University of Washington – ECE Department
EE 242 Lab 3: Frequency Domain Representation of Signals
Background Material

In this lab, we will learn how to build periodic signals from component sinusoids and how to transform signals from the time domain to the frequency domain. The concepts we'll focus on include: implementation of the Fourier Series synthesis equation, using a discrete implementation of the Fourier Transform (DFT) with a digitized signal, and understanding the relationship between the discrete DFT index k and frequency ω for both the original continuous signal $x(t)$.

1.0 Fourier Series

For the first part of this lab, you will be synthesizing signals with sinusoids. Some numpy functions that may be useful to you (importing numpy as np), include np.pi, np.sin(x), and np.cos(x). For example, to create a signal x0 that corresponds to a 300Hz tone lasting 250ms, using an 8kHz sampling frequency, use:

```
fs = 8000
t = np.arange(0, 0.25, 1/fs)
x0 = np.cos(2*np.pi*300*t)
```

In this case, we have only a single sinusoid, so we would not hear any difference if we changed the phase. When you start adding up multiple sinusoids, the phase will matter.

In class, you learned that a periodic signal can be expressed as a sum of complex exponentials:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{jk\omega_0 t} \quad \text{where} \quad a_k = \frac{1}{T}\int_{<T_0>} x(t)e^{-jk\omega_0 t} \tag{1}$$

When the signal is real-valued, then we know that $a_{-k} = a_k^*$, in which case we can write

$$\begin{aligned}
x(t) &= a_0 + \sum_{k=1}^{\infty}(a_k e^{jk\omega_0 t} + a_{-k}e^{-jk\omega_0 t}) \\
&= a_0 + \sum_{k=1}^{\infty}(|a_k|e^{jk\omega_0 t + j\angle a_k} + |a_{-k}|e^{-jk\omega_0 t + j\angle a_{-k}}) \\
&= a_0 + \sum_{k=1}^{\infty}|a_k|(e^{j(k\omega_0 t + \angle a_k)} + e^{-j(k\omega_0 t + j\angle a_k)}) \\
&= a_0 + \sum_{k=1}^{\infty} 2|a_k|\cos(k\omega_0 t + \angle a_k)
\end{aligned}$$

To consider a specific example, if you wanted to build a square wave with period $T_0 = 40ms$ and Fourier Series coefficients $a_0 = 0.5$ and

$$a_k = \frac{1}{k\pi}\sin(k\pi/2)e^{-jk\pi/2}$$

using sinusoids, the implementation (taking advantage of the time shift property of Fourier series) would be:

$$x(t) = 0.5 + \sum_{k=1}^{N} \frac{2}{k\pi}\sin(k\pi/2)\cos(k(\frac{2\pi}{.04})(t - .01))$$

Of course, a true square wave would require an infinite sum, but you cannot implement that in practice, so you would choose some N that gave you a good approximation and was within your compute budget.

2.0 Frequency Domain 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 $[-\frac{f_s}{2}, \frac{f_s}{2})$ where f_s 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. The length of the time window also impacts the results. You'll learn more about this is you take EE 342. For now, we'll just give you the window, sampling time and the DFT size. However, you will still see artifacts in the results due to the fact that the DFT gives an approximation of the true frequency content.

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

The numpy fft function will work with any value of nfft, but the efficient algorithms are based on using a value of nfft that is a power of 2. If nfft is longer than the signal, the FFT will simply add zero-valued time signals at the end. Since a longer signal window have more energy, you need to scale the FFT result by the time-signal window length in order to relate the FFT result to the original signal.

Most implementations of the FFT give you a vector where the frequency samples correspond to the range $[0, f_s)$. It turns out that the $[\frac{f_s}{2}, f_s)$ 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 $[-\frac{f_s}{2}, \frac{f_s}{2})$.

If your DFT is nfft=N samples in length and the range is $[0, f_s)$, then the interval between frequency samples (the frequency resolution) corresponds to $\Delta f = \frac{f_s}{N}$. So, if you want to plot the frequency content, you need to scale the FFT index accordingly, e.g. the k-th element of the vector corresponds to frequency kf_s/N. If you are interested in $X(f)$ at a particular frequency f, then you need to use the index that is the nearest integer to $\frac{f}{\Delta f}$. 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 T_s, you need to round $\frac{t}{T_s}$ to the nearest integer to access the appropriate value in the time vector.

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

- np.abs(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

The magnitude is important for understanding the frequency content, so sometimes only magnitude is plotted. Because a phase of π is the same as $-\pi$, phase is sometimes plotted only in the $(-\pi, pi]$ range. Because our hearing is sensitive to energy at different frequencies on a log scale, magnitude is usually plotted on a log scale.

While not relevant to this lab, it is important to recognize that there are other issues to consider. For example, 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.