Lesson 06: Filter Analysis

Now that we're comfortable plotting, it's time we use our newfound skill for something of high importance for Electrical Engineers: filter analysis. MATLAB offers a powerful Signal Processing Toolbox[1] that allows us to perform this analysis with relative ease.

The Fast Fourier Transform

Since MATLAB exists in the *real world*, we need to craft a Fourier Transform that can exists in the confines of our measly computer. We define the multidimensional ***Discrete Fourier Transform (DFT)*** with the following:

$$X[\boldsymbol{k}] = \sum_{\boldsymbol{m}=0}^{n-1} x[\boldsymbol{m}]\, \mathrm{e}^{-\mathrm{j}2\pi \boldsymbol{k}\cdot\boldsymbol{m}/n} \tag{1}$$

The equation above allows us to compute a n-point DFT of a d-dimensional signal. The value of n allows us to vary our resolution in the frequency domain with so-called ***bin frequencies*** with the following ***bin spacing***:

$$\Delta\omega = \frac{2\pi}{n} \tag{2}$$

Aside— Increasing n results in a higher resolution in the frequency domain at the cost of computational time.

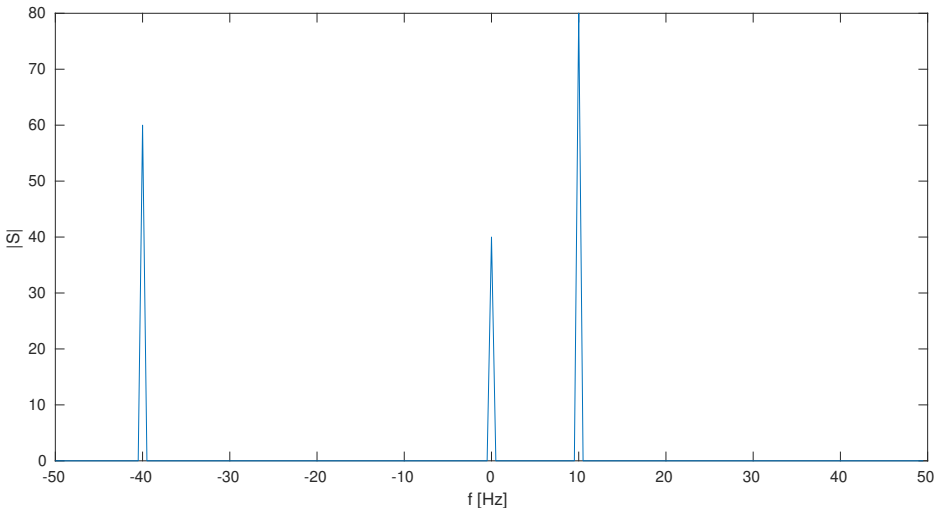
The ***Fast Fourier Transform (FFT)*** is nothing more than an *optimization* of the DFT, going from $O(n^2)$ to $O(n \log n)$ by removing redundant computations.[2] Today, almost everyone implements the DFT as an FFT, and as such, you'll almost always incorrectly hear the DFT referred to as the FFT!

In MATLAB, we can compute the FFT with the `fft()` function:

[1] `https://www.mathworks.com/products/signal.html`

[2] Veritasium made a great video on the history behind the FFT that I highly recommend watching: `https://www.youtube.com/watch?v=nmgFG7PUHfo`


```
fs = 100;

T = 1 / fs;
l = 200;
t = (0:l - 1) * T;
s = 0.2;
s = s + 0.4 * exp(1j * 2 * pi * 10 * t);
s = s + 0.3 * exp(-1j * 2 * pi * 40 * t);
S = fftshift(fft(s));

f = figure;
plot(fs / l * (-l / 2:l / 2 - 1), S);
ylabel('|S|');
xlabel('f [Hz]');
```



Due to how the DFT is defined, the output of `fft()` will be DC, followed by positive and negative frequencies. As such, it's typically desirable to center our output around DC with `fftshift()`. Other things to note are that the output may be complex-valued so we typically plot the magnitude of the FFT and that the frequency range is from $-f_s/2$ to $f_s/2$.

