Report - Assignment 8

Akshay Anand (EE16B046)

April 3, 2018

Abstract

In this week's assignment, the functions in sympy were used to solve the equations for a low-pass and a high-pass filter, employed using opamps, and as such get the transfer function of the system. Then, this transfer function was used to find the output waveform when an input with different frequency components were given, like $\left(\sin\left(2000\pi t\right) + \cos\left(2 \times 10^6 \pi t\right)\right) u_0\left(t\right)$ using the functions that were used in last week's assignment, particularly *scipy.signal.lsim* and *scipy.signal.impulse*. The latter was used to find the waveform when the laplace transform of the output was known and the former was used to find the output when the input signal in the time domain is known, as well as the transfer function (in the frequency domain). Then, the input to the system was varied from a step function, to an exponentially decaying sinusoid and the various outputs that were obtained were noted.

Libraries and Global Variables Used

```
from sympy import *
import matplotlib.pyplot as plt
import numpy as np
import math
import scipy.signal as sp

init_session        # Denotes that, an interactive sympy session has begun
s = symbols('s')    # The symbol for denoting the 's' parameter in Laplace
    transforms
```

1 Low-Pass Filter

In this, a low-pass filter circuit was realised using capacitors, resistors, and an opamp in negative feedback. The various KCL equations at the different nodes, that were used for solving the circuit are:

$$V_m = \frac{V_o}{G}$$

$$V_p = V_1 \frac{1}{1 + j\omega R_2 C_2}$$

$$V_o = G\left(V_p - V_m\right)$$

$$\frac{V_i - V_1}{R_1} + \frac{V_p - V_1}{R_2} + j\omega C_1 \left(V_o - V_1\right) = 0$$

In the above equations, G is the ratio in which the output voltage is divided in the resistive feedback path of the opamp, as well as numerically equal to the gain of the opamp.V_p is the voltage at the positive terminal of the opamp, while V_m is that in the negative terminal. V_i is the input node of the system, and the resistor R_1 is connected between this and node 1 (corresponding to V_1). R_2 is connected between V_1 and V_p, while C_2 is connected between V_p and ground. Finally, the capacitor C_1 is connected between V_1 and V_o, in another feedback path that would be blocked for DC signals.

These above equations were solved using matrices in sympy using the following code:

```
def lowpass(R1,R2,C1,C2,G,Vi):
    s=symbols('s')
    A=Matrix([[0,0,1,-1/G],[-1/(1+s*R2*C2),1,0,0], [0,-G,G,1],[-(1/R1)-(1/R2)-(s*C1
        ),1/R2,0,s*C1]])
    b=Matrix([0,0,0,-Vi/R1])
    V=A.inv()*b
    return (A,b,V)
```

In this, matrix A is the coefficient matrix and B is the matrix having the independent sources.

The various functions related to calculating the time domain output are declared below. First, is the one that returns the frequency domain output of the system for a particular input. It takes 2 parameters: the laplace transform of the input and filterType, which is whether to use highpass or lowpass filter. High pass filter will be implemented in the next section.

```
def get_response_freq_domain(h,filterType):
    if (filterType == 'l'):                 # Low pass filter condition
        A,b,V = lowpass(10000,10000,1e-9,1e-9,1.586,h)
    elif (filterType == 'h'):               # High pass filter condition
        A,b,V = highpass(10000,10000,1e-9,1e-9,1.586,h)
    Vo=V[3]                                 # The 4th element is Vo

    # The below lines convert the sympy expression to a numpy function and get
        values for the resultant equation
    w=np.logspace(0,8,801)
    ss=1j*w
    hf=lambdify(s,Vo,'numpy')
    v=hf(ss)
    return Vo,w,abs(v),np.angle(v)          # Vo is returned so as to calculate time
        domain output later
```

Now, the time domain output has to be calculated from the frequency domain sympy expression that is returned. For that, the plot_output_time_domain() function is declared.

```
def get_numpy_array_from_Poly(num,den):    # This is called only from the function
    declared below
    isFloat = False
    try:
        num = Poly(num).all_coeffs()        # If the numerator is just a constant
            value, this line will throw an error
    except GeneratorsNeeded:
        num = num                           # If an error is thrown above, it means
            numerator is just a constant and hence, use as is
        isFloat = True
    den = Poly(den).all_coeffs()            # Get coefficients of denominator

    # The below steps are required as the coefficients obtained above are not in a
        proper format to be used in numpy.poly1d(), so a numpy array is generated
        from the above values, and then returned to the parent function.
    den2 = []
    num2 = []
    for i in den:
        den2.append(float(i))
    den2 = np.array(den2)

    if (isFloat):    # If numerator is a constant, no need to iterate through it
        num2 = num
    else:
```

```
        for i in num:
            num2.append(float(i))
        num2 = np.array(num2)
    return num2,den2

def get_output_time_domain(Y,t,steps):
    simplY = simplify(Y)      # This gets the simplified expression of laplace
        transform
    num = fraction(simplY)[0]    # Gets numerator of transform
    den = fraction(simplY)[1]    # Gets denominator of transform
    num2,den2 = get_numpy_array_from_Poly(num,den)  # This converts the Poly()
        object obtained above to a numpy array

    # Calculate poly1d expressions for numerator and denominator
    num2 = np.poly1d(num2)
    den2 = np.poly1d(den2)

    # Use scipy signal toolbox to get final output waveform
    Y = sp.lti(num2,den2)
    t = np.linspace(0.0,t,steps)
    t,y=sp.impulse(Y,None,t)
    return t,y
```

The second function is called from the main program for calculating the output, which in turn calls the first function to find and return the coefficients of the laplace transform of the output, so that *scipy.signal.impulse* can calculate the output correctly.

