Programming Exam 2

November 14, 2023

1 Programming Exam 2

In the lecture, we have been studying how sound waves travel through rooms. We know that a direct wave travels straight from the source to our ears, but some of the sound wave bounce off objects within the room before they reach our ears. The latter waves are called reflections and there are two types: early and late. If we want to design a filter that can model a room, we need to account for all three of these sound waves. In this programming exam, our task is to design and implement what is called Schroeder's reverberator. This reverberator is a pretty good representation of how sound reverberates in a room, but it is not perfect. It is created by putting 4 plain reverberators in paraller and feeding that output to 2 all pass reverberators in series.

1.1 (a) Plain and Allpass Reverberators

In the first part of this programming exam, we will define our plain and all pass reverberators. After this section, we will arrange these reverberators into 2 instances of Schroeder's reverberator. In this section, we also define a sinusoidal signal to test our reverberators. Once we know that everything is working correctly, we can move on to the next part.

1.1.1 (a-i) Define Input Signal

In this section we are defining an audio signal that contains two frequencies, and we filter this audio with each of our reverberators. We will have a total of 3 seconds of audio, and we should be able to hear a clear two tone signal when we play the output as an audio signal.

[2]:
```python
import scipy.signal as spsig
import scipy.io.wavfile
import numpy as np
import matplotlib.pyplot as plt
# part a-i Define input signal
f1 = 240
f2 = 600
fs = 8000
n=np.arange(0,24000,1)
#define input signal
x = np.cos((f1/fs)*2*np.pi*n) + np.cos((f2/fs)*2*np.pi*n)

#normalize signal
x=x/abs(x.max())*0.99
scipy.io.wavfile.write("sines_input.wav", fs, x.astype('float32'))
```

When I listen to the sinusoidal audio I created it sounds like 2 frequencies mixed together. This is what I expected because the signal we defined is the summation of 2 sinusoidal functions. Each function has a specific frequency which does not change. When we add them together, we hear a constant tone of 2 frequencies mixed together. This is the audio sound we will pass through our reverberators to ensure they are working properly.

1.1.2 (a-ii) Plain Reverberator

Now we will design the plain reverberator. This reverberator simulates the early reflections in a room. When we put a few of these in parallel, they will be able to account for all the early reflections for a sound wave. We will again be using our filter395 function whcih we defined in a previous homework assignment.

[13]:
```
#part a-ii
#Filter 395 function which we use to define reverberators
def filter395(x, b, a):
    L = len(b)                                #length of feedforward vector
    M = len(a)                                #length of feedback vector
    v = np.zeros(L)                           #initialize all the state vectors

    if a[0]!= 1:
        b = b/a[0]
        a = a/a[0]

    w = np.zeros(M)
    y = np.zeros(len(x))
    a_end = a[1:]
    for i in range(len(x)):                   #loop over the input
        w_end = w[1:]
        v[0] = x[i]                           #get current input
        w[0] = (b@v) - (a_end@w_end)          #calculate internal state for output
        y[i] = w[0]                           #assign the output
        v = np.roll(v,1)                      #shift the internal states
        w = np.roll(w,1)
    return y

#define coefficient vectors
bp = [0] * 1
bp[0] = 1
ap = np.zeros(101)
ap[0] = 1
ap[100] = -0.93
#run input x through filter
yp = filter395(x, bp, ap)
yp=yp/abs(yp.max())*0.99
scipy.io.wavfile.write("sines_pr.wav", fs, yp.astype('float32'))
```

When we listen to the output of our plain reverberator, it sounds like the higher frequency has been attenuated, while the lower frequency remains. At the start of the audio, it sounds like the higher ferquency stutters for a little and then you can barely hear it. The plain reverberator should output some delayed, attenuated versions of the input signal. The lower frequency is not affected as much, because this reverberator acts like a comb filter and f1 lies in a peak. This filter is working exactly how we want it to.

1.1.3 (a-iii) Allpass Reverberator

Now that we have taken care of the early reflections, we need to adress the late reflections. In order to do this, we will design an allpass reverberator. It is very similar to the plain reverberator, but this filter does not affect the magnitude of the soundwave, it affects the phase. In reality, the early reflections blend together because they are coming in at different times. Even though the magnitude response of this reverberator is flat, the transient response decays exponentially. Because of this, we still have the echoes, but they come out smoother and smeared together. This smearing effect helps us simulate a better reverberation effect.

