-point digital audio signal, then in the N-point centered DFT array `X = fftshift(fft(x))` the analog frequency goes from $-\frac{Fs}{2}$ to $\frac{Fs}{2} - \frac{Fs}{N}$ in steps of $\frac{F_s}{N}$.

For practical digital audio signals, the magnitude of the centered DFT is usually plotted in dB as $20 \log_{10} |X[k]|$. Note that this will make a numerical error if $|X[k]| = 0$. So, if there are places k where $|X[k]| = 0$, you have to change it to a small nonzero number instead when you compute the logarithm.

For a digital audio signal with a sampling rate of F_s Hz,

- to convert (Hertzian) analog frequency to radian digital frequency, multiply the analog frequency by $\frac{2\pi}{F_s}$.
- to convert (Hertzian) analog frequency to Hertzian digital frequency, multiply the analog frequency by $\frac{1}{F_s}$.
- to convert (Hertzian) analog frequency to normalized digital frequency, multiply the analog frequency by $\frac{2}{F_s}$.

5. Recall that for professional compact disc digital audio, the sampling rate is $F_s = 44.1$ kHz. On a piano, the first "A note" that is located above middle C on the keyboard has an analog frequency of 440 Hz. Consider the Matlab code below, which does the following:

   - Makes a two-second digital audio cosine signal with analog frequency 440 Hz (this will require 44,100 $\times$ 2 = 88,200 samples). Such a signal is called a "pure tone."
   - Plays the 440 Hz pure tone through the sound card.
   - Plots the centered DFT magnitude in dB as a function of Hertzian analog frequency, radian digital frequency, and normalized digital frequency.
   - Writes the signal to a wave file.
   - Reads the signal back in from the wave file.
   - Plays the read in signal through the sound card.

```
%-------------------------------------------------------
% P5a
%
% Make a 2 second digital audio signal that contains a pure
% cosine tone with analog frequency 440 Hz.
% - play the signal through the sound card
% - plot the centered DFT magnitude in dB against
%     Hertzian analog freq, radian digital freq,
%     and normalized digital freq.
```

```
% - Write the signal to a wave file, read it back in, and
%     play it through the sound card again.
%

Fs = 44100;                     % sampling frequency in Hz
N = Fs * 2;                     % length of the 2 sec signal
n = 0:N-1;                      % discrete time variable
f_analog = 440;                 % analog frequency in Hz
w_dig = 2*pi*f_analog/Fs;       % radian digital frequency
x = cos(w_dig * n);             % the signal

% Normalize samples to the range [-1,1]
%  Not really needed here b/c cos is already in this range,
%  but done anyway to illustrate how you normalize.

x = x / max(abs(x));

sound(x,Fs,16);                 % play it through sound card

X = fftshift(fft(x));           % centered DFT
Xmag = abs(X);                  % centered DFT magnitude
XmagdB = 20*log10(Xmag);        % convert to dB

% Plot the centered magnitude against analog frequency
w = -pi:2*pi/N:pi-2*pi/N;       % dig rad freq vector
f = w * Fs /(2*pi);             % analog freq vector
figure(1);
plot(f,XmagdB);
xlim([-20000 20000]);
title('Centered DFT Magnitude for 440 Hz Pure Tone');
xlabel('analog frequency, Hz');
ylabel('dB');

% Plot the centered magnitude against radian digital freq
figure(2);
plot(w,XmagdB);
xlim([-pi pi]);
title('Centered DFT Magnitude for 440 Hz Pure Tone');
xlabel('radian digital frequency \omega');
ylabel('dB');

% Plot against normalized digital frequency
```

```
figure(3);
plot(w/pi,XmagdB);
xlim([-1 1]);
title('Centered DFT Magnitude for 440 Hz Pure Tone');
xlabel('normalized digital frequency \omega/\pi');
ylabel('dB');

% wait 3 seconds in case sound card is still busy
pause(3);

audiowrite('A-440.wav',x,Fs);   % write to wave file
[x2,Fs] = audioread('A-440.wav'); % read it back in
sound(x2,Fs,16);                % play it again Sam!
```

(a) Type in this code and run it. You can type it in line-by-line at the command prompt or you can create an *m*-file. You can also download it from `canvas.ou.edu`.