The above function can be used if the laplace tranform of the input is known. If, only the time domain representation of the input is known, the following function can be used to calculate the output using *scipy.signal.lsim* instead of *scipy.signal.impulse*. The function takes 3 parameters: The sympy expression of the transfer function, time domain input wave as well as the time interval to calculate output.

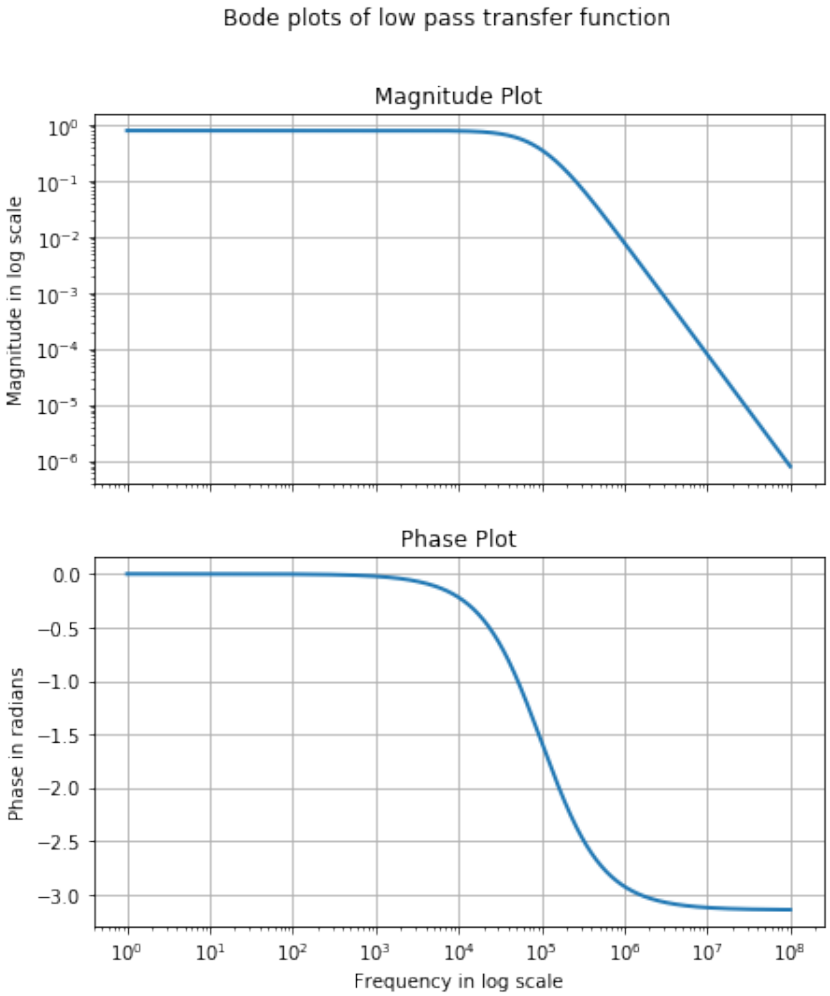
```
def get_output_with_lsim(H,x,t):

    # The lines below are similar to that in get_output_time_domain() function
    simplH = simplify(H)
    num = fraction(simplH)[0]
    den = fraction(simplH)[1]
    num2,den2 = get_numpy_array_from_Poly(num,den)

    num2 = np.poly1d(num2)
    den2 = np.poly1d(den2)

    H = sp.lti(num2,den2)
    t,y,sec=sp.lsim(H,x,t)    # Here, lsim() is used to find output from time domain
        input.
    return t,y
```

1.1 Transfer function of system

If, an input V_i is given as an impulse (i.e laplace transform input is 1), then, the transfer function of the above system can be known. Thus, the transfer function, so calculated and plotted is:

```
Vi = 1
H_l,w,v,ph = get_response_freq_domain(Vi,'l')

# The Bode plots of transfer function are plotted
fig, axes = plt.subplots(2, 1, figsize=(7, 8), sharex = True)
plt.suptitle('Bode␣plots␣of␣low␣pass␣transfer␣function')
```

```
axes[0].loglog(w,v,lw=2)
axes[0].grid()
axes[0].set_ylabel('Magnitude in log scale')
axes[0].set_title('Magnitude Plot')

axes[1].semilogx(w,ph,lw=2)
axes[1].grid()
axes[1].set_xlabel('Frequency in log scale')
axes[1].set_ylabel('Phase in radians')
axes[1].set_title('Phase Plot')
plt.show()
```



Thus, it is evident that, the system above acts as a low pass filter, as it passes low frequencies with almost no attenuation, while highly attenuates high frequency inputs. Also, the 3dB bandwidth (magnitude at phase of $\frac{\pi}{2}$), is around 10^5 Hz, which means that, any frequency below this, it will pass with minimal attenuation.