[30]:
```
#part a-iii
ba = np.zeros(101)
ba[0] = -0.93
ba[100] = 1
aa = np.zeros(101)
aa[0] = 1
aa[100] = -0.93
#run the input through filter
ya = filter395(x, ba, aa)
ya=ya/abs(ya.max())*0.99
scipy.io.wavfile.write("sines_ar.wav", fs, ya.astype('float32'))
```

An allpass reverberator does not affect the magnitude of the signal, but changes the phase. The goal of the allpass reverberator is to simulate the late reflections of the input sound. When we listen to the output of the allpass reverberator, again at the start is where we hear its effects. The sound volume starts loud, but then you can hear it quickly attenuate and then quickly return to its original magnitude. In this case, the higher frequency is unaffected, and the stuttering occurs for the lower frequency. Even though the magnitude response is flat, the transients still follow the same pattern. The lower frequency is affected because the locations of the poles and zeros are switched comopared to the last section. Compared to the plain reverberator, the delays seem longer and smoother. Compared to the original input, it sounds similar, except we can hear the delays at the start of the audio. The reason we only hear it at the start is because for a reverberator, we are only interested in the transients and not the steady state as much.

1.2 (b) Schroeder's Reverberator 1

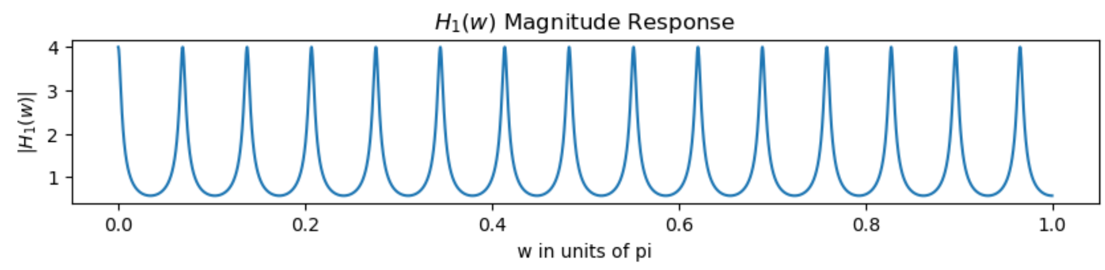
Now that we have all the pieces to implement Schroeder's reverberator. Like I stated early, we will put 4 plain reverberators in parallel which are followed by 2 allpass reverberators in series, for a total of 6 reverberators. Each reverberator will have its own specifications which we will have to figure out. We will be plotting the magnitude and impulse responses for each of the reverberators. We will also compute the pole radii and the 60 dB effective time constant in samples and seconds. Finally, we will filter the happy.wav audio through our Schroeder's reverberator and observe the outcome.

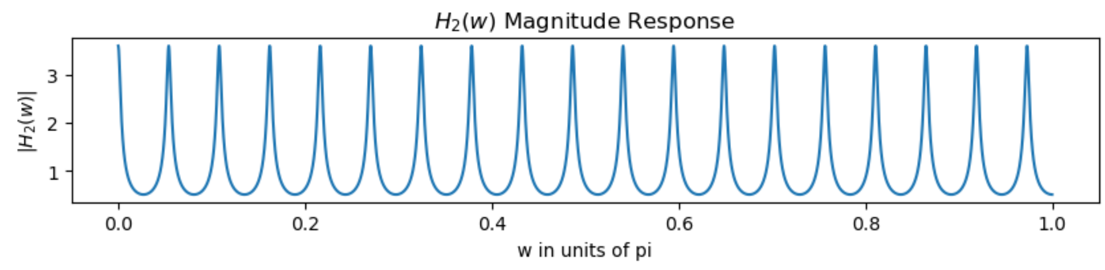
[32]:
```
# part b
#define delays as specified in the textbook
D1 = 29
D2 = 37
D3 = 44
D4 = 50
D5 = 27
D6 = 31
#feedback and forward coefficients for first reverberator
a1 = np.zeros(D1+1)
a1[0] = 1
a1[D1] = -0.75
b1 = [0] * 1
b1[0] = 1
#feedback and forward coefficients for second reverberator
a2 = np.zeros(D2+1)
a2[0] = 1
a2[D2] = -0.75
b2 = [0] * 1
b2[0] = 0.9
#feedback and forward coefficients for third reverberator
a3 = np.zeros(D3+1)
a3[0] = 1
a3[D3] = -0.75
b3 = [0] * 1
b3[0] = 0.8
#feedback and forward coefficients for fourth reverberator
a4 = np.zeros(D4+1)
a4[0] = 1
a4[D4] = -0.75
b4 = [0] * 1
b4[0] = 0.7
#feedback and forward coefficients for fith reverberator
a5 = np.zeros(D5+1)
a5[0] = 1
a5[D5] = -0.75
b5 = np.zeros(D5+1)
b5[0] = -0.75
b5[D5] = 1
#feedback and forward coefficients for sixth reverberator
a6 = np.zeros(D6+1)
a6[0] = 1
a6[D6] = -0.75
b6 = np.zeros(D6+1)
b6[0] = -0.75
b6[D6] = 1
```