(b) Modify the Matlab code to generate and play a cosine pure tone with an analog frequency of 5 kHz.

Now we are going to do some filtering. The following Matlab code will design a lowpass digital Butterworth filter:

```
Wp = 0.4;
Ws = 0.6;
Rp = 1;
Rs = 60;
[Nf, Wn] = buttord(Wp,Ws,Rp,Rs);
[num,den] = butter(Nf,Wn);
```

The frequency response magnitude is shown in Fig. 1 on the next page. The x-axis is in normalized digital frequency and the y-axis is in dB. Since $|H(e^{j\omega})|$ is even symmetric, we normally plot it for the non-negative frequencies only.

The parameter `Wp` specifies the passband edge frequency. For this filter, we set `Wp` to a normalized digital frequency of 0.4. This makes the passband go from DC to 0.4π rad/sample. In the passband, $|H(e^{j\omega})| \approx 1 = 0$ dB. The parameter `Rp` specifies the allowable passband ripple, which is the amount that $|H(e^{j\omega})|$ is allowed to deviate from 0 dB in the passband. For this filter, we set `Rp` to 1. This means that $|H(e^{j\omega})|$ has to be between -1 dB and +1 dB everywhere in the passband.

The parameter `Ws` specifies the stopband edge frequency. For this filter, we set `Ws` to a normalized digital frequency of 0.6. So the stopband goes from 0.6π rad/sample up to π rad/sample. The parameter `Rs` specifies the minimum stopband attenuation. For this filter, we set `Rs` to 60 dB. This means that everywhere in the stopband $|H(e^{j\omega})|$ has to be below -60 dB.

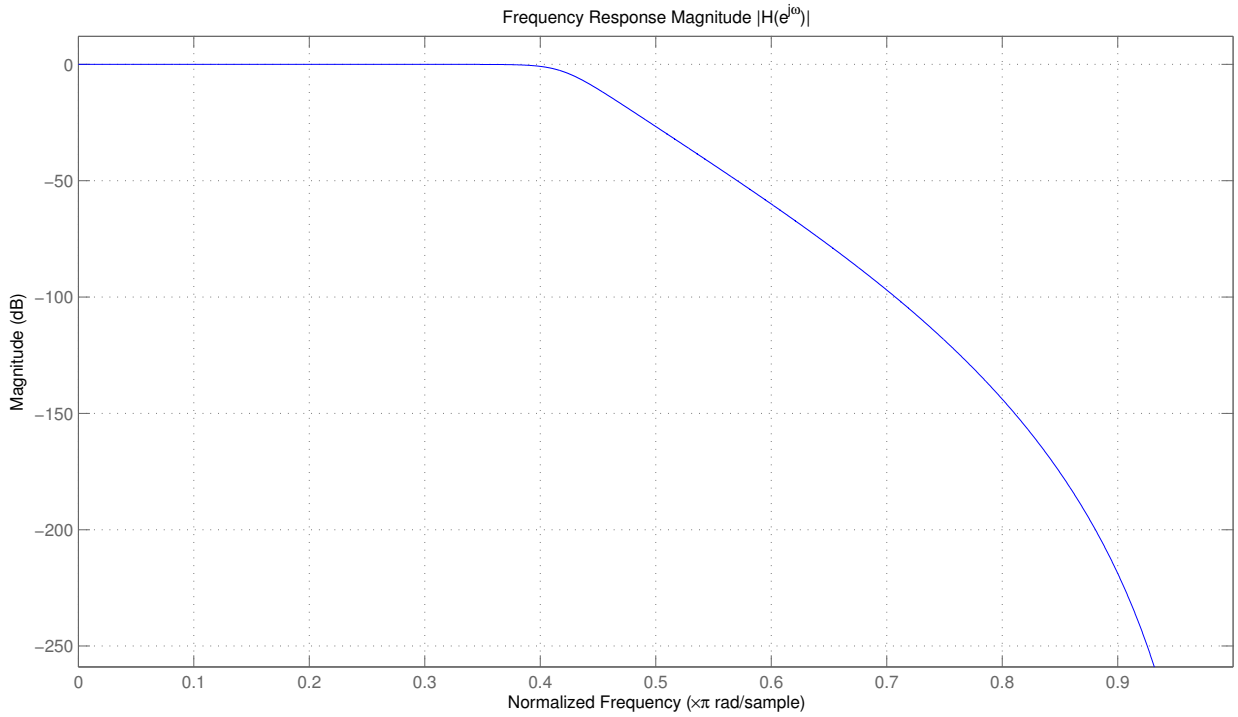


Figure 1: Lowpass digital Butterworth filter frequency response.