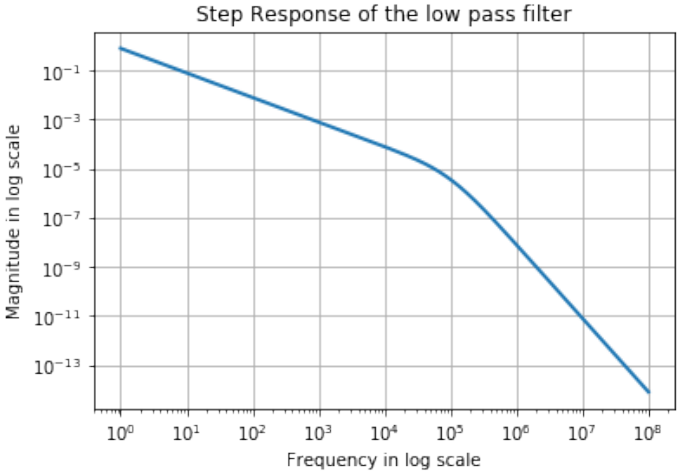
1.2 Step Response

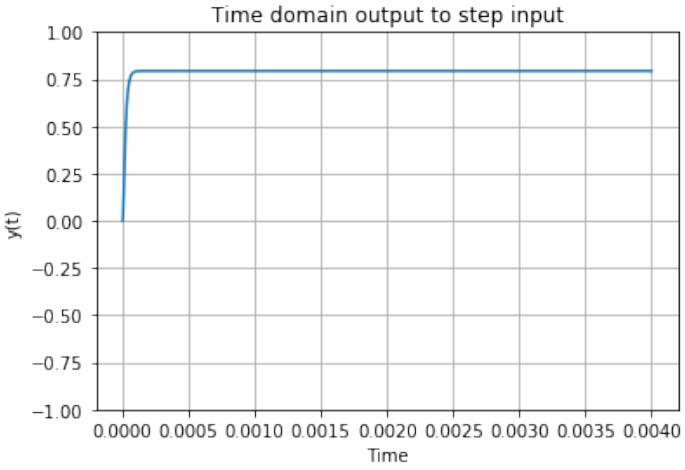
The response of the low pass filter to a step input is calculated. For this, the laplace transform of the step input ($\frac{1}{s}$) is used as input to the above functions and the step response in both time and frequency domains are plotted.

```
Vi = 1/s
Y,w,v = get_response_freq_domain(Vi,'l')
t,y = get_output_time_domain(Y,4e-3,10001)

# The magnitude plot of the Laplace transform of the output is plotted
plt.loglog(w,v,lw=2)
plt.grid()
plt.title('Step Response of the low pass filter')
```

```
plt.xlabel('Frequency in log scale')
plt.ylabel('Magnitude in log scale')
plt.show()

# The time domain output is plotted
plt.plot(t,y)
plt.grid()
plt.title('Time domain output to step input')
plt.xlabel('Time')
plt.ylabel('y(t)')
plt.ylim(-1,1)
plt.show()
```

The output plots obtained where



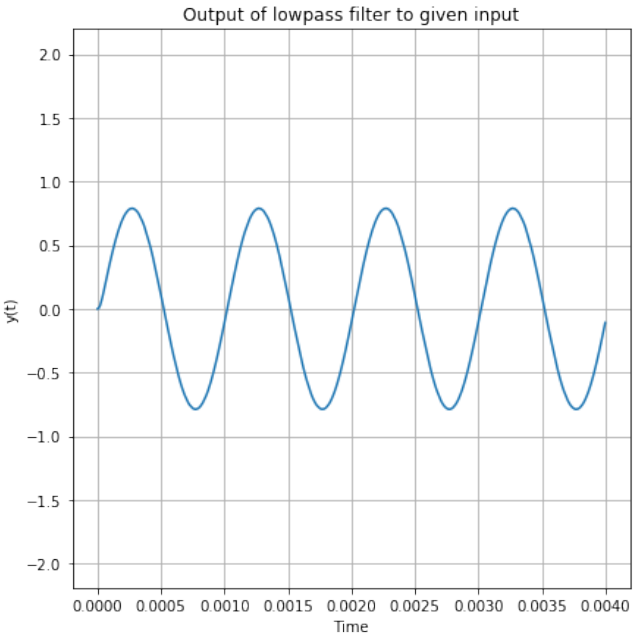


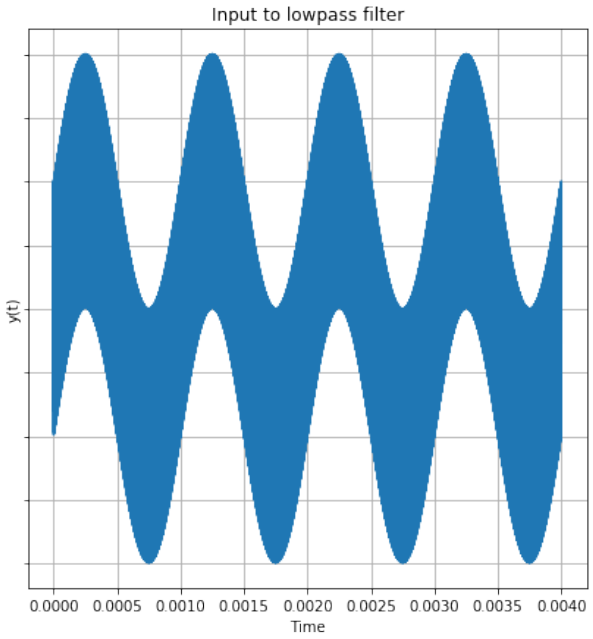
1.3 Input as $(\sin(2000\pi t) + \cos(2 \times 10^6 \pi t))\, u_0(t)$

