Project Report for ECE 351

Lab 6: Partial Fraction Expansion

Isaias Munoz

October 2, 2022

University of Idaho

Contents

1 Introduction

The purpose of this lab is to use the in built house function *scipy.signal.residue*() to perform partial fraction expansion.

2 Methodology

2.1 Part 1

The first task that was assigned before the Lab began and that was to solve $y(t)$ fot a step input using partial expansion.

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

The solution was found by taking it to the Laplace domain and then computing partial fraction expansion by hand. What was obtain is below.

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

I then move to the first two actual tasks which were to plot the step response of the above equation and then use *scipy.signal.residue*() function to perform partial fraction expansions on the S-domain and compare both results.

```
#For a minute i thought it wasnt working but i had extra brackets no need
# to create a new time range
# t1 = np . arange (t[0] , t[len(t)-1]+steps , steps )
# ystep = np.zeros ( t.shape  ) #initializing y as all zeroe
def stepsponse(t):
    y=(.5 - .5*np.exp(-4*t) + np.exp(-6*t))*stepfunc(t)
    return y
# ystep=ystep=(.5 + np.exp(-6*t) - .5*np.exp(-4*t1))*stepfunc(t1)
ystep=stepsponse(t)

plt . figure ( figsize = (12 , 8) )
plt . plot (t , ystep )
plt . ylabel ('Y output')
plt . xlabel ('t convolution')
plt . title ('Step function by hand')
plt . grid ()

```

Figure 1: Code for graphing the hand solved step function

Figure 1 has the code to graph the hand solved step function found. It is a function being passed the time range and computing the output. I then graphed it and the graph can be found under the results sections. It is important to note that inside the function I am calling another function $stepfunc()$ and that is due to a previous function that I have been using a lot from Lab 2. I continued with task 2 and the code for implementing it is Figure 2.

```
65  #task 2
66  #plotting H(S) found in prelab
67  #H(S)=s^2+6s+12/s^2+10s+24
68
69  # scipy.signal.step() takes two things essentially the transfer func denom and
70  #numera and then the time range
71
72
73  num = [1, 6, 12]
74   #Creates a matrix for the numerator coeffecients of transfer func numera
75  den = [1, 10, 24]
76  #Creates a matrix for the denominator coeffecients of transfer func denom
77
78  zout , mout = scipy.signal.step((num,den),T=t) #gives a step response
79  #scipy.signal.step can be shorten to sig.step() like in lab 3
80
81  print(zout,mout)
82  plt . figure ( figsize = (12 , 8) )
83  plt . plot (zout , mout )
84  plt . ylabel ('Y output')
85  plt . xlabel ('t range')
86  plt . title ('Step function using scipy.signal.step ')
87  plt . grid ()
88
```

Figure 2: Code for using $scipy.signal.step()$

Similar to Lab 5 i created a coefficient matrix of the numerator and denominator of the transfer function found earlier from the differential equation above. Lines 73-75 have the coefficients. I then passed it to another matrix and called the in house function $scipy.signal.step()$ and passed it the numerator and denominator as well as the time range. Th function is suppose to give me the step response. I then compared it to the my hand calculated and they were equivalent. The result for the graph using the in house function can be seen under the result sections.

Task 3 consisted in finding the "convolution" with the step function but using *scipy.signal.resdiue*(). In the Laplace domain a convolution turns into a multiplication between the transfer function and step function. Therefore a $1/s$ term is multiplied with my regular transfer function and now the coefficients change a little and a new numerator and denominator need to be passed to *sig.residue*().

```
#task3

#Y(S)=H(S)X(S) where X(S) is step func=1/s
print ('this is the solution to the the first DEQ')
a = [1, 6, 12]
b = [1, 10, 24,0]



rout, pout,kout= sig.residue(a,b)
#scipy.signal.residue() can also be called
#rout gives u the roots so ur A and B and C term after the partial expansion
#pout gives us the poles where they cross in this case (0) and (s+4) and (s+6)
#kout dont know what it gives u lol
print (rout,pout,kout)

#Make and print the partial fraction decomp


#end of part 1 tasks
##########################################################################
```

Figure 3: Code for using *scipy.signal.residue*()

As seen in Figure 3 my coefficients change do the $1/s$ multiplication and now I have a zero as the place holder of my constant for the denominator. I passed the numerator and denominator to *sig.residue*() and had it give me the roots, poles and residue of the given terms. This comes handy because instead of solving by hand like I did earlier it solves it for me. Comparing the results to the partial expansion are under the result sections they match to what I calculated by hand earlier.