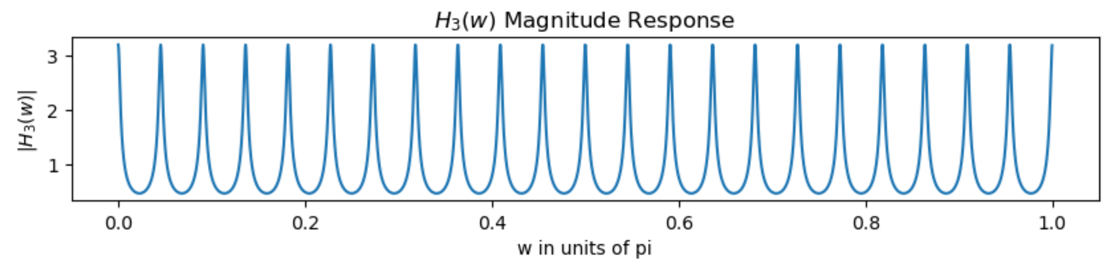
1.2.1 (b-i) Magnitude Response

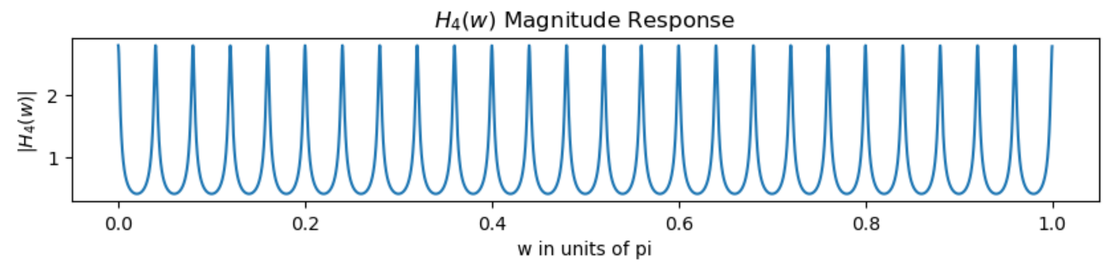
Now that we have defined the feedback and feed-forward coefficient vectors for each of our reverberators, we want to plot the magnitude response for each reverberator. We know that the plain reverberator is essentially an IIR comb filter, so we expect to see these filters contain several peaks. We know that the all pass reverberator magnitude response should just be 1 and pass all the frequencies.

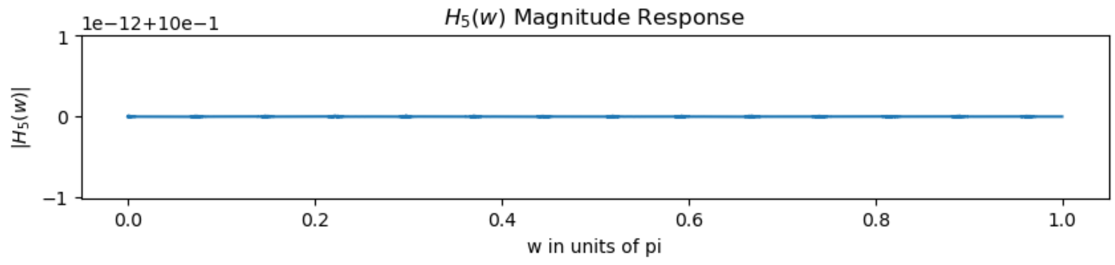
[33]:
```
#part b-i
#Compute magnitude responses for each filter
w1,H1=spsig.freqz(b1,a1,5196)
w2,H2=spsig.freqz(b2,a2,5196)
w3,H3=spsig.freqz(b3,a3,5196)
w4,H4=spsig.freqz(b4,a4,5196)
w5,H5=spsig.freqz(b5,a5,5196)
w6,H6=spsig.freqz(b6,a6,5196)

fig, axs = plt.subplots(6, figsize=(10,16))

axs[0].plot(w1/np.pi, abs(H1))
axs[0].set_title('$H_1(w)$ Magnitude Response' )
axs[0].set(ylabel = '$|H_1(w)|$', xlabel = 'w in units of pi')

axs[1].plot(w2/np.pi, abs(H2))
axs[1].set_title('$H_2(w)$ Magnitude Response')
axs[1].set(ylabel = '$|H_2(w)|$', xlabel = 'w in units of pi')

axs[2].plot(w3/np.pi, abs(H3))
axs[2].set_title('$H_3(w)$ Magnitude Response')
axs[2].set(ylabel = '$|H_3(w)|$', xlabel = 'w in units of pi')

axs[3].plot(w4/np.pi, abs(H4))
axs[3].set_title('$H_4(w)$ Magnitude Response')
axs[3].set(ylabel = '$|H_4(w)|$', xlabel = 'w in units of pi')

axs[4].plot(w5/np.pi, abs(H5))
axs[4].set_title('$H_5(w)$ Magnitude Response' )
axs[4].set(ylabel = '$|H_5(w)|$', xlabel = 'w in units of pi')

axs[5].plot(w6/np.pi, abs(H6))
axs[5].set_title('$H_6(w)$ Magnitude Response' )
axs[5].set(ylabel = '$|H_6(w)|$', xlabel = 'w in units of pi')

plt.subplots_adjust(wspace=0.4,hspace=0.8)
```