For this, the get_output_with_lsim() function is used, wherein we know the time domain input, and the transfer function of the system (calculated above as H). Using these and *scipy.signal.lsim*, the output time domain wave is calculated. In the below code segment, both the input and output waves are plotted for comparison.

```
t = np.linspace(0.0,4e-3,100001)      # Time for which output is to be calculated
x = np.sin(2000*math.pi*t) + np.cos(2*(10**6)*math.pi*t)    # Input wave is declared
t,y = get_output_with_lsim(H_l,x,t)    # The transfer function calculated before is
    used here

fig, axes = plt.subplots(1, 2, figsize=(15, 7), sharey = True)
axes[0].plot(t,y)
axes[0].grid()
axes[0].set_title('Output of lowpass filter to given input')
axes[0].set_xlabel('Time')
axes[0].set_ylabel('y(t)')

axes[1].plot(t,x)
axes[1].grid()
axes[1].set_title('Input to lowpass filter')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('y(t)')
plt.show()
```

The graphs obtained are:





Thus, it is found that, although input had high frequency components, the circuit filters all of this, and gives output as only the low frequency component.

2 High-Pass Filter

The only difference in this implementation is that the KCL equations used for calculating V_o would be different. The KCL equations for a high pass filter using opamps, resistors and capacitors are

$$V_m = \frac{V_o}{G}$$

$$V_p = V_1 \frac{j\omega R_3 C_2}{1 + j\omega R_3 C_2}$$

$$V_o = G\left(V_p - V_m\right)$$

$$j\omega C_1 \left(V_i - V_1\right) + j\omega C_2 \left(V_p - V_1\right) + \frac{V_o - V_1}{R_1} = 0$$

In the above equations, G is the ratio in which the output voltage is divided in the resistive dividing feedback path of the opamp, as well as numerically equal to the gain of the opamp.V_p is the voltage at the positive terminal of the opamp, while V_m is that in the negative terminal. V_i is the input node of the system, and the capacitor C_1 is connected between this and node 1 (corresponding to V_1). C_2 is connected between V_1 and V_p, while R_3 is connected between V_p and ground. Finally, the resistor R_1 is connected between V_1 and V_o, in another feedback path.

A python function is declared that solves the above equations similar to that done for lowpass filter.

```
def highpass(R1,R3,C1,C2,G,Vi):
    s=symbols('s')
    A=Matrix([[0,0,1,-1/G],[-(s*R3*C2)/(1+s*R3*C2),1,0,0], [0,-G,G,1],[-(s*C1)-(s*
        C2)-(1/R1),s*C2,0,1/R1]])
    b=Matrix([0,0,0,-Vi*s*C1])
    V=A.inv()*b
    return (A,b,V)
```

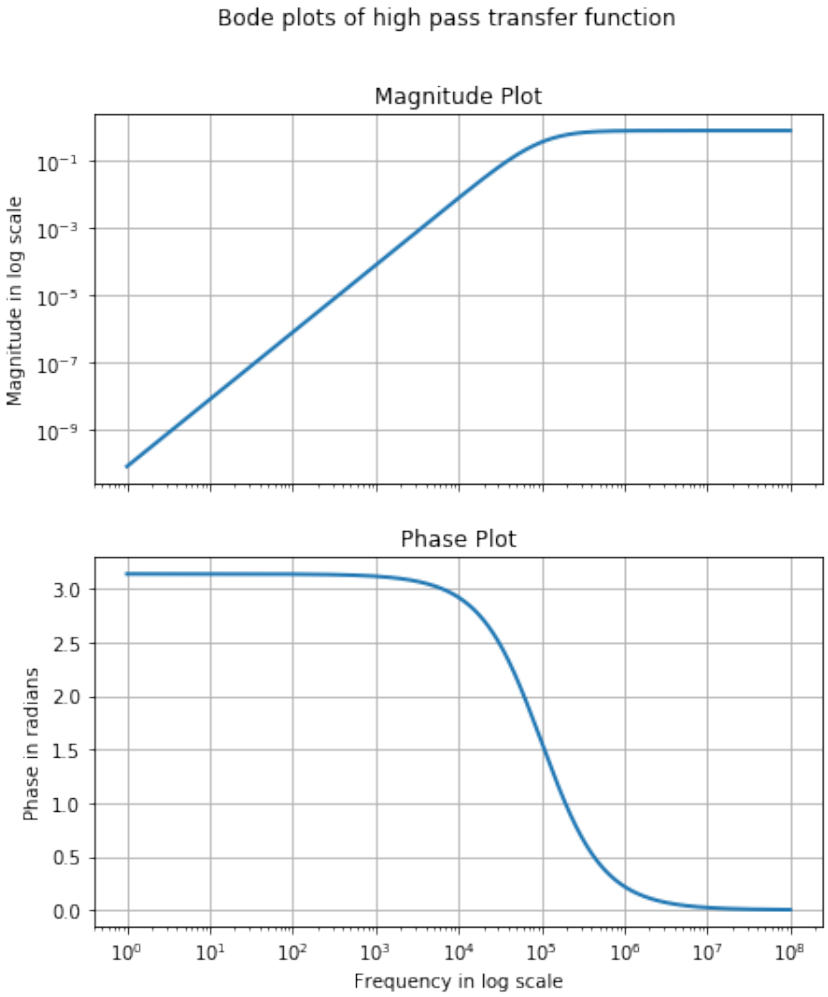
All the other functions that were declared to calculate and plot the output transforms and signals can be used here also, as except the equations, nothing else changes in the overall logic of the program.