The region between `Wp` and `Ws` is called the transition band. For this filter, the transition band goes from 0.4 to 0.6 in units of normalized digital frequency, which is 0.4π to 0.6π rad/sample.

The main features of the digital Butterworth filter are that it is maximally flat in the passband, the passband is monotonic (there is no rippling), and the phase is approximately linear in the passband.

The parameter `Nf` returned by `buttord` gives the filter order. This is the highest power of $e^{-j\omega}$ that appears in the numerator or denominator of the frequency response $H(e^{j\omega})$. It is also the highest power of z^{-1} that appears in $H(z)$. The parameter `Wn` gives the Butterworth natural frequency, which is the point in the transition band where the frequency response magnitude has dropped by 3 dB compared to the passband. For this filter, the order is 12.

The vectors `num` and `den` returned by `butter` contain the coefficients for the numerator and denominator polynomials of $H(e^{j\omega})$, which are the same as the numerator and denominator coefficients of the transfer function $H(z)$.

The Matlab statement to run the filter is `y = filter(num,den,x);` where `x` is the input signal and `y` is the output signal.

You can also design a highpass digital Butterworth filter like this:

```
Wp = 0.6;
```

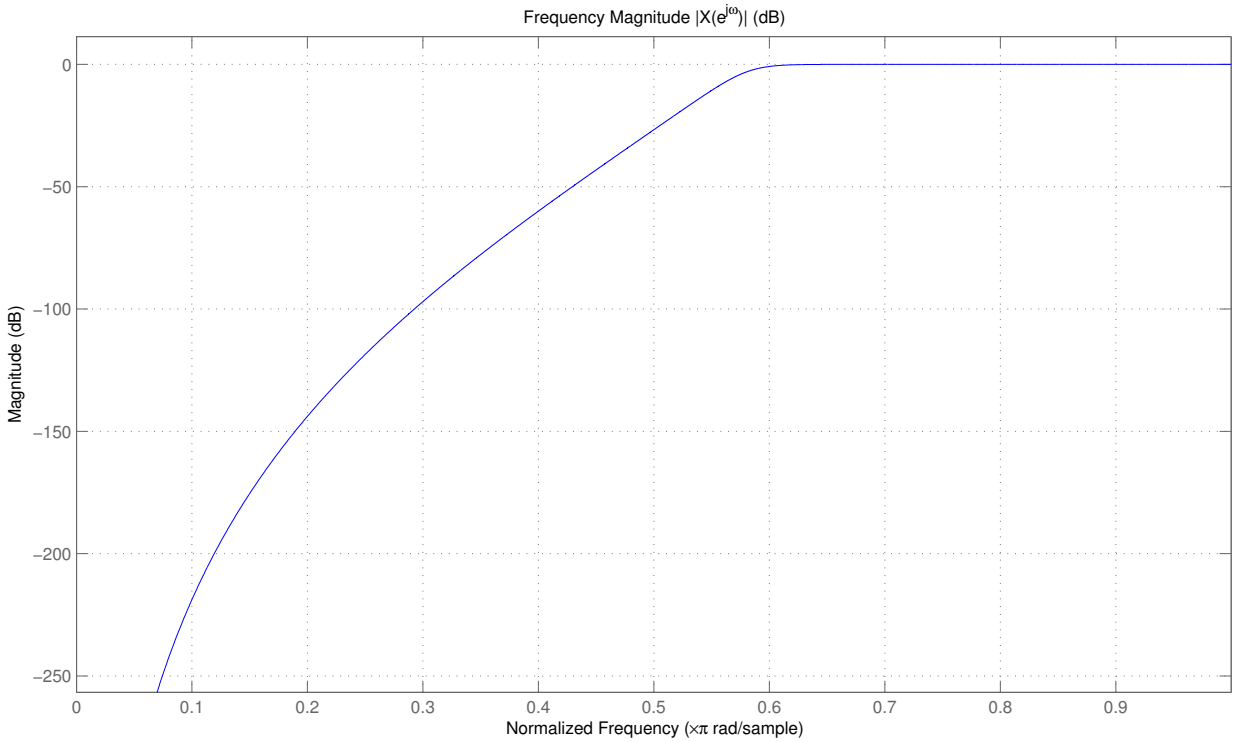


Figure 2: Highpass digital Butterworth filter frequency response.

```
Ws = 0.4;
Rp = 1;
Rs = 60;
[Nf, Wn] = buttord(Wp,Ws,Rp,Rs);
[num,den] = butter(Nf,Wn,'high');
```

