ECE 351

Signals and Systems

Lab 6

Partial Fraction Expansion

Submitted By :
Zachary Pfaff
https://github.com/ZachPfaff

Contents

Partial Fraction Expansion

Zachary Pfaff

February 15th, 2022

1 Introduction

When working with Laplace transforms in ECE 350, its common to use partial fraction expansion to find the inverse Laplace transform of a function we are working on. Partial fraction expansion is a very useful tool in signals and systems, however it can be tedious to calculate. For this reason, the goal of the following lab is to utilize python code to find the partial fraction expansion of various different functions.

2 Equations

Prior to working with any python code, the step response of the following function needed to be determined by hand.

$$y''(t) + 10y'(t) + 24y(t) = x''(t) + 6x'(t) + 12x(t)$$

Through the use of Laplace transforms, the function was represented by the following partial fraction expansion equation.

$$F(s) = \frac{s^2 + 6s + 12}{s(s+6)(s+4)} = \frac{A}{s} + \frac{B}{s+6} + \frac{C}{s+4}$$

$$A = \frac{1}{2}$$
$$B = 1$$
$$C = -\frac{1}{2}$$

After using partial fraction expansion, the equation was derived into the following using inverse Laplace transforms.

$$f(t) = [\frac{1}{2} + e^{-6t} - \frac{1}{2}e^{-4t}]u(t)$$

It is the goal of this lab to plot this signal by using both the hand calculated equation and the scipy.signal.step() command, then to use scipy.signal.residue() to list out the partial fraction expansion of the signal.

3 Methodology

The first task for part 1 of the lab was to plot the hand calculated response from 0 to 2 seconds. To make sure my hand calculated response was correct, I used scipy.signal.step() to re-create the same graph. The partial fraction expansion was

then determined by using scipy.signal.residue. The following code was implemented into Python for part 1 of the lab, take note that the denominator for the residue command has a wider matrix than the denominator for the step command.

```
#PART 1
y = ((1/2)+np.exp(-6*t) - (1/2)*np.exp(-4*t)) * step(t)

num1 = [1, 6, 12]
den1 = [1, 10, 24]
den2 = [1, 10, 24, 0]

#step response
yout, xout = sig.step((num1, den1), T = t)

#partial fraction expansion
r1, s1, p1 = sig.residue(num1, den2)

print(r1)
print(s1)

#Part 1 graphs
plt.figure(figsize =(12 ,8))
plt.subplot (1,1,1)
plt.plot(t, y)
plt.title('hand calculated h(t)')
plt.ylabel('y')
plt.grid(True)

plt.figure(figsize =(12 ,8))
plt.subplot (1,1,1)
plt.plot(yout, xout)
plt.title('Step() Function')
plt.ylabel('y')
plt.grid(True)
```

Listing 1: Part 1 Code

The code for the next part of lab follows a similar process, except this time the partial fraction expansion is determined from a different function using scipy.signal.residue() from time 0 to 4.5 seconds. The following equation below is to be evaluated through partial fraction expansion, then plotted using the cosine method.

$$y^5(t) + 18y^4(t) + 218y^3(t) + 2036y^2(t) + 9085y^1(t) + 25250y(t) = 25250x(t)$$

After the equation is evaluated through the cosine method, it will be evaluated using the scipy.signal.step() command as used in part 1 to see if the two plots come out to be the same. The code for part 2 can be seen below.

```
#PART 2
t2 = np.arange(0, 4.5 + steps, steps)

num2 = [25250]
den3 = [1, 18, 218, 2036, 9085, 25250, 0]
den4 = [1, 18, 218, 2036, 9085, 25250]

r2, s2, p2 = sig.residue(num2, den3)
print(r2)
print(s2)

def cosine(r2, s2, t2):
    y = np.zeros((len(t2)))
    for i in range(len(r2)):
            y += (2 * abs(r2[i]) * np.exp(np.real(s2[i]) * t2)
                  * np.cos(np.imag(s2[i]) * t2 + np.angle(r2[i])))
   * step(t2)

    return y

y2 = cosine(r2, s2, t2)

yout2, xout2 = sig.step((num2, den4), T = t)

#Part 2 graphs
plt.figure(figsize = (12 ,8))
plt.subplot (1,1,1)
plt.plot(t2, y2)
plt.title('Cosine Method')
plt.ylabel('f1')
plt.grid(True)

plt.figure(figsize =(12 ,8))
plt.subplot (1,1,1)
plt.plot(yout2, xout2)
plt.title('Step() Function')
plt.ylabel('y')
plt.grid(True)
```