2.1 Transfer function of the system

Similar to before, if given an impulse as an input (with laplace transform of V_i as 1), we would get the transfer function.

```
Vi = 1
H_h,w,v,ph = get_response_freq_domain(Vi,'h')  # Here, 'h' is passed to the
    function to use the high pass filter equations

# The transfer function is plotted
fig, axes = plt.subplots(2, 1, figsize=(7, 8), sharex = True)
plt.suptitle('Bode plots of high pass transfer function')
axes[0].loglog(w,v,lw=2)
axes[0].grid()
axes[0].set_ylabel('Magnitude in log scale')
axes[0].set_title('Magnitude Plot')

axes[1].semilogx(w,ph,lw=2)
axes[1].grid()
axes[1].set_xlabel('Frequency in log scale')
axes[1].set_ylabel('Phase in radians')
axes[1].set_title('Phase Plot')
plt.show()
```

Graph obtained is:

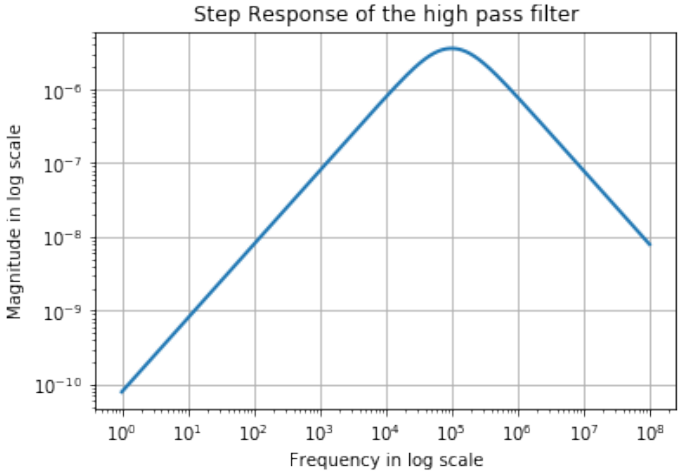


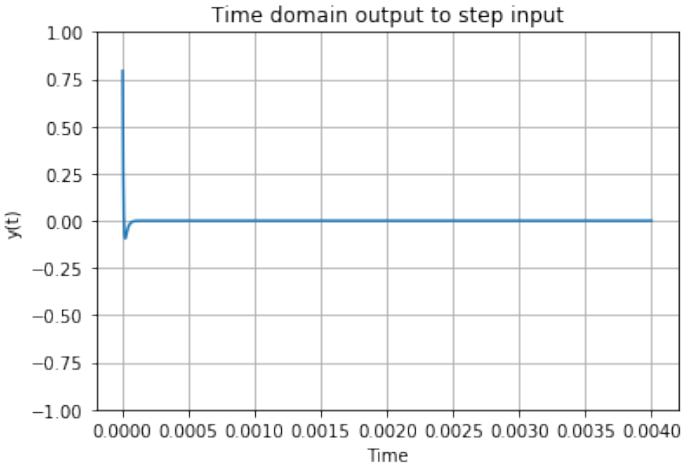
Thus, it is evident that, this system will highly attenuate low frequency components of input and pass through high frequency components with minimal attenuation. Here also, the 3dB bandwidth is around 10^5 Hz, which means that any frequency above this, it will pass with minimal attenuation.

2.2 Step response of the system

In this, the input V_i is given as $\frac{1}{s}$, which is the step input and the output obtained is plotted.

```
Vi = 1/s
Y,w,v = get_response_freq_domain(Vi,'h')
t,y = get_output_time_domain(Y,4e-3,10001)

# The magnitude plot of the Laplace transform of the output is plotted
plt.loglog(w,v,lw=2)
plt.grid()
plt.title('Step Response of the high pass filter')
plt.xlabel('Frequency in log scale')
plt.ylabel('Magnitude in log scale')
plt.show()

# The time domain output is plotted
plt.plot(t,y)
plt.grid()
plt.title('Time domain output to step input')
plt.xlabel('Time')
plt.ylabel('y(t)')
plt.ylim(-1,1)
plt.show()
```





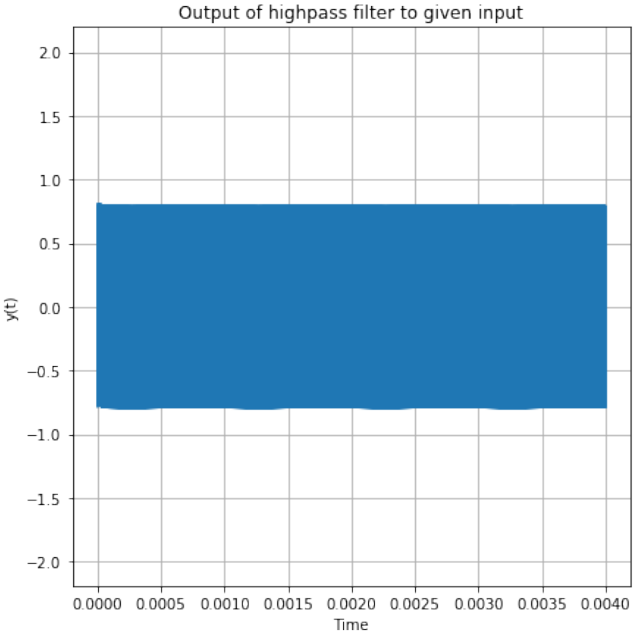
2.3 Input as $(\sin(2000\pi t) + \cos(2 \times 10^6 \pi t))\, u_0(t)$