Notice that this time `Wp` > `Ws`. That is because the stopband is now on the left starting at DC, followed by the transition band in the middle and the passband on the right. The frequency response magnitude for this filter is shown in Fig. 2.

The parameter `Wp` again specifies the passband edge frequency, which we set to a normalized digital frequency of 0.6. So the passband goes from 0.6π rad/sample to π rad/sample. As before, we set `Rp` to 1, so $|H(e^{j\omega})|$ has to be between -1 dB and +1 dB everywhere in the passband.

We set the stopband edge frequency `Ws` to a normalized digital frequency of 0.4. So the stopband goes from DC to 0.4π rad/sample. We set the minimum stopband attenuation parameter `Rs` to 60, so $|H(e^{j\omega})|$ is below -60 dB everywhere in the stopband.

The transition band lies between the stopband and the passband. For this filter, the transition band goes from 0.4 to 0.6 in normalized digital frequency, which is 0.4π rad/sample to 0.6π rad/sample in radian digital frequency.

For any given filtering problem, we usually want to use the smallest filter order `Nf` that we can. A higher filter order means more delay, more complexity, and increased implementation cost. It also means that the frequency response phase $\arg H(e^{j\omega})$ will be more nonlinear, which is undesirable for digital audio. Here are the factors that increase the filter order:

- for a given passband ripple `Rp` and minimum stopband attenuation `Rs`, making the width of the transition band smaller will increase the order.
- for a given transition bandwidth, decreasing `Rp` or increasing `Rs` will increase the order.

Now, having `Rp` bigger than 1 dB could distort the signal in the filter passband, which is bad. So, for a given filtering problem, we generally want to use the widest transition bandwidth $|\texttt{Ws} - \texttt{Wp}|$ and the smallest stopband attenuation `Rs` that will do the job.

6. Consider the Matlab code below, which does the following:
   - Makes the signal `x1` a 250 Hz pure tone that lasts for 4 sec.
   - Plays `x1` through the sound card.
   - Makes the signal `x2` a swept frequency chirp that goes from 1 kHz to 3 kHz. The details of how I made the chirp signal are not important to you for this assignment. But in case you are interested, here they are. Human hearing perceives the signal $x(t) = \cos[\varphi(t)]$ as a tone with a time-varying frequency $\varphi'(t)$. The quantity $\varphi'(t)$ is called the *instantaneous frequency*. For a chirp, $\varphi'(t)$ changes linearly with time, which means that the instantaneous phase $\varphi(t)$ has to be quadratic in time. A digital audio chirp signal is given by $x[n] = \cos(\varphi[n])$, where the instantaneous phase $\varphi[n]$ is quadratic in n. So I set $\varphi[n] = an^2 + bn + c$. I set the initial phase offset c to zero to get $\varphi[n] = an^2 + bn$ and $\varphi'[n] = 2an + b$. At $n = 0$, I wanted the analog starting frequency of the chirp to be 1 kHz, which is a radian digital frequency of $\omega_1 = 2\pi \times 1000/F_s$. At $n = 0$, this gave me $\varphi'[0] = b = \omega_1$. At $n = N - 1$, I wanted the analog ending frequency of the chirp to be 3 kHz, which is a radian digital frequency of $\omega_2 = 2\pi \times 3000/F_s$. At $n = N - 1$, this gave me $\varphi'[N-1] = 2a(N-1) + \omega_1 = \omega_2$, or $a = \frac{\omega_2 - \omega_1}{2(N-1)}$. So the desired digital chirp signal is given by $x[n] = \cos(\varphi[n])$ where $\varphi[n] = \frac{\omega_2 - \omega_1}{2(N-1)} n^2 + \omega_1 n$.
   - Plays `x2` through the sound card.
   - Makes the signal $\texttt{x3} = \texttt{x1} + \texttt{x2}$.
   - Normalizes `x3` to the range [-1, 1] and plays it through the sound card.
   - Designs a lowpass digital Butterworth filter to process the signal `x3` by keeping the 250 Hz pure tone but filtering out the chirp. I set the passband edge frequency at 250 Hz, which is a radian digital frequency of $2\pi \times 250/F_s$. To get the value of `Wp` for `buttord`, I divided this by π to convert it to normalized digital frequency. I set the stopband edge frequency to the starting frequency of the chirp, which is 1 kHz in analog frequency and $2\pi \times 1000/F_s$ in radian digital frequency. To