2.2 Part 2

Part 2 was to use *scipy.signal.residue*() to perform partial fraction expansion on the function below since by hand it would be difficult to analyze.

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

Below in figure 4 is the code used to solve the function.

```
#Beginnig of Part 2 task 1

a1=[25250]
b1=[1,18,218,2036,9085,25250,0]
arout, apout,akout= sig.residue(a1,b1)

print (' this is the Root solutions to the long DEQ')
print(arout)
print (' this is the Poles solutions to the long DEQ')
print(apout)
for i in range (len(apout)):
    print( apout[i])
print (' this is the K (residue of a given term) solutions to the long DEQ')
print(akout)
#plottingnew time range
# steps = .001 # Define step size
tnew = np . arange (0 , 4.5 + steps , steps )

def cosmethod(arout,apout,tnew):
    y=0

    for i in range (len ( arout ) ): #startingforloop
       # y+=2*abs(arout[i])*np.exp(np.real(apout[i]*tnew))*np.cos(np.imag(apout
        y += (2 * abs(arout[i]) * np.exp(np.real(apout[i]) * tnew)* np.cos(np.i
    return y
# print(abs(3+4j))

ystepcosmet=cosmethod(arout,apout,tnew)

plt . figure ( figsize = (12 , 8) )
plt . plot (tnew , ystepcosmet )
plt . ylabel ('Y output')
plt . xlabel ('new time range')
plt . title ('Step function by cosine method')
plt . grid ()
#from previous a1 and b1
    #a1=[25250]
# b1=[1,18,218,2036,9085,25250,0]
```

Figure 4: Code for using *scipy.signal.residue*() on new function

I proceeded with solving for the transfer function which was easily done and created a numerator and denominator matrix that I passed on to *scipy.signal.residue*(). I printed out the roots, poles, residue of the terms. and then proceeded to use the cosine method to plot out the results. Defining the cosine function is important to notice that *scipy.signal.residue*() gives out the roots of the cosine method since it is doing partial fraction already. All you have to do is add the repeated terms by a loop and build the function given since were using the cosine method, this took me a while to grasp. Lastly I needed to verify using the *scipy.signal.step*(). FIgure 5 has the code to do so.

```python

newb1 = [1, 18, 218, 2036, 9085, 25250]

lout, nout=scipy.signal.step((a1,newb1),T=tnew)

# num = [1, 6, 12]
#  #Creates a matrix for the numerator coeffecients of transfer func numera
# den = [1, 10, 24]
# #Creates a matrix for the denominator coeffecients of transfer func denom

# zout , mout = scipy.signal.step((num,den),T=t) #gives a step response
# #scipy.signal.step can be shorten to sig.step() like in lab 3

# print(zout,mout)
plt . figure ( figsize = (12 , 8) )
plt . plot (lout, nout )
plt . ylabel ('Y output')
plt . xlabel ('t range')
plt . title ('Verify the sine method step function using scipy.signal.step')
plt . grid ()
```

Figure 5: Code for using *scipy.signal.residue*()

It is important to note that a new denominator array was formed because we are calling the in house function *scipy.signal.step*() which is already taking into account the multiplication of the step function so the denominator array changes I then graphed it and its graph can be found under the result section of this lab.

