ECE 351 - 52

Signals and Systems 1

Lab 6

Submitted By :
Owen Blair

Contents

1 Important Notes

It is important to note that plt.show() is needed at the end of the python code to properly show the plots of each function. It was not included in every section of code that plots a function(s) because it is assumed that its included at the end the code.

The web address to the GitHub where LaTeX code is stored is here:
https://github.com/Blairis123/ECE351_Reports

The web address to the GitHub where the Python code is here:
https://github.com/Blairis123/ECE351_Code

2 Part 1 Deliverables

The purpose of part 1 is to plot the hand found step response of the equation in the pre-lab handout and compare it to the scipy.signal.step() step resonse. The hand found step response is:

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

Figure 1:Hand Found vs. sig.step Step Response contains the plots of the hand found step response on an interval of $0 <= t <= 2$. *Listing 1: Finding and Plotting Step Response* contains the code needed to implement the hand found step response, the step response from the scipy.signal.step() function, and print the roots and poles of the step response in the S-domain. *Figure 2: Roots and Poles Output* contains a screenshot of the printed output from the program showing the output of the roots and poles of the step response in the S-domain.

3 Part 2 Deliverables

Part 2 is about using the scipy.signal.residue() function and implementing the cosine method to apply inverse Laplace functions. The plots are on a time interval of $0 <= t <= 4.5$. The cosine method and the scipy.signal.setp() plots can be seen in *Figure 3: Cosine Method vs. scipy.signals.setp()* and the code used to implement the cosine function and plot the result and the scipy.signal.setp() plot can be seen in *Listing 2: Cosine Method and scipy.signal.step() Code.*