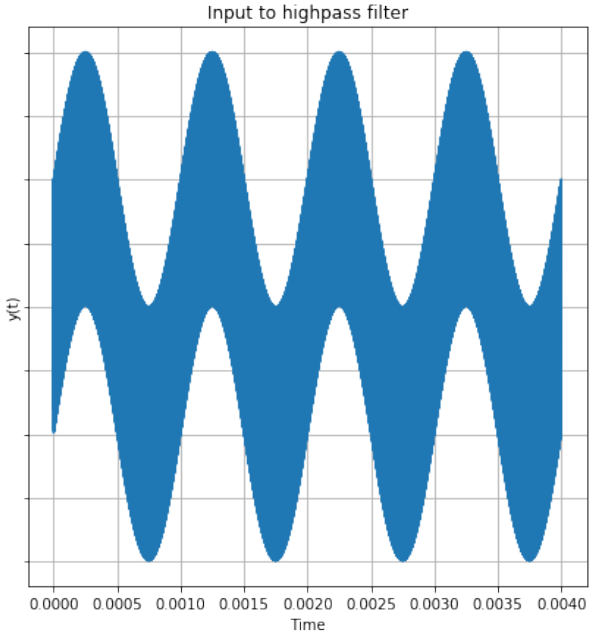
In this, the same input, which was given to the lowpass filter, is given to the highpass filter. In the lowpass filter case, it was seen that, only the low frequency component was passed, so here the expected output is that only the high frequency component should pass.

```
t = np.linspace(0.0,4e-3,100001)      # Time for which output is to be calculated
x = np.sin(2000*math.pi*t) + np.cos(2*(10**6)*math.pi*t)    # Input wave is declared
t,y = get_output_with_lsim(H_h,x,t)    # The transfer function calculated before is
    used here

fig, axes = plt.subplots(1, 2, figsize=(15, 7), sharey = True)
axes[0].plot(t,y)
axes[0].grid()
axes[0].set_title('Output of highpass filter to given input')
axes[0].set_xlabel('Time')
axes[0].set_ylabel('y(t)')

axes[1].plot(t,x)
axes[1].grid()
```

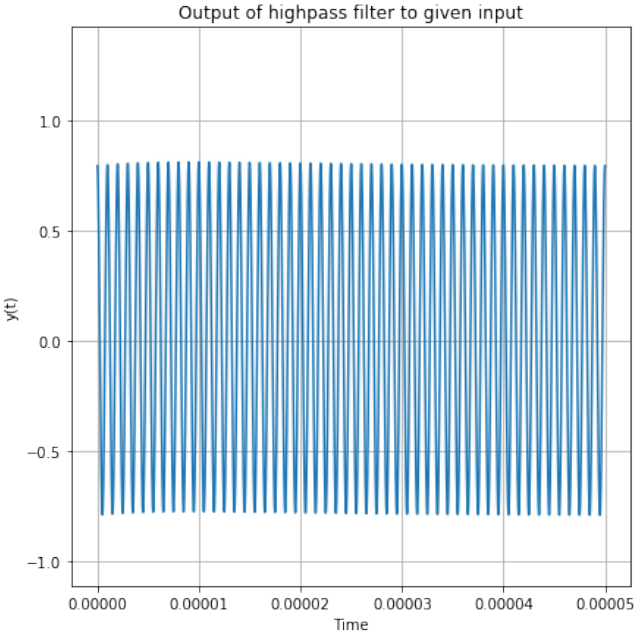
```
axes[1].set_title('Input to highpass filter')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('y(t)')
plt.show()
```

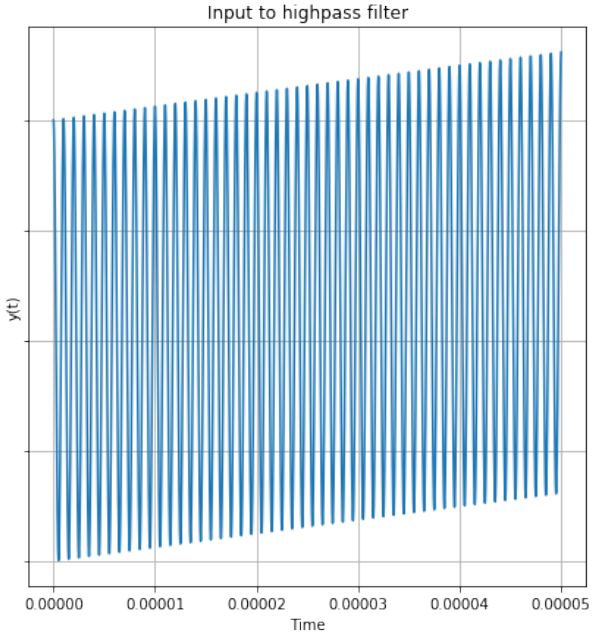
Outputs obtained are:





Thus, it is seen that, the output is indeed only the high frequency component of the input. Outputs with a shorter time range are plotted below, to actually see the high frequency wave.