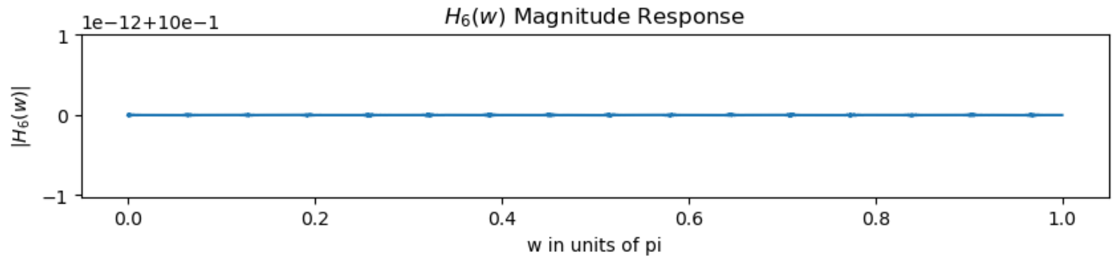

$H_1(w)$ Magnitude Response
$|H_1(w)|$
4
3
2
1
0.0
0.2
0.4
0.6
0.8
1.0
w in units of pi


$H_2(w)$ Magnitude Response
$|H_2(w)|$
3
2
1
0.0
0.2
0.4
0.6
0.8
1.0
w in units of pi


$H_3(w)$ Magnitude Response
$|H_3(w)|$
3
2
1
0.0
0.2
0.4
0.6
0.8
1.0
w in units of pi


$H_4(w)$ Magnitude Response
$|H_4(w)|$
2
1
0.0
0.2
0.4
0.6
0.8
1.0
w in units of pi


$H_5(w)$ Magnitude Response
1e−12+10e−1
$|H_5(w)|$
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0
w in units of pi


$H_6(w)$ Magnitude Response
1e−12+10e−1
$|H_6(w)|$
1
0
−1
0.0
0.2
0.4
0.6
0.8
1.0
w in units of pi

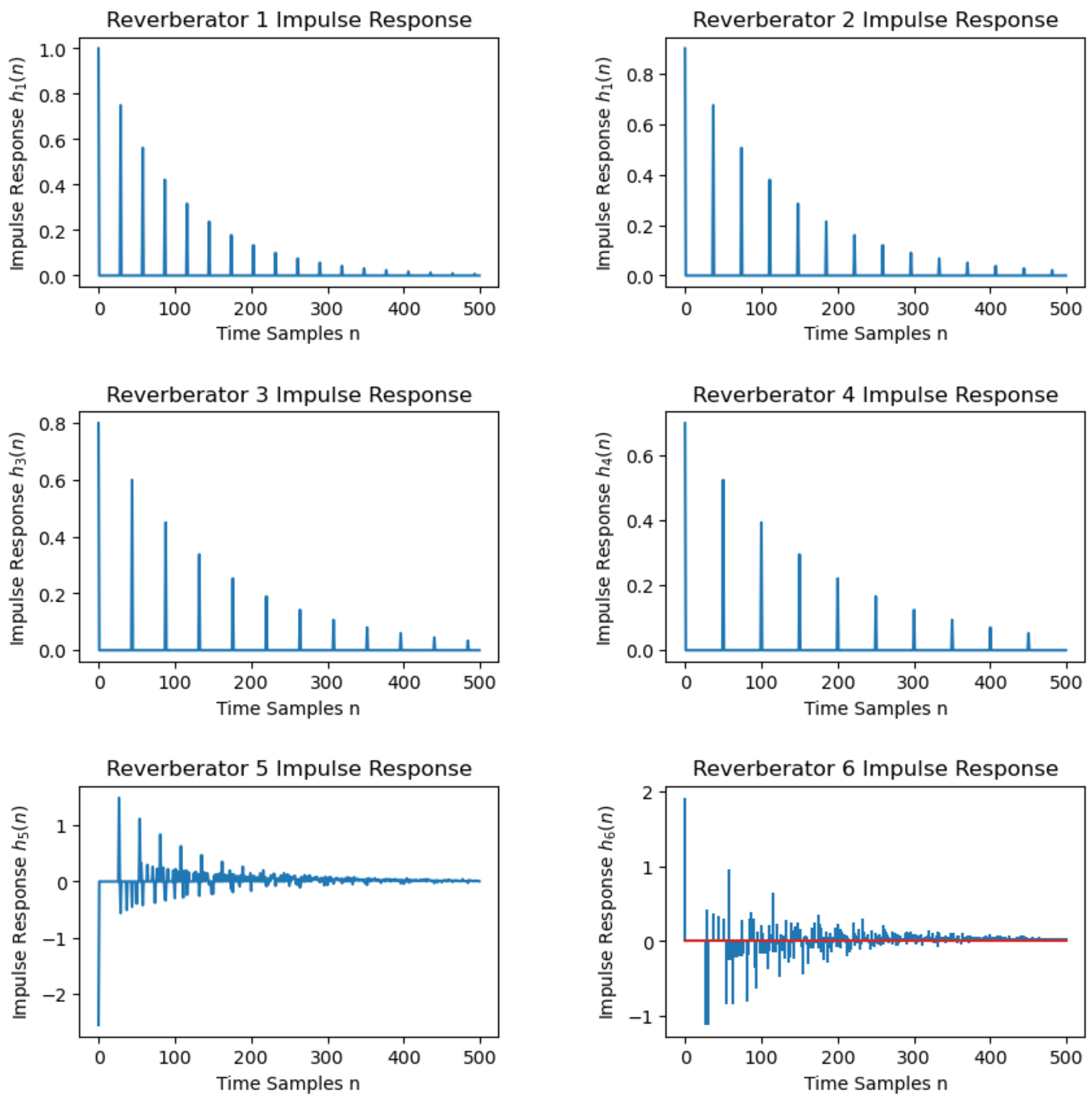
For the plain reverberators, we get exactly what we expect. We see each of them have the magnitude response of an IIR comb filter, with each of them having a different number of peaks corresponding to the delays assigned to each of them. The magnitude response of the first filter is not attenuated at all, but the following three filters are attenuated more and more. For the allpass reverberators, we achieved the desired flat magnitude response, but for whatever reason, the value is at 0 instead of 1. The allpass reverberators should just pass the entire frequency band.