4 Results

Figure 1:Hand Found vs. sig.step Step Response

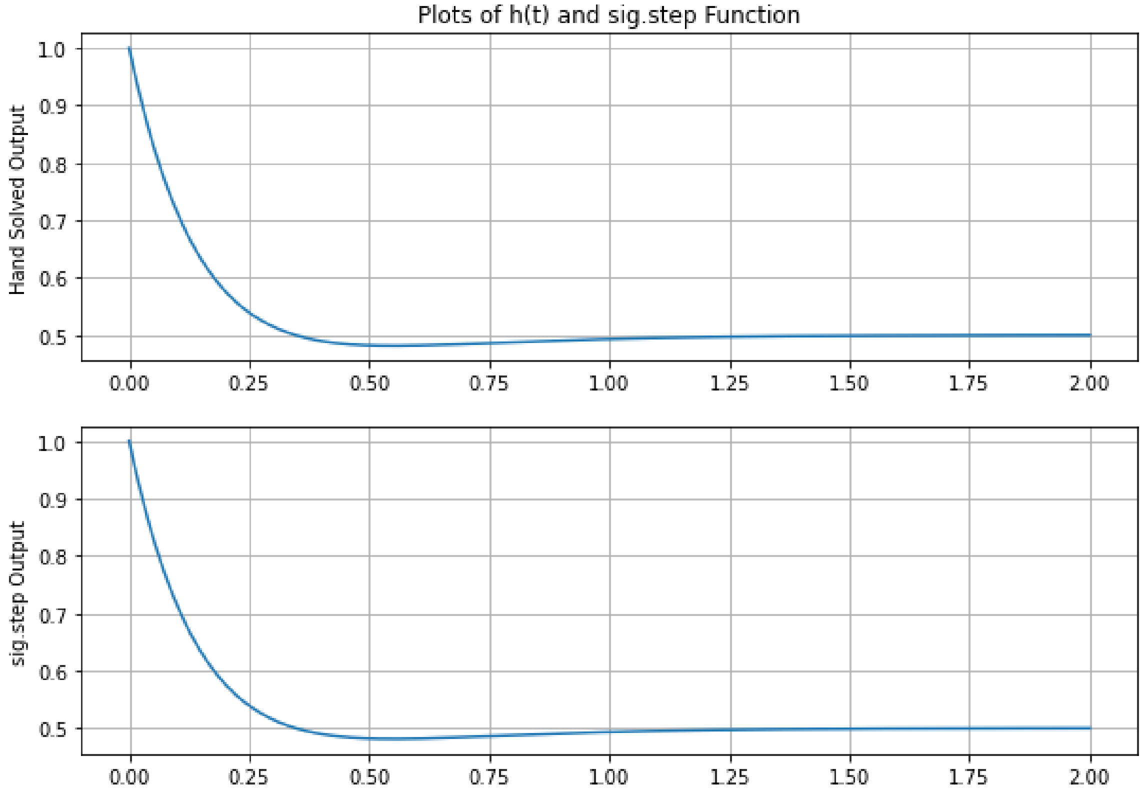
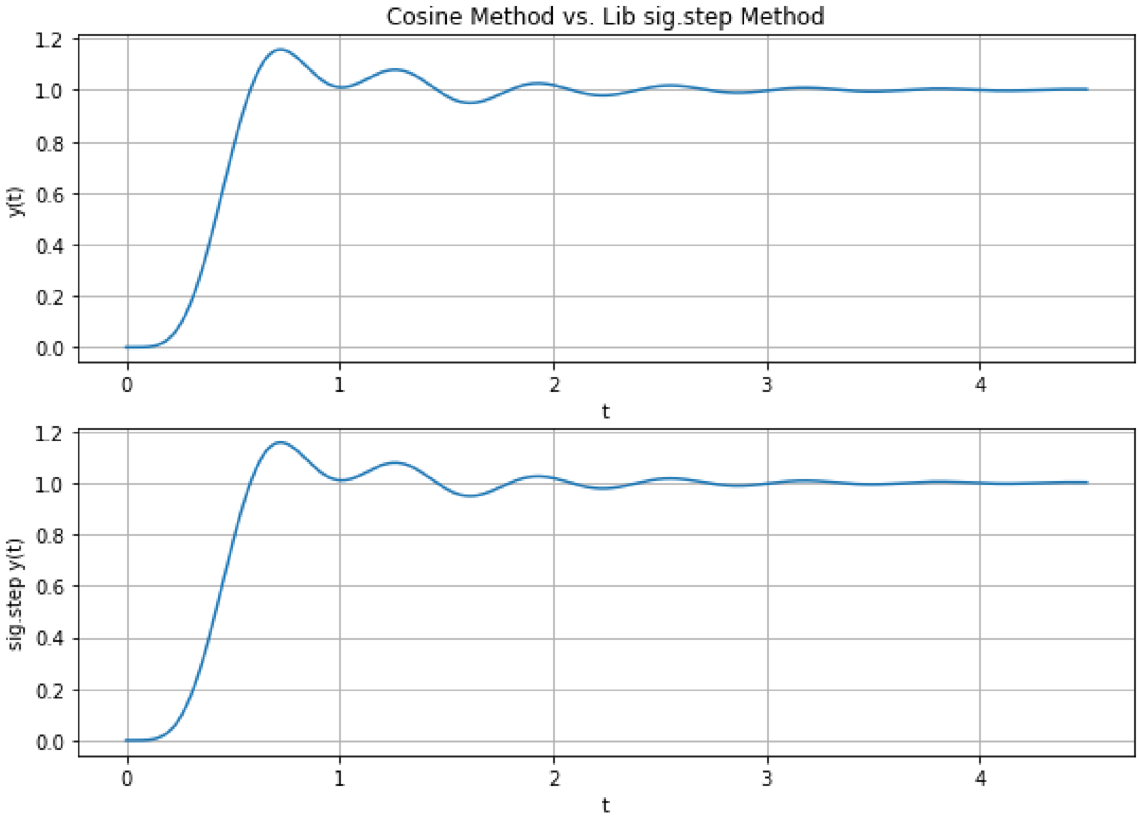


Figure 2: Roots and Poles Output

```
In [21]: runfile('D:/U_of_I/Fall_2021/ECE-351/Lab6/Lab6_OwenBlair.py', wdir='D:/U_of_I/Fall_2021/
ECE-351/Lab6')
Partial frac decomp Roots
[ 0.5 -0.5  1. ]

Partial frac decomp Poles
[ 0. -4. -6.]
```

Figure 3: Cosine Method vs. scipy.signals.setp()