The slow increase in the input is the beginning of the low frequency component of the wave for the input, which is not seen in the output, as this component is filtered off.

2.4 Exponentially decaying wave

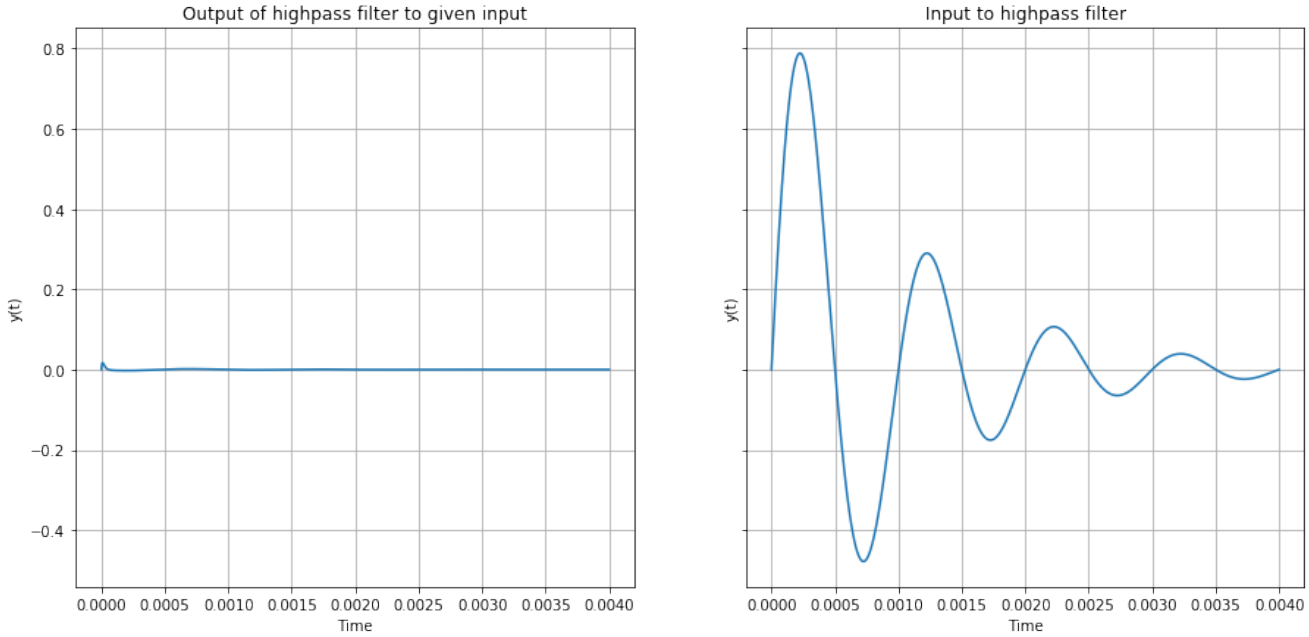
In this, 2 exponentially decaying waves are taken - with high frequency ($2 \times 10^6 Hz$) and low frequency ($2000 Hz$) are taken, and the outputs obtained are compared.

2.4.1 Low Frequency decaying wave

A decaying wave of frequency 2000 Hz, is given as input to the high pass filter and output obtained is observed. The input given is $\sin(2000\pi t)\, e^{-1000t}$

```
t = np.linspace(0.0,4e-3,100001)      # Time for which output is to be calculated
x = (np.sin(2000*math.pi*t))*np.exp((-10**3)*t)
t,y = get_output_with_lsim(H_h,x,t)    # The transfer function calculated before is
    used here

fig, axes = plt.subplots(1, 2, figsize=(15, 7), sharey = True)
axes[0].plot(t,y)
axes[0].grid()
axes[0].set_title('Output of highpass filter to given input')
axes[0].set_xlabel('Time')
axes[0].set_ylabel('y(t)')

axes[1].plot(t,x)
axes[1].grid()
axes[1].set_title('Input to highpass filter')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('y(t)')
plt.show()
```

The output obtained is:



It is seen that, the output is almost 0. This is because, as the system is a high pass filter, and the input has low frequency, the output will have almost 0 amplitude as the low frequency is highly attenuated. Thus, when the magnitude is high initially, a small peak is observed, but afterwards, the output is almost 0.