1.2.2 (b-ii) Impulse Response

Now we want to see how our system responds when we input an impulse. From the impulse response, we can ascertain characteristics about the filter and how it behaves.

[34]:
```python
#part b-ii
#define impulse response
delta = np.zeros(500)
delta[0] = 1
n=np.arange(0,500,1)
y1 = filter395(delta, b1, a1)
y2 = filter395(delta, b2, a2)
y3 = filter395(delta, b3, a3)
y4 = filter395(delta, b4, a4)
#add all these impulses to input to first allpass
x5 = y1+y2+y3+y4
y5 = filter395(x5, b5, a5)
y6 = filter395(y5, b6, a6)

fig, axs = plt.subplots(3,2, figsize=(10,10))

axs[0,0].plot(n, y1)
axs[0,0].set_title('Reverberator 1 Impulse Response' )
axs[0,0].set(ylabel = 'Impulse Response $h_1(n)$', xlabel = 'Time Samples n')

axs[0,1].plot(n, y2)
axs[0,1].set_title('Reverberator 2 Impulse Response' )
axs[0,1].set(ylabel = 'Impulse Response $h_1(n)$', xlabel = 'Time Samples n')

axs[1,0].plot(n, y3)
axs[1,0].set_title('Reverberator 3 Impulse Response' )
axs[1,0].set(ylabel = 'Impulse Response $h_3(n)$', xlabel = 'Time Samples n')

axs[1,1].plot(n, y4)
axs[1,1].set_title('Reverberator 4 Impulse Response' )
axs[1,1].set(ylabel = 'Impulse Response $h_4(n)$', xlabel = 'Time Samples n')

axs[2,0].plot(n, y5)
```

```
axs[2,0].set_title('Reverberator 5 Impulse Response' )
axs[2,0].set(ylabel = 'Impulse Response $h_5(n)$', xlabel = 'Time Samples n')

axs[2,1].stem(n, y6,markerfmt=' ')
axs[2,1].set_title('Reverberator 6 Impulse Response' )
axs[2,1].set(ylabel = 'Impulse Response $h_6(n)$', xlabel = 'Time Samples n')

plt.subplots_adjust(wspace=0.4,hspace=.5)
```



These impulse responses are consistent with our expectations. The first four reverberators are plain reverberators, which means that the input sound signal will be delayed and attenuated. When we input an impulse, this is clear to see. At time zero, the value starts at 1 and then as time passes, the impulse is delayed and attenuated. The only difference between the first 4 reverberator impulse responses is the gain factor at the front. We chose for each successive reverberator to have a little bit more attenuation than the previous reverberator. For the all pass reverberators, we see the same pattern, except there the delays are smeared. This happens because the allpass reverberator changes the phase of the input signal and moves everything around. Overall, our y6 perfectly matches Figure 8.2.19 in the book.

1.2.3 (b-iii) Define Radii of Poles

We are interested in the pole zero plot, because this will help us understand the time constant. The radius of the poles give us insights. If the radius is close to 1 or close to the unit circle, the effective time constant will be a lot longer. If the effective time constant is large, it can take a long time for the filter response to settle down.