Listing 2: Part 2 Code

4 Results

For part 1 of the lab, the partial fraction expansion from the original signal function was printed to perfectly match the hand calculated partial fraction expansion. The results for this are seen below, note that the numerator polynomial coefficients from the printed matrix match variables A, B, and C in the hand calculated section.

```
In [71]: runfile
[ 0.5 -0.5  1. ]
[ 0. -4. -6.]
```

Figure 1: Results from scipy.signal.residue() command

The following plots below show plots from the step response coded using the hand calculated method and the scipy.signal.step() command method. Both plots are from part 1 of the lab.

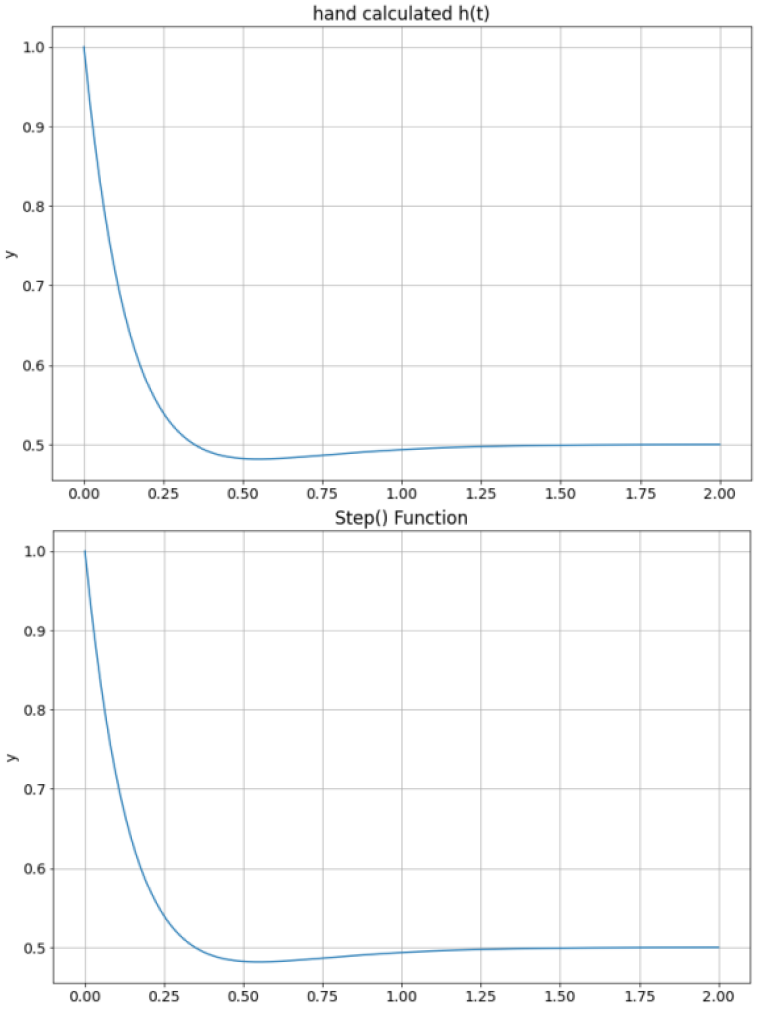

Figure 2: Part 1 Graphs

For part 2, the printed partial fraction expansion came out to be the following. No-

tice that there are many more terms than in part 1 and the terms are complex. For this reason, we either needed to use the cosine method (which we did) or the sine method to evaluate the function.

```
[ 1.          +0.j          -0.48557692+0.72836538j -0.48557692-0.72836538j
 -0.21461963+0.j           0.09288674-0.04765193j  0.09288674+0.04765193j]
[  0. +0.j  -3. +4.j  -3. -4.j -10. +0.j  -1.+10.j  -1.-10.j]
```

Figure 3: Part 2 Partial Fraction Expansion Terms

The following two plots below show the time domain response using the cosine method and the scipy.signal.step() command method.