5 Listings

Listing 1: Finding and Plotting Step Response

```
#---------PT 1-------------------------------------------#
    #Define step size
steps = 1e-2

    #t for part 1
start = 0
stop = 2
    #Define a range of t_pt1. Start @ 0 go to 20 (+a step) w/
    #a stepsize of step
t = np.arange(start, stop + steps, steps)

    #Plot prelab results
```

```
h = (0.5 + (-0.5 * np.exp(-4 * t)) + (np.exp(-6 * t))) * stepFunc(t
    , 0, 1)

    #Make the H(s) using the sig.step() function!!

num = [1, 6, 12] #Creates a matrix for the numerator
den = [1, 10, 24] #Creates a matrix for the denominator

tout , yStep = sig.step((num , den), T = t)

den_residue = [1, 10, 24, 0]

    #Make and print the partial fraction decomp
roots, poles, _ = sig.residue(num, den_residue)

print("Partial frac decomp Roots")
print(roots)
print("")
print("Partial frac decomp Poles")
print(poles)


    #Make plots for pt1
plt.figure(figsize=(10,7))
plt.subplot(2,1,1)
plt.plot(t,h)
plt.grid()
plt.ylabel('Hand Solved Output')
plt.title('Plots of h(t) and sig.step Function')

plt.subplot(2,1,2)
plt.plot(t,yStep)
plt.grid()
plt.ylabel('sig.step Output')
```

Listing 2: Cosine Method and scipy.signal.step() Code

```
#------------PART 2-------------------------

    #Define step size
steps = 1e-2

    #t for part 1
start = 0
stop = 4.5
    #Define a range of t_pt1. Start @ 0 go to 20 (+a step) w/
    #a stepsize of step
t_pt2 = np.arange(start, stop + steps, steps)
```

```


#System is:
#y^(5)(t) + 18y^(4)(t) + 218y^(3)(t) + 2036y^(2)(t) + 9085y^(1)(t)
    + 25250y(t)
#          = 25250x(t)
#
#The ^(number) signifies the diritive of the function y(t). I.e. y
    ^(6)(t) would
#be the 6th diritive of the funciton y(t)


    #Make numerator and denomentaor for sig.residue()
num_pt2 = [25250]
den_pt2 = [1, 18, 218, 2036, 9085, 25250, 0]

roots_pt2, poles_pt2, _2 = sig.residue(num_pt2, den_pt2)

print("Roots and Poles for pt 2")
print("Roots_pt2")
print(roots_pt2)
print("")
print("Poles_pt2")
print(poles_pt2)

    #COSINE METHOD! Using the poles found previously
ytCosineMethod = 0

    #Range iterates through each root
for i in range(len(roots_pt2)):
    angleK = np.angle(roots_pt2[i])
    magOfK = np.abs(roots_pt2[i])
    W = np.imag(poles_pt2[i])
    a = np.real(poles_pt2[i])

        #DEBUG!!!
    print("angle = ", angleK)
    print("magnatude = ", magOfK)
        #END DEBUG

    ytCosineMethod += magOfK * np.exp(a * t_pt2) * np.cos(W * t_pt2
    + angleK) * stepFunc(t_pt2, 0, 1)

#Make the lib generated step response
den_pt2_step = [1, 18, 218, 2036, 9085, 25250]
tStep_pt2, yStep_pt2 = sig.step((num_pt2,den_pt2_step), T = t_pt2)

    #Show Plots
plt.figure(figsize=(10,7))
```

```
plt.subplot(2,1,1)
plt.plot(t_pt2, ytCosineMethod)
plt.grid()
plt.xlabel('t')
plt.ylabel('y(t)')
plt.title('Cosine Method vs. Lib sig.step Method')


plt.subplot(2,1,2)
plt.plot(tStep_pt2, yStep_pt2)
plt.grid()
plt.xlabel('t')
plt.ylabel('sig.step y(t)')
```

6 Questions

6.1 Question 1

The reason that the cosine method works for non-complex poles is because ω can be equal to zero and the method will work. Because ω is the magnitude of the imaginary part of a complex number, when it is equal to zero the number is not imaginary. Within the cosine method t is multiplied by omega and the angle is zero so the result will be that the cosine will be a 1. This doesn't change the result of the inverse Laplace because the $\|K\|e^{at}$ component is being multiplied by 1. It also turns out that the $\|K\|e^{at}$ component is the result of the inverse Laplace of a simple non-complex root.