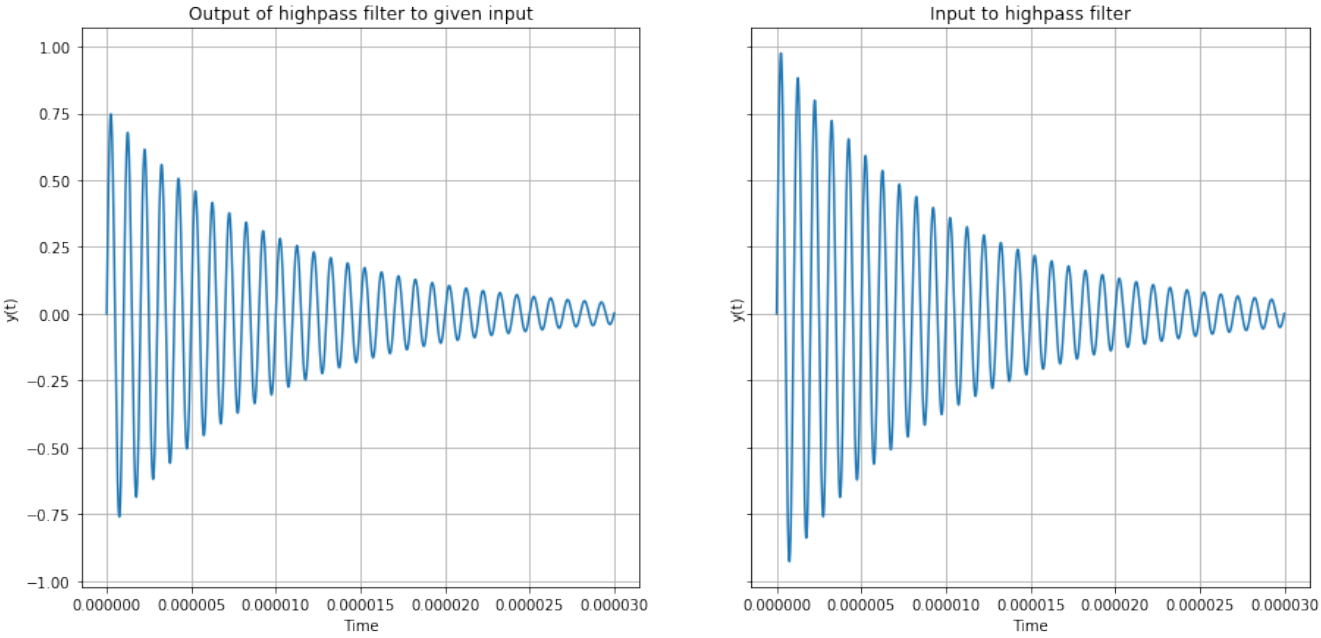
2.4.2 High Frequency decaying wave

Here, a high frequency decaying wave is given as input and the output plot obtained is observed. The input given is $\sin\left(2 \times 10^6 \pi t\right) e^{-10^5 t}$

```
t = np.linspace(0.0,3e-5,100001)      # Time for which output is to be calculated
x = (np.sin(2*(10**6)*math.pi*t))*np.exp((-10**5)*t)
t,y = get_output_with_lsim(H_h,x,t)    # The transfer function calculated before is
    used here

fig, axes = plt.subplots(1, 2, figsize=(15, 7), sharey = True)
axes[0].plot(t,y)
```

```
axes[0].grid()
axes[0].set_title('Output␣of␣highpass␣filter␣to␣given␣input')
axes[0].set_xlabel('Time')
axes[0].set_ylabel('y(t)')

axes[1].plot(t,x)
axes[1].grid()
axes[1].set_title('Input␣to␣highpass␣filter')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('y(t)')
plt.show()
```

Here, the output obtained is:



Thus, compared to the last one, it is seen that, the amplitude is not nearly as attenuated as before (only a small amount due to the fact that the transfer function was not exactly 1 at high frequencies). The only way in which the amplitude of output decreases is when the the input amplitude also decreased.

3 Conclusion

1. From the step-response, of the low pass filter, it is seen that, at steady state, output is a constant value (at a very low attenuation). This is because, low pass filter will pass DC signals , and at time $t > 0$, step input is a DC input. At $t = 0$, though the rise is gradual, as the capacitors themselves take time to charge, and once they have done, the output becomes DC.

2. For the other signal input to the filter, it is seen that, the low pass filter passes only the low frequency component and almost completely attenuates the high frequency part, as expected. Also, from the Bode plots of the transfer function of the low pass filter, it is observed that as the low frequency component of the input (1000 Hz) is less than the 3dB bandwidth of the system it passes with little attenuation (a gain of around 0.8), whereas for that of the high frequency component (10^6 Hz), which is higher than the 3dB bandwidth of around 10^5, gain is almost 10^{-2}, and hence is not visible in the graph.

3. From the step-response of the high pass filter, it is seen that at $t > 0$, output is 0. This is because, at these instants, input is a DC value and the high pass filter will attenuate this, and thus output will be almost 0. At $t = 0$, though, there is a peak for the output. This is because, at this point of discontinuity in the input, as seen from the frequency domain, a lot of high frequency components would be there, hence these would be passed, and so, an output is observed only at this point. These high frequency components are due to the discontinuity, which cannot be replicated using the fourier transform as the fourier transform decomposes the function into a bunch of continuous time sinusoids, hence, Gibb's phenomenon occurs at the discontinuity, causing the initial peak in the step response.

4. When a regular input with 2 frequency components is given, it is seen that the system passes only the high frequency component (10^6 Hz), with a slightly lesser gain than 1, which is higher than the 3dB bandwidth of the system while blocking the low frequency component (1000 Hz), which is lesser than the 3dB bandwidth of the system at around 10^5 Hz and therefore has a gain of almost 10^{-4}.

5. For a low frequency decaying input, output is almost zero, as the low frequency wave will be highly attenuated by the system, whereas for a high frequency decaying input, the attenuation is minimal, as the system has a very low attenuation for the high frequency signals, and therefore the output will more or less follow the input.