[59]:
```
# part b-iii
rho1 = pow(0.75, 1/D1)
print(f'Reverberator 1 Pole Radius = {rho1}')

rho2 = pow(0.75, 1/D2)
print(f'Reverberator 2 Pole Radius = {rho2}')

rho3 = pow(0.75, 1/D3)
print(f'Reverberator 3 Pole Radius = {rho3}')

rho4 = pow(0.75, 1/D4)
print(f'Reverberator 4 Pole Radius = {rho4}')

rho5 = pow(0.75, 1/D5)
print(f'Reverberator 5 Pole Radius = {rho5}')

rho6 = pow(0.75, 1/D6)
print(f'Reverberator 6 Pole Radius = {rho6}')
```

```
Reverberator 1 Pole Radius = 0.9901289701456767
Reverberator 2 Pole Radius = 0.992254957463671
Reverberator 3 Pole Radius = 0.9934830987922074
Reverberator 4 Pole Radius = 0.9942628790464048
Reverberator 5 Pole Radius = 0.9894016707183191
Reverberator 6 Pole Radius = 0.9907628600762322
```

Each of these poles is pretty close to the unit circle, which will make the value of the effective time constant larger. We can get the value of the radius by using the equation p = a^(1/D) so the only thing affecting the radius is the amount of delays used in each reverberator. If we had larger delays, the radii would be even closer to the unit circle.

1.2.4 (b-iv) Define 60 dB Effective Time Constant in Samples

Now that we know the radius of the poles, we want to find the 60 dB effective time constant. The effective time constant in sample will always be the same, no matter the input, but the effective time constant in seconds will depend on the sampling frequency of the input signal. As always, the effective time constant is dictated by the radius of the largest pole.

[63]:
```
#part b-iv

neff_samp = np.log(0.001)/np.log(rho4)          #0.001 because that is the␣
 ↪value of epsilon for 60dB neff
print(f'The 60 dB effective time constant in samples is {np.ceil(neff_samp):.
 ↪0f} samples.')
```

```
The 60 dB effective time constant in samples is 1201 samples.
```

The effective time constant can also be found by the equation: neff = ln(e)/ln(a) * D, where D is the largest delay. Reverberator 4 has the largest delay, so it will have the longest effective time constant. By the time that reverberator 4's transients die out, all the other reverberator's transients will already have died out as well. We rounded our effective time constant up because you can't have partial samples. The transients will have died down at n = 1201 samples.

1.2.5 (b-v) Define 60 dB Effective Time Constant in Seconds

Now that we have defined the 60 dB effective time constant in samples, we want to see how long it would take in seconds for a particular signal. In this case, we will be using the happy.wav audio provided to us. The effective time constant in seconds is easily found by multiplying neff with the sampling period.

[64]:
```
# part b-v
fs,happy=scipy.io.wavfile.read('Happy.wav')
neff_sec = neff_samp * (1/fs)
print(f'The 60 dB effective time constant in seconds for the happy.wav signal␣
 ↪is {neff_sec:.2f} seconds.')
```

```
22050
The 60 dB effective time constant in seconds for the happy.wav signal is 0.05
seconds.
```

The effective time constant is very short compared to the length of happy. It only takes 50 miliseconds for the transients of our reverberator to die out. Compared to the happy signal, we will only be able to hear the transients for a very short amount of time.

1.2.6 (b-vi) Filter happy Signal through Reverberator

Finally, we will put the happy signal through our reverberator and listen to the output. Hopefully, the output will sound like the sound signal is played in a room. We will discuss the effects of our reverberator and explain what is happening.

[65]:
```
# part b-vi
y1 = filter395(happy, b1, a1)
y2 = filter395(happy, b2, a2)
y3 = filter395(happy, b3, a3)
y4 = filter395(happy, b4, a4)
```

```
x5 = y1+y2+y3+y4
y5 = filter395(x5, b5, a5)
happy6 = filter395(y5, b6, a6)
happy6=happy6/abs(happy6).max()*happy.max()
#write out to audio file
scipy.io.wavfile.write("happy_schroeder1.wav", fs, happy6.astype('int16'))
```

The reverberator should slightly muffle and diffuse the sounds within the input signal. It will make the input audio sound more natural and full compared to the original audio. Schroeder's reverberator can be used to model the behaviors of sound in a closed environment like a concert hall. When we listen to the output of our reverberator, we can slightly hear the sounds diffuse more evenly and reverberate. The reverberator accounts for the reflections bouncing off the walls and back into our ears. In the original signal, it sounds like the audio is going directly into the mic. After we filter it through our reverberator, we can hear the echos if we listen closely.