get the value of `Ws` for `buttord`, I divided this by π to convert it to normalized digital frequency. These values of `Wp` and `Ws` place the 250 Hz pure tone `x1` in the filter passband and the chirp signal `x2` entirely in the filter stopband. I set the maximum passband ripple `Rp` to 1 dB and the minimum stopband attenuation `Rs` to 60 dB. The lowpass filtered signal is called `y1`.

- Calls the Matlab `fvtool` and `freqz` functions to display the filter frequency response.
- Plays the lowpass filtered signal `y1` through the sound card.
- Designs a highpass digital Butterworth filter to process the signal `x3` by keeping the chirp signal but filtering out the 250 Hz pure tone. I set the stopband edge frequency at 250 Hz, which is a radian digital frequency of $2\pi \times 250/F_s$. For the `buttord` parameter `Ws`, I divided this by π to convert it to normalized digital frequency. This places the pure tone in the filter stopband. I set the passband edge frequency at the starting frequency of the chirp, or 1 kHz, which is a radian digital frequency of $2\pi \times 1000/F_s$. For the `buttord` parameter `Wp`, I divided this by π to convert it to normalized digital frequency. This places the chirp signal entirely in the filter passband. I used 1 dB for the maximum passband ripple `Rp` and 60 dB for the minimum stopband attenuation `Rs`. The highpass filtered signal is called `y2`.
- Adds the highpass filter to `fvtool` and calls `freqz` to plot the frequency response.
- Plays the highpass filtered signal `y2` through the sound card.

```
%-------------------------------------------------------
% P6a
%
% Make some digital audio signals and demonstrate filtering.
% All signals are 4 seconds in duration.
% - Make x1 a 250 Hz pure tone.
% - Play x1 through the sound card.
% - Make x2 a swept frequency chirp from 1 kHz to 3 kHz.
% - Play x2 through the sound card.
% - Make x3 = x1 + x2.
% - Play x3 through the sound card.
% - Apply a lowpass digital Butterworth filter to x3 to
%     keep the pure tone and reject the chirp.
% - Play the filtered signal through the sound card.
% - Apply a highpass digital Butterworth filter to x3 to
%     keep the chirp and reject the pure tone.
% - Play the filtered signal through the sound card.
%
```

```
Fs = 44100;                         % sampling frequency in Hz
N = Fs * 4;                         % length of the 4 sec signal
n = 0:N-1;                          % discrete time variable

% Make x1 a 250 Hz pure tone
f_analog = 250;                     % pure tone analog frequency
w_dig = 2*pi*f_analog/Fs;           % radian digital frequency
x1 = cos(w_dig * n);                % the pure tone
sound(x1,Fs,16);                    % play it through sound card
pause(5);                           % wait for sound card to clear

% Make x2 a chirp.  Sweep analog freq from 1 kHz to 3 kHz
f_start_analog = 1000;
w_start_dig = 2*pi*f_start_analog/Fs;
f_stop_analog = 3000;
w_stop_dig = 2*pi*f_stop_analog/Fs;
phi = (w_stop_dig-w_start_dig)/(2*(N-1))*(n.*n) + w_start_dig*n;
x2 = cos(phi);
sound(x2,Fs,16);                    % play it through sound card
pause(5);                           % wait for sound card to clear

% Add the two signals
x3 = x1 + x2;
x3 = x3 / max(abs(x3));             % normalize the range to [-1,1]
sound(x3,Fs,16);                    % play it through sound card
pause(5);                           % wait for sound card to clear

% Use a lowpass digital Butterworth filter to keep the 250 Hz
%   pure tone and reject the chirp.
Wp = w_dig/pi;                      % normalized passband edge freq
Ws = w_start_dig/pi;                % normalized stopband edge freq
Rp = 1;                             % max passband ripple
Rs = 60;                            % min stopband attenuation
[Nf, Wn] = buttord(Wp,Ws,Rp,Rs);  % design filter order
[num,den] = butter(Nf,Wn);          % design the filter
h=fvtool(num,den);                  % show frequency response
figure(2);
freqz(num,den,1024);                % plot frequency response
title('Lowpass Frequency Response');
y1 = filter(num,den,x3);            % apply the filter
y1 = y1 / max(abs(y1));             % normalize filtered signal
sound(y1,Fs,16);                    % play it through sound card
```

```
pause(5);                          % wait for sound card to clear

% Use a highpass digital Butterworth filter to keep the chirp
%   and reject the 250 Hz pure tone.
Ws = w_dig/pi;                     % normalized stopband edge freq
Wp = w_start_dig/pi;               % normalized passband edge freq
Rp = 1;                            % max passband ripple
Rs = 60;                           % min stopband attenuation
[Nf, Wn] = buttord(Wp,Ws,Rp,Rs); % design filter order
[num2,den2] = butter(Nf,Wn,'high'); % design the filter
Hd = dfilt.df1(num2,den2);         % make filter object
addfilter(h,Hd);                   % add filter 2 to fvtool
figure(3);
freqz(num2,den2,1024);             % plot frequency response
title(' Highpass Frequency Response');
y2 = filter(num2,den2,x3);         % apply the filter
y2 = y2 / max(abs(y2));            % normalize filtered signal
sound(y2,Fs,16);                   % play it through sound card
```

