ECE 450 - Homework #9

Collin Heist

October 24, 2019

1 ECE 450 - Homework #9

1.0.1 Package Imports

```
In [1]: import numpy as np
        import seaborn as sns
        import pandas as pd
        import matplotlib.pyplot as plt
        from scipy import signal as sig
        from control import margin, tf
        import warnings
        warnings.filterwarnings('ignore')
```

1.0.2 Generic function to generate the $H(s)$ for a Buttersworth Filter of a given n poles

```
In [2]: def buttersworth_tf(order):
            pole_list = [np.sin((np.pi * (2 * k - 1))/(2 * order)) + complex(0, np.cos((np.pi *
            s_pole_list = [[1, pole] for pole in pole_list]

            return convolve_all(s_pole_list)
```

1.0.3 Generic function to get the minimum n for a desired cutoff

```
In [3]: def minimum_n(deviation, omega_p):
            return int(np.ceil(np.log10(1 / deviation ** 2 - 1) / (2 * np.log10(omega_p))))
```

1.0.4 Generic function to convolve any number of equations

```
In [4]: def convolve_all(values):
            temp_conv = values[0]
            if len(values) > 1:
                for next_val in values[1:]:
                    temp_conv = np.convolve(temp_conv, next_val)

            return temp_conv
```

1.0.5 Generic function to generate a lowpass, highpass, or bandpass Buttersworth filter

In [31]:
```
def lowpass_buttersworth(cutoff_freq=1, order=None, passband_deviation=None, passband_f
    # Move to cutoff frequency of 1 if necessary
    passband_freq = passband_freq if passband_freq is None else passband_freq / cutoff_

    order = order if order is not None else minimum_n(1 - passband_deviation, passband_

    # Generate the Buttersworth Transfer Function
    num = [1]
    den = buttersworth_tf(order)

    # Shift back to the given cutoff frequency
    num = np.multiply(cutoff_freq ** order, num)
    den = [term * (cutoff_freq ** t_order) for t_order, term in enumerate(den)]

    return num, den

def highpass_buttersworth(cutoff_freq=1, order=None, passband_deviation=None, passband_
    # Move to cutoff frequency of 1 if necessary
    passband_freq = passband_freq if passband_freq is None else cutoff_freq / passband_

    order = order if order is not None else minimum_n(passband_deviation, passband_freq

    # Generate the Buttersworth Transfer Function
    num = np.zeros(order + 1)   # Make the s^order term 1 to move to a highpass filter
    num[0] = 1
    den = buttersworth_tf(order)

    # Shift back to the given cutoff frequency
    den = [term * (cutoff_freq ** t_order) for t_order, term in enumerate(den)]

    return num, den

def bandpass_buttersworth(center_freq=1, bandwidth=1, order=2):
    assert order % 2 == 0, "The order of the bandpass filter must be even."

    num = [1]
    den = buttersworth_tf(int(order / 2))

    # Shift back to the given bandwidth frequency
    num = np.multiply(bandwidth ** int(order / 2), num)
    den = [term * (bandwidth ** t_order) for t_order, term in enumerate(den)]

    # Transform up to the center frequency
    temp_den = np.zeros(len(den) + int(order / 2) + 1)
    for place, den_constant in enumerate(den):
        # The order of the applied (s^2 + val^2)
```

```
            ord2 = len(den) - place - 1

            # List of the applied shift for convolving
            stuff = [[1, 0, center_freq ** 2]] * ord2 if ord2 != 0 else [[1]]
            prod = np.multiply(den_constant, convolve_all(stuff))
            for _ in range(place): # Apply the multiplication of s^(n/2)
                prod = np.append(prod, 0)

            # Cumulatively calculate the new denominator
            temp_den = np.add(temp_den, np.pad(prod, (len(temp_den) - len(prod), 0), 'const

        # Multiply the numerator by s^(n/2)
        num = np.pad(num, (0, int(order / 2)), 'constant')

        return num, temp_den
```

1.0.6 Generic function to solve a set of state matrices

```
In [6]: def state_solver(A_matrix, B_matrix, force_function, initial_conditions,
                         time_range=[0, 10], dt=0.01):
            time_values = np.arange(time_range[0], time_range[1], dt)
            x_vals = np.array(initial_conditions)
            state_variables = [[] for _ in range(len(initial_conditions))]

            # Loop through each instance in time, calculate the state variable at that time
            for time in time_values:
                if isinstance(force_function(time), np.ndarray):
                    x_vals = x_vals + dt * (A_matrix @ x_vals) + dt * (B_matrix @ force_function
                else:
                    x_vals = x_vals + (dt * (A_matrix @ x_vals)) + dt * (B_matrix * force_functi
                for index, _ in enumerate(state_variables):
                    state_variables[index].append(x_vals[index][0])

            return state_variables, time_values
```