1.3 (c) Schroeder's Reverberator 2

Next, we will define a more practical version of Schroeder's reverberator. The reveberator defined in the book employs delays that are not physically realistic for most situations. In this section we will again look at the radii of the poles, the 60 dB effective time constant in samples and seconds, the impulse response, and the output of our reverberator when happy is the input. Unlike the last problem, we will also look at how long it takes for our filter to process the happy signal.

[3]:
```
D1 = 754
D2 = 905
D3 = 1015
D4 = 1542
D5 = 102
D6 = 42
#feedback and forward coefficients for first reverberator
a1 = np.zeros(D1+1)
a1[0] = 1
a1[D1] = -0.826
b1 = [0] * 1
b1[0] = 0.826
#feedback and forward coefficients for second reverberator
a2 = np.zeros(D2+1)
a2[0] = 1
a2[D2] = -0.802
b2 = [0] * 1
b2[0] = 0.802
#feedback and forward coefficients for third reverberator
a3 = np.zeros(D3+1)
a3[0] = 1
a3[D3] = -0.778
b3 = [0] * 1
b3[0] = 0.778
#feedback and forward coefficients for fourth reverberator
```

```
a4 = np.zeros(D4+1)
a4[0] = 1
a4[D4] = -0.758
b4 = [0] * 1
b4[0] = 0.758
#feedback and forward coefficients for fith reverberator
a5 = np.zeros(D5+1)
a5[0] = 1
a5[D5] = -0.7
b5 = np.zeros(D5+1)
b5[0] = -0.7
b5[D5] = 1
#feedback and forward coefficients for sixth reverberator
a6 = np.zeros(D6+1)
a6[0] = 1
a6[D6] = -0.7
b6 = np.zeros(D6+1)
b6[0] = -0.7
b6[D6] = 1
```

1.3.1 (c-i) Define Radii of Poles

Now that we have defined the feedback and feedforward coefficients for the reverberators, we will calculate and display the radii for each reverberator's poles. From the radii of the poles, we will be able to interpret how long the effective time constant will be.

[4]:
```
# part c-i
rho1 = pow(0.826, 1/D1)
print(f'Reverberator 1 Pole Radius = {rho1}')

rho2 = pow(0.802, 1/D2)
print(f'Reverberator 2 Pole Radius = {rho2}')

rho3 = pow(0.778, 1/D3)
print(f'Reverberator 3 Pole Radius = {rho3}')

rho4 = pow(0.758, 1/D4)
print(f'Reverberator 4 Pole Radius = {rho4}')

rho5 = pow(0.7, 1/D5)
print(f'Reverberator 5 Pole Radius = {rho5}')

rho6 = pow(0.7, 1/D6)
print(f'Reverberator 6 Pole Radius = {rho6}')
```

```
Reverberator 1 Pole Radius = 0.9997465036138081
Reverberator 2 Pole Radius = 0.9997562212425004
Reverberator 3 Pole Radius = 0.9997527116105654
Reverberator 4 Pole Radius = 0.99982033268339
Reverberator 5 Pole Radius = 0.9965092935552172
Reverberator 6 Pole Radius = 0.991543696816334
```

Because the delays are much longer for this reverberator, the poles end up being a lot closer to the unit circle. The larger the delay, the closer to the unit circle the poles will be. The radii of the poles was easy to find, when we used the simple equation: p = a^(1/D). When the poles are closer to the unit circle, the peaks are sharper, but the effective time constant will be larger.

1.3.2 (c-ii) Define 60 dB Effective Time Constant in Samples

Using our knowledge of the largest pole radius, we can easily calculate the 60 dB effective time constant for our reverberator. The effective time constant defines the amount of samples required for the transients to die out. In the reverberator's case, we are more interested in the transients than the steady state. This is because we are trying to simulate the echos of a sound wave that bounce off walls. Each time the signal bounces, it gets attenuated more, and these reflections are the main purpose for a reverberator.

[5]:
```
#part c-ii
neff_samp = np.log(0.001)/np.log(rho4)           #0.001 because that is the␣
  ↪value of epsilon for 60dB neff
print(f'The 60 dB effective time constant in samples is {np.ceil(neff_samp):.
  ↪0f} samples.')
```

```
The 60 dB effective time constant in samples is 38445 samples.
```

This effective time constant is much larger than our first reverberator's effective time constant. This means that we will be able to hear the reflections much better for this reverberator.

1.3.3 (c-iii) Define 60 dB Effective Time Constant in Seconds

We want to know how long in seconds it takes for the transients to die down. This is how long the reverberations will last in seconds. We will determine the effective time constant in seconds for the happy audio signal.

[6]:
```
# part c-iii
fs,happy=scipy.io.wavfile.read('Happy.wav')
neff_sec = neff_samp * (1/fs)
print(f'The 60 dB effective time constant in seconds for the happy.wav signal␣
  ↪is {neff_sec:.2f} seconds.')
```

```
The 60 dB effective time constant in seconds for the happy.wav signal is 1.74
seconds.
```

Again the effective time constant for this reverberator is much larger than the effective time constant for our first reverberator. The reverberations should last about 1.74 seconds, which is almost one third of the original audio signal. We will be able to clearly hear the effects of our reverberator.