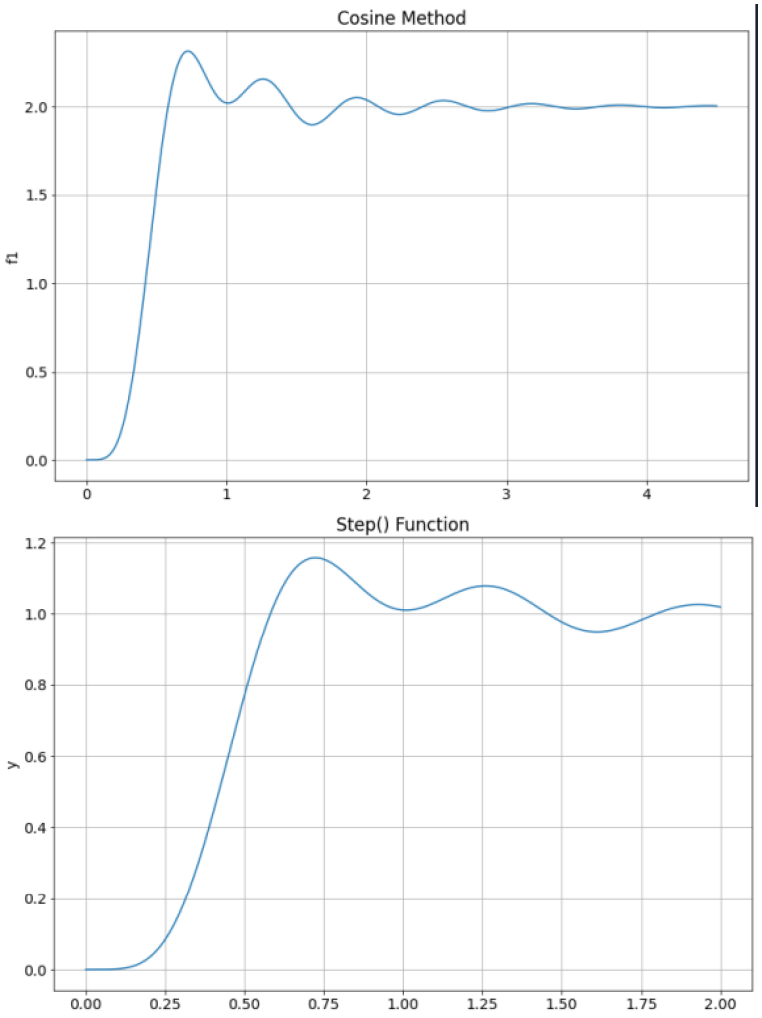

Figure 4: Part 1 Graphs

5 Conclusion

By the end of lab 6 I felt I was able to implement partial fraction expansion into python code, that being said though I ran into a few problems with the lab. First of all, the plots for part two do not look the same although they are identical. The maximum y value for the cosine method is higher than that of the step function. Through testing, I found that I could completely replicate the step function plot with the cosine function by dividing my cosine function equation by 2 (or by removing the 2 at the beginning of the cosine function), however, this goes against Dr. Sullivan's definition of the cosine function so I chose to keep it in the equation.

The next problem I had was getting my residue commands to output the proper partial fraction expansion matrices. Through trial and error I found that I needed to expand the denominator by a dimension, the reason for this being however was unclear to me. Many of the online resources I looked at for this information did not go into specifics on why this is the case. That being said, I was still able to re-create the proper time domain functions.

6 Questions

1. For a non-complex pole-residue term, you can still use the cosine method, explain why this works.

The cosine method will work for terms that are not complex because the complex values will just be evaluated at 0. When this is the case, the cosine function will come out to be 1 since the $\cos(0) = 1$.

$$2ke^{-at}cos(0t + 0)]u(t) = [2ke^{-at}]u(t)$$

2. Leave any feedback on the clarity of the expectations, instructions, and deliverables.

I personally had some trouble with understanding the residue command, even with online sources it was a little confusing for me to grasp. Other than that, I didn't have any other problems following lab instructions.