1.0.7 Generic function to plot the responses of a system

```
In [7]: # Color list for multiple lines on each subplot
        colors = ["red", "blue", "green", "gray", "purple", "orange"]
        step_size = 0.005

        # Generic Function to create a plot
        def create_plot(x, y, xLabel=["X-Values"], yLabel=["Y-Values"],
                        title=[("Plot", )], num_rows=1, size=(18, 14), logx=False):
            plt.figure(figsize=size, dpi=300)
            for c, (x_vals, y_vals, x_labels, y_labels, titles) in enumerate(zip(x, y, xLabel, y
                for c2, (y_v, t) in enumerate(zip(y_vals, titles)):
                    plt.subplot(num_rows, 1, c + 1)
```

```
            # Add a plot to the subplot, use transparency so they can both be seen
            plt.plot(x_vals, y_v, label=t, color=colors[c2], alpha=0.70)
            plt.ylabel(y_labels)
            plt.xlabel(x_labels)
            plt.grid(True)
            plt.legend(loc='lower right')
            if logx:
                plt.xscale("log")

    plt.show()
```

1.0.8 Generic function to generate the magnitude and phase of $H(j\omega)$ values

```
In [8]: def magnitude_phase_response(num, den, omega_range, omega_step=10, gain_num=None, gain_d
            if isinstance(gain_num, (np.ndarray, list)) and isinstance(gain_den, (np.ndarray, li
                num = convolve_all([num, gain_num])
                den = convolve_all([den, gain_den])

            system = sig.lti(num, den)
            w, h_mag, h_phase = sig.bode(system, np.arange(omega_range[0], omega_range[1], omega
            _, phase_margin, _, crossover_w = margin(h_mag, h_phase, w)

            return w, h_mag, h_phase, phase_margin, crossover_w
```

1.1 Problem 8.2.1

```
In [10]: num, den = lowpass_buttersworth(10 ** 3, passband_deviation = 0.15, passband_freq = 850
         print ("H(s) numerator: ", num, "\nH(s) denominator: ", np.real(den))
```

```
H(s) numerator:  [1000000000]
H(s) denominator:  [1.e+00 2.e+03 2.e+06 1.e+09]
```

The transfer function is thus:

$$H(s) = \frac{1000000000}{s^2 + 2 \cdot 10^3 s + 10^9}$$

1.2 Problem 8.2.2

```
In [11]: num, den = highpass_buttersworth(50 * 10 ** 3, order=2)
         print ("H(s) numerator: ", num, "\nH(s) denominator: ", np.real(den))
```

```
H(s) numerator:  [1. 0. 0.]
H(s) denominator:  [1.00000000e+00 7.07106781e+04 2.50000000e+09]
```

The transfer function is thus:

$$H(s) = \frac{s^2}{s^2 + 7.07 \cdot 10^4 s + 2.5 \cdot 10^9}$$

1.3 Problem 8.2.3

```
In [105]: num, den = bandpass_buttersworth(10 ** 3, 300, 4)
          print ("H(s) numerator: ", num, "\nH(s) denominator: ", np.real(den))
```

```
H(s) numerator:  [90000     0     0]
H(s) denominator:  [0.00000000e+00 1.00000000e+00 4.24264069e+02 2.09000000e+06
 4.24264069e+08 1.00000000e+12]
```

The transfer function is thus:

