ECE 351 - Lab 6

Khoi Nguyen
https://github.com/3khoin

14 October 2021

Contents

1 Introduction

The goal of this lab was to use scipy.signal.residue() to perform partial fraction expansion. The methods implemented in this lab allowed for partial fraction expansion of functions that would have been otherwise cumbersome to calculate by hand.

2 Equations

Prelab

$$H(s) = \frac{Y(s)}{X(s)} = \frac{s^2 + 6s + 12}{s^2 + 10s + 24}$$

$$y_{step}(t) = \frac{1}{2}(1 - e^{-4t} + 2e^{-6t})u(t)$$

Part 2

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

Cosine method

$$y_c(t) = 2|k|e^{\alpha t}cos(\omega t + \angle k)u(t)$$

3 Methodology

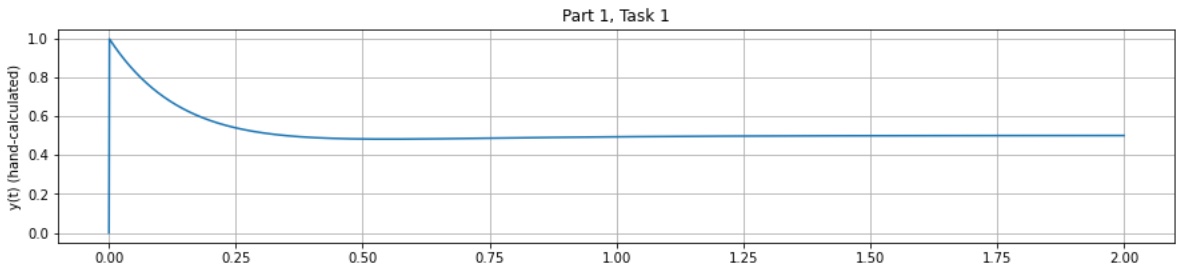
We first set up user-defined functions for the unit step function as well as the step response y(t). We plotted y(t) on the interval 0 $\leq$ t $\leq$ 2 s, then used scipy.signal.step() and H(s) to plot and verify the step response. Then, using Y(s) = H(s)X(s), and entering the appropriate Y(s) values into scipy.signal.residue(), we found the partial fraction expansion values R and P and compared them to those calculated by hand in the prelab.

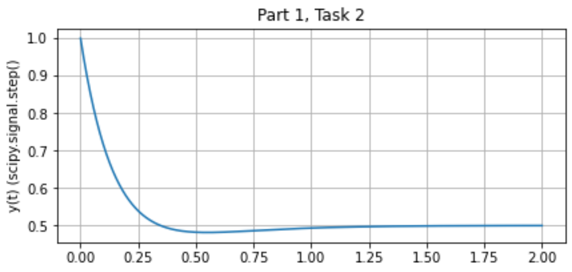
Next, we found R and P for the Part 2 equation by using scipy.signal.residue() as well. We plotted the time-domain response for the Part 2 equation on the interval 0 $\leq$ t $\leq$ 4.5 s, the resulting plot coming from our implementation of the cosine method. We verified the previous plot by plotting the resulting transfer function from the Part 2 equation using scipy.signal.step().

4 Results

4.1 Part 1

Plotted below are the two plots of the step response y(t) from the prelab. The first is a direct implementation of the hand calculated y(t), while the other was obtained by using scipy.signal.step() and H(s). The two plots are functionally identical; the initial vertical line in the Part 1 Task 1 plot is due to the user-defined unit step function being affected by resolution.





Part 1, Task 1 + Task 2

The partial fraction expansion results R and P for Y(s) are listed below. A, B, and C (the coefficients calculated in the prelab) were $\frac{1}{2}$, $-\frac{1}{2}$, and 1, which correspond to the values in the R array. The denominator of Y(s) was calculated to be $s(s+4)(s+6)$, so the poles were 0, -4, and -6, which match the P array. K was not included in this listing due to being irrelevant to the lab.

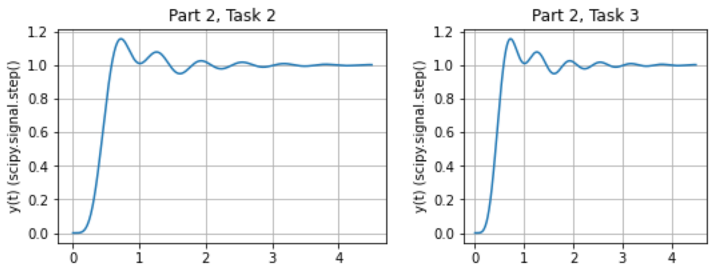
```
Part 1 Task 3:
[ 0.5 -0.5  1. ] [ 0. -4. -6.]
```

4.2 Part 2

The partial fraction expansion results R and P for the step response of the Part 2 equation are listed below.

```
Part 2 Task 1:
[ 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]
```

The time-domain response for the equation is plotted below with two different methods; the first with the resulting equation of the cosine method, and the second with scipy.signal.step() and H(s). Both plots are identical.



Part 1, Task 1 + Task 2

The cosine method was implemented with the function listed below.

```python
def cos_method(roots, poles, t):
  y = 0
  for i in range(len(poles)):
    mag = np.absolute(roots[i])
    ang = np.angle(roots[i])
    alpha = np.real(poles[i])
    omega = np.imag(poles[i])
    y += mag * np.exp(alpha * t) * np.cos(omega * t + ang) * u(t)
  return y
```

Using the cosine method as listed:

$$y_c(t) = 2|k|e^{\alpha t}cos(\omega t + \angle k)u(t)$$

$|k|$ is taken to be the magnitude of each root (and will hence be just the value of real roots). α is the real portion of each pole. ω is the imaginary portion of each pole. $\angle$k is the phase of each root (and is hence 0 degrees for real roots). The method coded above adds the cosine method portion resulting from each root/pole to the final plot. The factor of 2 is actually not present in the code, as it exists to account for portions contributed by conjugate pairs; real roots/poles each contribute a factor of 1.

5 Error Analysis

Most of the lab was not too tedious to implement; a real difficulty presented itself when we had to implement the cosine method. Initially, we attempted to separate the real poles from the complex pole pairs to add them separately to the final equation. Eventually, we figured out that the factor of 2 in the cosine method equation already accounts for the existence of complex conjugate pairs. We then tried to implement the magnitude and angle as the magnitude and phase of the poles, the poles being our only argument aside from t to the function, but, through analysis, realized that these were supposed to derive from the roots. The final cosine method function took roots, poles, and t as arguments, and was implemented properly.

6 Questions

1. The cosine method is still valid for a non-complex pole-residue term because for a non-complex term, $|k|$ will just be the value of the root, ω will be 0 (since there is no complex term in the pole), and $\angle$k will be 0 (since there is no complex term in the root). The factor of 2 will not be present either, since there is no complex conjugate pair. Hence, the cosine method equation simplifies to $y_c(t) = ke^{\alpha t}u(t)$, where the cosine term is evaluated as $\cos(0) = 1$. The resulting equation is just the standard inverse transform of any Laplace term $k\frac{1}{s-\alpha}$.

2. Expectations, instructions, and deliverables were clear for the most part. Implementation of the cosine method was left to be somewhat ambiguous without any supporting listings.

7 Conclusion

In this lab, we learned partial fraction expansion through both manual means as well as its implementation using scipy.signal.residue(). Using the latter proved to be much needed when it came to performing partial fraction expansion on higher-order systems. We also learned to utilize and implement the cosine method, and understood why it is applicable to both real and complex terms.