3 Results

3.1 Part 1

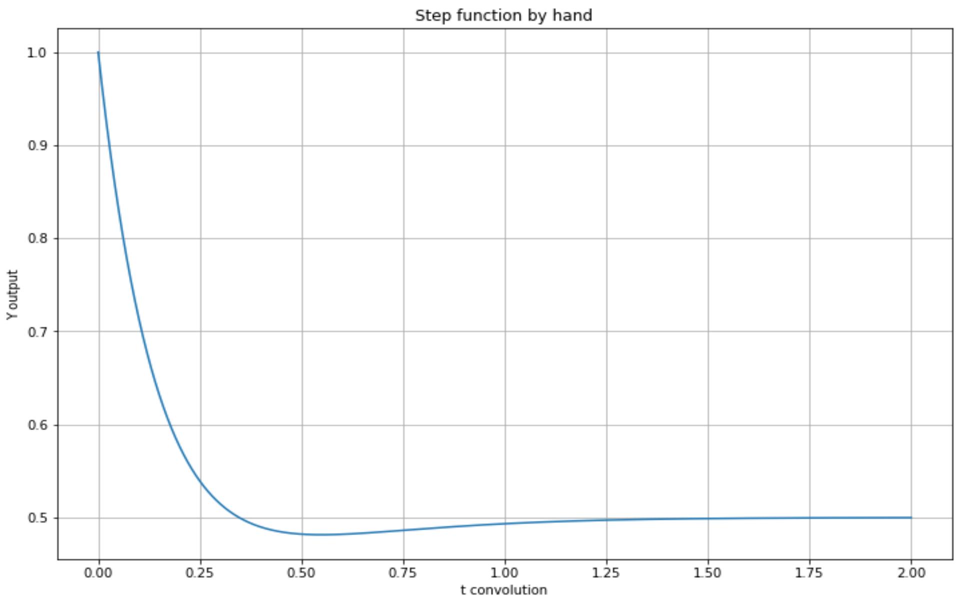
1 

Figure 6: Step input solved by hand

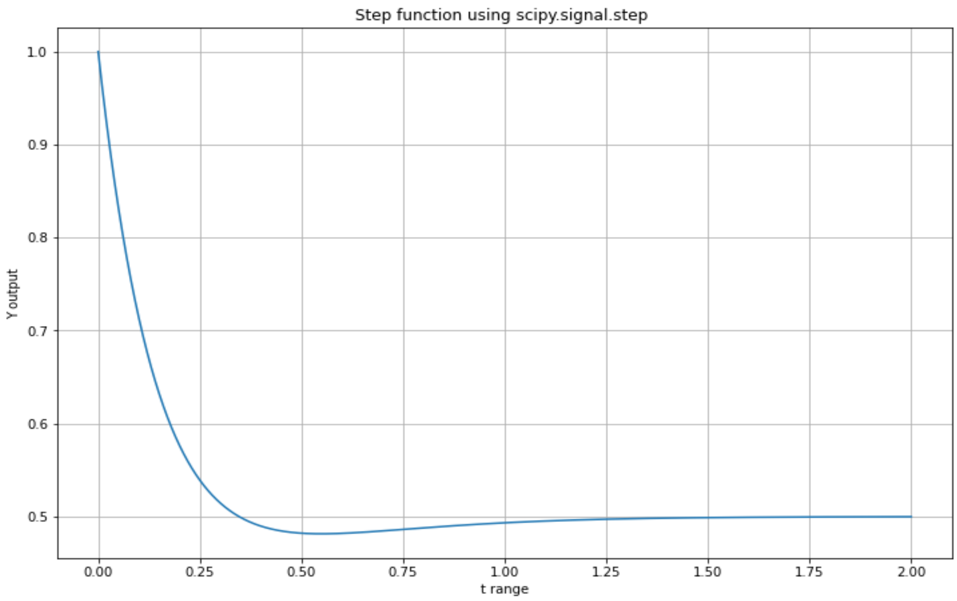
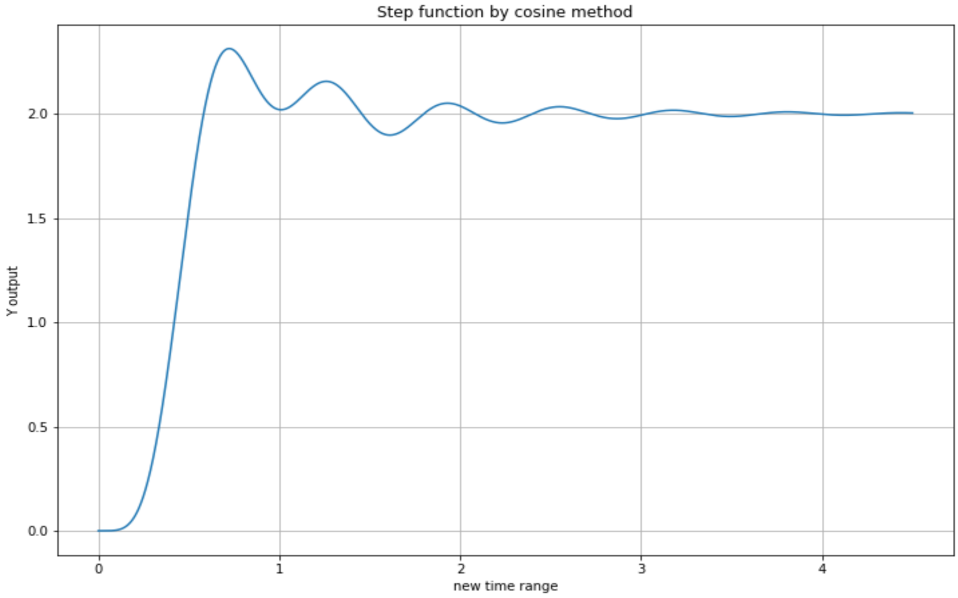
1 

