University of Washington – ECE Department
EE235 Lab 5 – Frequency Domain
Background Material

In this lab, we will learn how to transform signals from the time domain to the frequency domain and try to identify the frequencies of $e^{j\omega t}$ that comprise a periodic signal. The concepts we'll focus on are finding the Fourier Series coefficients a_k of x(t), identifying frequency components of x(t), and understanding the relationship between Fourier Series coefficient index k and frequency ω.

Concepts TO REVIEW	Concepts You Will Learn
• Generating a signal in time domain • Playing audio signal • Subplots – usage, labeling, titles • Extracting a list of elements in a vector • Vector concatenation • Loops, functions	• Using fft and fftshift functions • Finding magnitude and phase • A few other useful functions

1.0 Frequency Analysis on a Computer

In this lab, because we are working on a computer, the signal that we work with will be digitized (i.e. discrete time) and finite length. Similarly, the resulting Fourier transforma will also have to be discrete and finite-length. This means that we will actually be using a discrete Fourier transform (DFT), which differs from the continuous-time Fourier transform that you in class in two ways: i) the frequency domain is discrete, and ii) the frequency domain is over the window [-fs/2,fs/2) where fs is the sampling frequency (in Hz). (We could also use radians, but typically the sampling frequency is given in Hz.) For the time domain, we saw that if we have a small enough sampling time, then for practical purposes the signal can be treated as continuous. We will have similar constraints in the frequency domain, which determines the length of the DFT. You'll learn more about this if you take EE341. For now, we'll just give you the sampling time and the DFT size. However, you will need to deal with the fact that the DFT doesn't have perfect frequency resolution.

To compute the DFT, we will use a popular and efficient algorithm called the Fast Fourier Transform (FFT). Specifically, we'll use the implementation in the numpy, with examples below assuming you import numpy as np. The functions that you'll need are

```
xf = np.fft.fft(x,nfft)        # FFT of time signal x nfft frequency samples
xfs = np.fft.fftshift(xf)      # shift FFT of xf to be centered around 0
yt = np.fft.ifft(yf,nt)        # inverse FFT of yf, optional nt specifies length of yt
```

Most implementations of the FFT give you a vector where the frequency samples correspond to the range [0,fs). It turns out that the [fs/2,fs) range corresponds to the negative frequencies, so the fftshift function is for people who want to plot the result showing negative frequencies (centering the frequency plot around 0). After the fftshift, the frequency range is [-fs/2,fs/2)

If your DFT is length N and the range is [0,fs), then the interval between frequency samples corresponds to $\Delta f = fs/N$. So, if you are interested in X(f), then you need to use the index that is the nearest integer to $f/\Delta f$. This is similar to what we did with time signals. Recall that when accessing a sample of x(t) for t that is not an integer multiple of the sampling period Ts, you need to round t/Ts to the nearest integer to access the appropriate value in the time vector.

Since the Fourier Transform is complex-valued, we generally plot it with two plots, usually magnitude and phase. You can find the magnitude and phase using the numpy functions:

```
np.abs(x)      # |x|, either the absolute value if x is real or the magnitude if it is complex
np.angle(x)    # phase of x, for complex x
```

For purposes of this lab, you only need to look at the magnitude.

Ideally **np.fft.ifft(np.fft.fft(a,n))=a,** but numerical limitations of computers could cause this not to be the case. If you are not careful about complex values in processing in the frequency domain, you could end up with a complex signal when you expected a real signal. These issues will be very small in terms of magnitude, but you could have unexpected things in a phase plot.

2.0 Other useful functions

For this lab, you will be implementing signals with sinusoids. Some things that may be useful to you, assuming you import numpy as np:

```
np.pi          # the value of π
np.sin(x)      # sin(x), where x could be a single time or an array of time samples
np.cos(x)      # cos(x), where x could be a single time or an array of time samples
```

For example, if I wanted to create the signal d0 in your lab, I'd use:

```
fs=8000
t=np.arange(0,0.25,1/fs)
d0=np.sin(2*np.pi*941*t)+np.sin(2*np.pi*1336*t)
```

A useful way to finding elements in an array that meet a certain criterion is to create a mask, as in:

```
a = np.array([3, 4, 1, -6, 6])
ind = np.arange(len(a))
b = ind[a[ind] > 3]            # b = [1, 4]
```

This will be useful for finding peaks in the spectrum of a periodic signal.