1.3.4 (c-iv) Define Impulse Responses:

Again, we are interested in how our filter will respond when the input is a simple impulse. Just by looking at a systems impulse response, we can tell how the system will behave for almost any input signal. This time, instead of plotting the impulse responses, we will write them out to audio files and listen to the effects of each reverberator. The output of the last reverberator corresponds to the output of the entire Schroeder's reverberator.

[16]:
```
#part c-iv
delta = np.zeros(len(happy))
delta[0] = 1
n=np.arange(0,len(happy),1)

y1 = filter395(delta, b1, a1)
y2 = filter395(delta, b2, a2)
y3 = filter395(delta, b3, a3)
y4 = filter395(delta, b4, a4)
#add all these impulses to input to first allpass
x5 = y1+y2+y3+y4
y5 = filter395(x5, b5, a5)
y6 = filter395(y5, b6, a6)
#normalize
y1=y1/max(y1)*max(happy)
y2=y2/max(y2)*max(happy)
y3=y3/max(y3)*max(happy)
y4=y4/max(y4)*max(happy)
y5=y5/max(y5)*max(happy)
y6=y6/max(y6)*max(happy)
#write out to audio file
scipy.io.wavfile.write("y1.wav", fs, y1.astype('int16'))
scipy.io.wavfile.write("y2.wav", fs, y2.astype('int16'))
scipy.io.wavfile.write("y3.wav", fs, y3.astype('int16'))
scipy.io.wavfile.write("y4.wav", fs, y4.astype('int16'))
scipy.io.wavfile.write("y5.wav", fs, y5.astype('int16'))
scipy.io.wavfile.write("y6.wav", fs, y6.astype('int16'))
```

The output of the first four reverberators sounds kinda like a short static. To me it sounds a lot like when a quarter is spinning and slowing down and right before it stops completely it makes a sound similar to the output of these reverberators. The only difference between theses reverberators is that the delays are successively larger and it is very easy to hear the difference. The outputs of the allpass reverberators sound to me like a spray bottle spraying water. All the sounds are mixed together and diffused more evenly. This is exactly what I expect for both reverberators.

1.3.5 (c-v) Filter happy Signal through Reverberator

Now we will filter the happy audio signal through our reverberator and observe the effects. I think that we will be able to clearly hear the effects of the reverberator, unlike the first one.

[17]:
```python
#part c-v
y1 = filter395(happy, b1, a1)
y2 = filter395(happy, b2, a2)
y3 = filter395(happy, b3, a3)
y4 = filter395(happy, b4, a4)

x5 = y1+y2+y3+y4
y5 = filter395(x5, b5, a5)
happy6 = filter395(y5, b6, a6)
happy6=happy6/abs(happy6).max()*happy.max()
#write out to audio file
scipy.io.wavfile.write("happy_schroeder2.wav", fs, happy6.astype('int16'))
```

As I suspected, the effect of this reverberator is obvious. Compared to the original audio, it sounds like the signal filtered through our reverberator is played in a room. You can hear the sounds last longer and they sound fuller. This is because the filter outputs delayed attenuated versions of the input sound signal. The filter simulates the sound bouncing around in a room, and after each bounce it gets quieter and quieter. It acts like a really fast echo. Overall, the reverberator was designed correctly and gives us a nice output.

1.3.6 (c-vi) Timing

Finally, for the last part of this assignment, we will discuss the time it takes for our filter to process the happy signal. We want to know if this filter can be used in real time or only on already sampled signals.

[27]:
```python
import time
t = time.time() # start keeping track of time

y1 = filter395(happy, b1, a1)
y2 = filter395(happy, b2, a2)
y3 = filter395(happy, b3, a3)
y4 = filter395(happy, b4, a4)

x5 = y1+y2+y3+y4
y5 = filter395(x5, b5, a5)
happy6 = filter395(y5, b6, a6)
total_time = time.time()-t #stop keeping track of time
print(f'The total processing time of this reverberator is {total_time:.2f}␣
 ↪seconds.')
print(f'The average processing time per sample is {total_time/len(happy):.5f}␣
 ↪seconds.')
```

```
The total processing time of this reverberator is 21.19 seconds.
The average processing time per sample is 0.00016 seconds.
```

By using the time.time function, we were easily able to find the time it took for our reverberator to process the happy signal. It took about 21 seconds for the signal to be processed completely, which means that we would not be able to process this in real time. The happy signal is only about

6 seconds, which is way shorter than the processing time. In order for this filter to work in real time, the processing time must be less than the length of the input signal in seconds.

```
[ ]:
```