Figure 7: Verifying using $scipy.signal.step()$

1
```
this is the solution to the the first DEQ
[ 0.5 -0.5  1. ] [ 0. -4. -6.] []
```

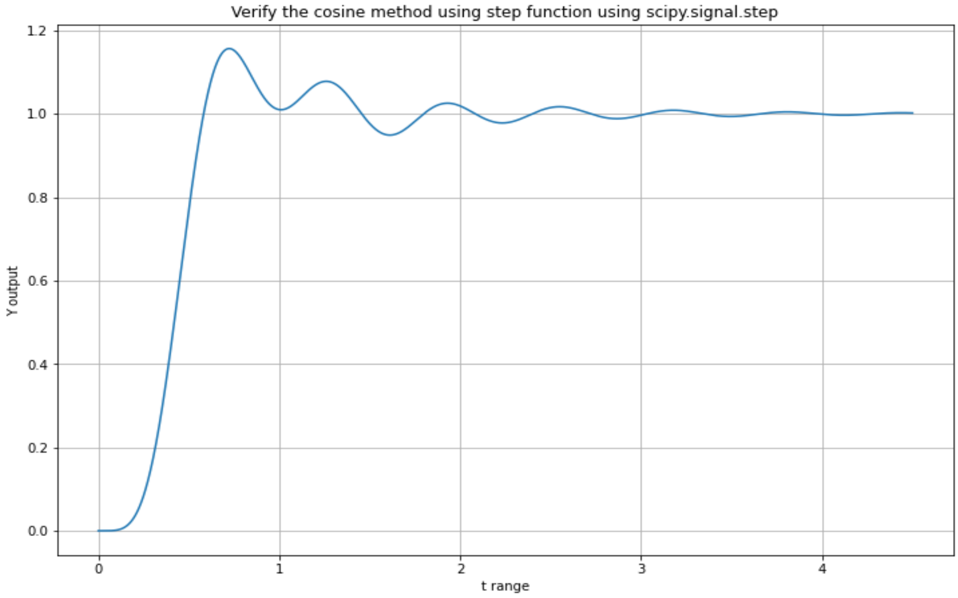
Figure 8: Verifying the roots, poles,residue using $scipy.signal.residue()$

3.2 Part 2



1

Figure 9: using *scipy.signal.residue*() on part 2



1

Figure 10: Verifying using *scipy.signal.step*()

```
this is the Root solutions to the long DEQ
[ 1.        +0.j         -0.48557692+0.72836538j -0.48557692-0.72836538j
 -0.21461963+0.j          0.09288674-0.04765193j  0.09288674+0.04765193j]
this is the Poles solutions to the long DEQ
[  0. +0.j  -3. +4.j  -3. -4.j -10. +0.j  -1.+10.j  -1.-10.j]
this is the K (residue of a given term) solutions to the long DEQ
[]
```

Figure 11: Verifying the roots, poles,residue using $scipy.signal.residue()$

4 Questions

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

Because if it is not complex it will cancel out at $cosine(o)$ when it is being evaluated leaving just real components. Which your left with whatever is in front of the cosine term's since cosine goes to 1 at $cos(0)$.

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

I was just confused on the cosine method for a while and the in house function $scipy.signal.residue()$. They are the same really but I was just mixing up poles vs roots. I was also mixing up the equation the cosine method produces. But once I figure it out I think I understood the lab better. We touched on the cosine method briefly and used the sine method much more in class which is why I was having a hard time understanding it.