(a) Type in this code and run it. You can type it in line-by-line at the command prompt or you can create an *m*-file. You can download the *m*-file from `canvas.ou.edu`.

(b) Modify the Matlab code to do the following:

- Make `x1` a four second cosine pure tone with analog frequency 1 kHz and play it through the sound card.
- Make `x2` a four second cosine pure tone with analog frequency 3 kHz and play it through the sound card.
- Make `x3` = `x1` + `x2` and play `x3` through the sound card.
- Apply a lowpass digital Butterworth filter to `x3` to keep the 1 kHz pure tone but filter out the 3 kHz pure tone. You can use 1 dB for the maximum passband ripple `Rp` and 60 dB for the minimum stopband attenuation `Rs`. You will need to set the passband edge frequency $\geq$ 1 kHz. Note that this is $2\pi \times 1000/F_s$ in radian digital frequency. Divide that by π to get the minimum value for the normalized digital passband edge frequency `Wp`. You will need to set the stopband edge frequency $\leq$ 3 kHz. This is $2\pi \times 3000/F_s$ in radian digital frequency. Divide that by π to get the maximum value for the normalized digital passband edge frequency `Ws`.
- Play the lowpass filtered signal through the sound card.

7. Get the wave files `noisysig.wav` from Moodle. The file `noisysig.wav` contains a digital audio signal that has been corrupted by additive noise.

If you just want to get it done, you can find the solution Online. But, if you want to learn something from this project, try to do it on your own. Have fun! -SS

In this problem, you will design a digital Butterworth filter to remove the noise.

(a) Use the Matlab `audioread` function to read each signal. Use Matlab to play the noisy signal through the sound card. You can also play the file using any wave-capable media player like *Windows Media Player* or VLC.

(b) Use the Matlab `length` function to find the length of each signal and plot the centered DFT magnitude in dB as a function of normalized digital frequency.

(c) Design a lowpass digital Butterworth filter to remove the noise. You can use 1 dB for the maximum passband ripple `Rp` and 60 dB for the minimum stopband attenuation `Rs`. Determine appropriate normalized digital frequency values for the passband edge frequency `Wp` and stopband edge frequency `Ws` by analyzing the centered DFT magnitude plots. You must design `Wp` and `Ws` so that the filter order `Nf` is **twelve or less**.

(d) Apply your filter to remove the noise. Play the filtered signal through the sound card and use the Matlab `audiowrite` function to save it to a wave file. Name this wave file `filteredsig.wav`. Upload your `filteredsig.wav` file along with your WORD or PDF solution file and matlab script files.