The Z-Transform

We define ***the Z-Transform*** as follows:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{3}$$

The Z-Transform allows us to convert a discrete-time signal into a complex-valued z-domain (or z-plane) representation. This is desirable as the transform domain gives us some nice properties:

- **Linearity:** $c_1x_1 + c_2x_2 \leftrightarrow c_1X_1 + c_2X_2$
- **Delay:** $x[n-1] \leftrightarrow z^{-1}X(z)$
- **Convolution:** $h * x \leftrightarrow H \cdot X$

The transfer domain also allows us to easily analyze the behavior of systems in the form of polynomial transfer functions:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \cdots + b_mz^{-m}}{a_0 + a_1z^{-1} + \cdots + a_nz^{-n}} \tag{4}$$

Although we can represent a polynomial transfer function in terms of its coefficients, this is undesirable due to limited machine precision; rather, we far prefer to keep transfer functions in pole-zero-gain form:

Aside— This isn't purely a preference thing. Analysis should *never* be done in polynomial form until the last possible instance due to the introduced floating-point quantization error!

$$H(z) = \frac{Z(z)}{P(z)} = K\frac{(z-z_1)^{n_1}\ldots(z-z_N)^{n_N}}{(z-p_1)^{m_1}\ldots(z-p_M)^{m_M}} \tag{5}$$

where K is the gain of the system, the numerator contains the zeros of the system, and the denominator contains the poles of the system.

Conversion Between Polynomial and Pole-Zero-Gain Form

In MATLAB, we store the polynomial coefficients as **row vectors** and the poles, zeros, and gains of a system as **column vectors**. It is important to remember this as the behavior of functions in the Signals Processing Toolbox depend on this semantic difference!

Say we wanted to represent the following pole-zero-gain transfer function:

$$H(z) = \frac{(z+3)(z-1)^2}{z(z-3)} \tag{6}$$

We can represent it in MATLAB like so:

```
z = [-3; 1; 1];
p = [0; 3];
k = 1;
```

Then to convert this to a polynomial transfer function, we can call `zp2tf()`:

```
[b, a] = zp2tf(z, p, k)
```

```
b =

     1     1    -5     3


a =

     1    -3     0
```

Conversely, we can represent the following polynomial transfer function:

$$H(z) = \frac{1 + z^{-1} - 5z^{-2} + 3z^{-3}}{1 - 3z^{-1}} \tag{7}$$

Like so in MATLAB:

```
b = [1, 1, -5, 3];
a = [1, -3];
```


Then to convert this to a pole-zero-gain transfer function, we can call `tf2zpk()`:

```
[z, p, k] = tf2zpk(b, a)
```

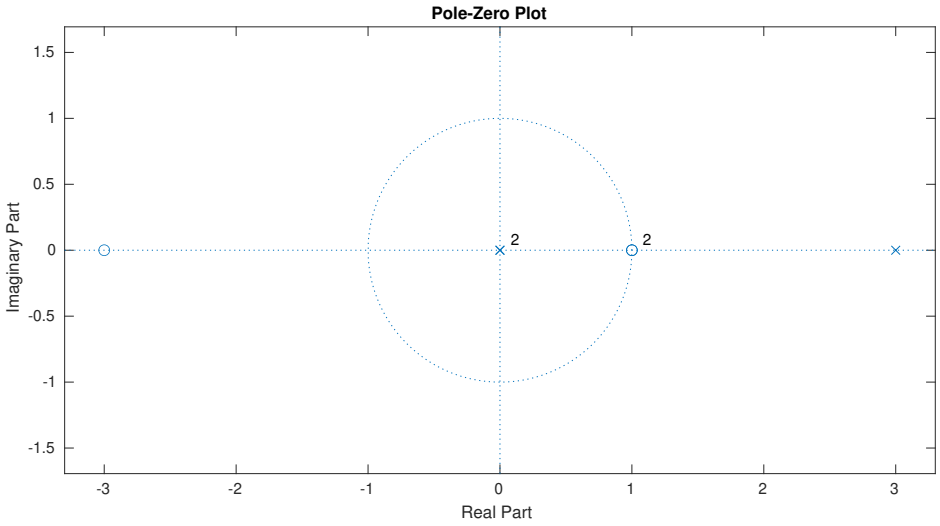
```
z =

  -3.0000 + 0.0000i
   1.0000 + 0.0000i
   1.0000 - 0.0000i


p =

     0
     0
     3


k =

     1
```

Aside— Be careful when using conversion functions, as some expect positive power coefficients while others expect negative power coefficients!