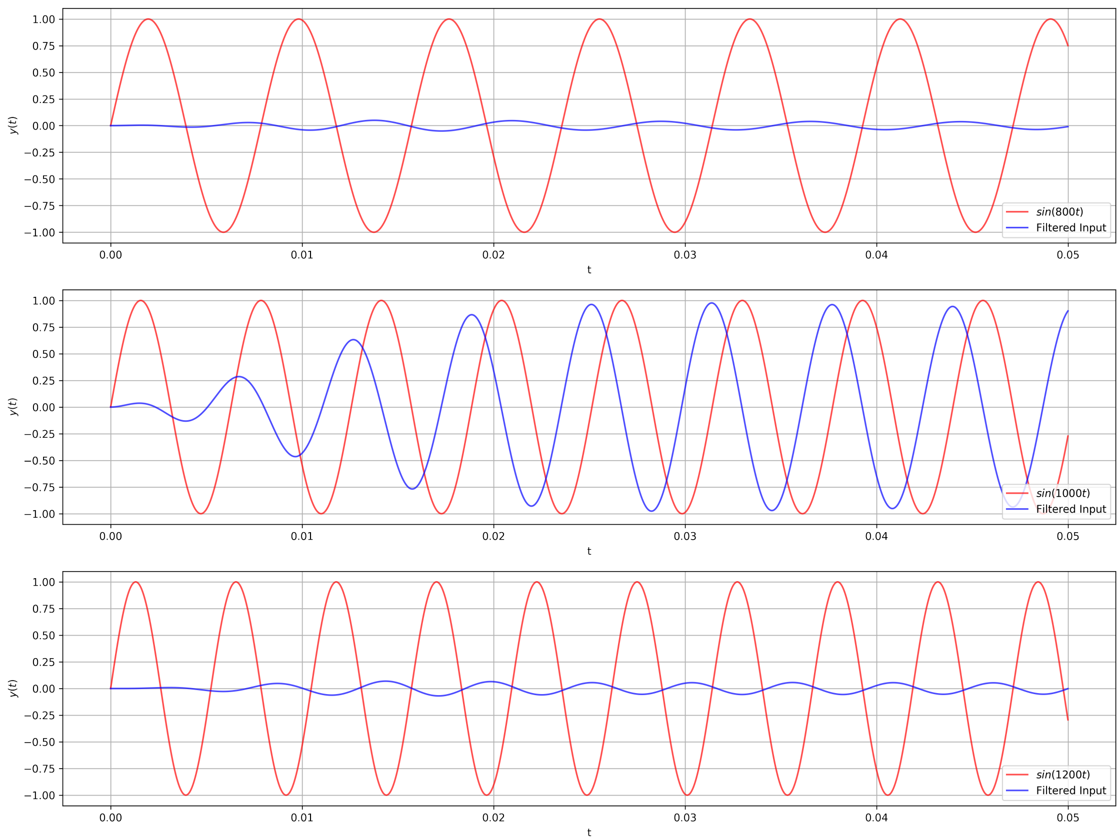
$$H(s)\frac{90000s^2}{s^4 + 4.24 \cdot 10^2 s^3 + 2.09 \cdot 10^6 s^2 + 4.24 \cdot 10^8 s + 10^{12}}$$

1.4 Problem 8.2.4

```
In [93]: input1 = lambda t: np.sin(800 * t)
         input2 = lambda t: np.sin(1000 * t)
         input3 = lambda t: np.sin(1200 * t)

         a, b, c, d = sig.tf2ss(num, den)
         name, t_range, dt = "Filtered Input", [0, 0.05], 0.00001
         vars0950, t = state_solver(a, b, input1, [0, 0, 0, 0], t_range, dt)
         vars0707, _ = state_solver(a, b, input2, [0, 0, 0, 0], t_range, dt)
         vars0100, _ = state_solver(a, b, input3, [0, 0, 0, 0], t_range, dt)

         create_plot([t, t, t],
                     [(input1(t), np.multiply(10, vars0950[0])),
                      (input2(t), np.multiply(80, vars0707[0])),
                      (input3(t), np.multiply(10000000, vars0100[2]))],
                     ["t", "t", "t"], ["$y(t)$", "$y(t)$", "$y(t)$"],
                     [("$sin(800t)$", name), ("$sin(1000t)$", name), ("$sin(1200t)$", name)], 3)
```



Clearly, the non $\omega = 1000$ signals are attenuated greatly.

1.5 Problem 8.2.5

I will choose an arbitrary center frequency of 120. This permits me to recalculate the passband and stopband frequencies.

$$\omega_p = \frac{120}{90} = 1.3$$

$$\omega_s = \frac{120}{220} = 0.54$$

The minimum n required for these are:

$$n = 1$$

```
In [104]: num, den = lowpass_buttersworth(120, 1, 0.2, 90)
          print ("H(s) numerator: ", num, "\nH(s) denominator: ", np.real(den))

H(s) numerator:  [120]
H(s) denominator:  [   1. 120.]
```

Resulting in the transfer function of:

$$H(s) = \frac{120}{s + 120}$$