Pole-Zero Plot

To plot our transfer function, we can utilize `zplane()`:

```
1 zplane(z, p);
```



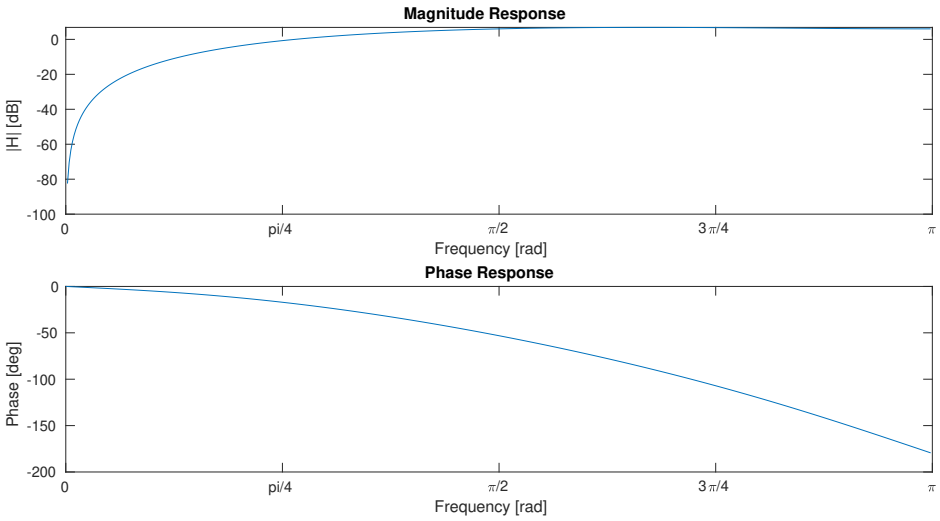
Keep in mind that we can also call `zplane()` with the `b` and `a` vectors.

Frequency Response

To get the frequency response of a digital filter, we can call `freqz()`. Although this function can plot the frequency response of a digital filter, we can do better as we can tune the plots to our needs.

Note that when plotting the phase response of our filter, we use `unwrap()`, which will resolve jump discontinuities in our phase angle. The following also covers the very basic functionality of `freqz()`, as the online documentation offers great examples for different use cases.


```m
[H, w] = freqz(b, a);

H_dB = 20 * log10(abs(H));
H_ph = rad2deg(unwrap(angle(H)));

f = figure;

subplot(2, 1, 1);
plot(w, H_dB);
title('Magnitude Response');
xlim([0, pi]);
xticks([0, pi / 4, pi / 2, 3 * pi / 4, pi]);
xticklabels({'0', 'pi/4', '\pi/2', '3\pi/4', '\pi'});
xlabel('Frequency [rad]');
ylabel('|H| [dB]');

subplot(2, 1, 2);
plot(w, H_ph);
title('Phase Response');
xlim([0, pi]);
xticks([0, pi / 4, pi / 2, 3 * pi / 4, pi]);
xticklabels({'0', 'pi/4', '\pi/2', '3\pi/4', '\pi'});
xlabel('Frequency [rad]');
ylabel('Phase [deg]');
```



The Laplace Transform

Since we're mostly concerned with ***Digital Signal Processing (DSP)***, we won't get much in-depth about analog filters as the analysis is close to that of digital filters: notably we use `freqs()` over `freqz()` and there is no `splane()`, although that has never stopped us!

```
function splane(z, p)
    plot(real(z), imag(z), 'o', real(p), imag(p), 'x');
    title('Pole-Zero Plot');
    xlabel('Real Part');
    ylabel('Imaginary Part');
    axis('equal');
    line([0, 0], ylim(), 'LineStyle', ':');
    line(xlim(), [0, 0], 'LineStyle', ':');
end
```

