EE 415 Project: Subband Coder Project

Branden Akana

Introduction

The purpose of this project was to implement a subband coder for audio files. The method used to do this is by designing a two-channel conjugate quadrature filter (CQF) which aims to compress audio signals such that the size of the signals when saved to disk is minimal, but still retains most of the quality of the original signal.

The project is designed to be made in MATLAB, but this project was made using Python. Several compromises had to be made due to the differences between these two languages, which will be discussed in the next sections. Issues that were encountered during the project will also be discussed.

Playing Audio in Python

The first issue was the matter of playing audio from a file. Playing an array containing a signal as audio is not a feature included by default in Python. This functionality was needed as the signal must be modified through filters before playing it back. To solve this, two additional Python modules, `soundfile`, which is able to read audio files as NumPy arrays which work nicely with plots, filters and FFTs, and `sounddevice`, which is able to play arrays as audio through the speakers, were installed.

By default, playing audio through Python is very loud. This was resolved by simply multiplying the signal by a factor of 0.5 before playing.

CQF Design

The CQF was implemented using a class. The functions for the FIR filters, downsampler and upsampler was provided by the SciPy library. The window method was used to design the FIR filter using the `firwin()` function and the downsampler and upsampler was done using the `resample()` function.

Saving Files and Differences Between Precisions

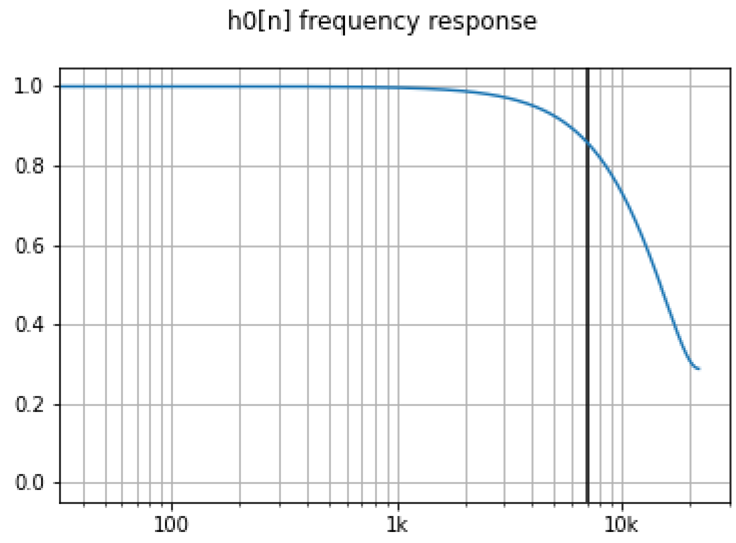
Python does not seem to have a direct equivalent to MATLAB's `fwrite()` and `fread()` functions in a way that also supports modifying precision. The closest method to achieve the same effect was using a combination of NumPy's `ndarray.astype()` and `save()` functions.

As the audio is represented by a list of floats, converting them to an int32, int16, or int8 results in an extreme loss of information (this could possibly work with quantization, but that was not implemented here). The next closest thing was converting them to different precisions of floats: float16, float32, and float64.

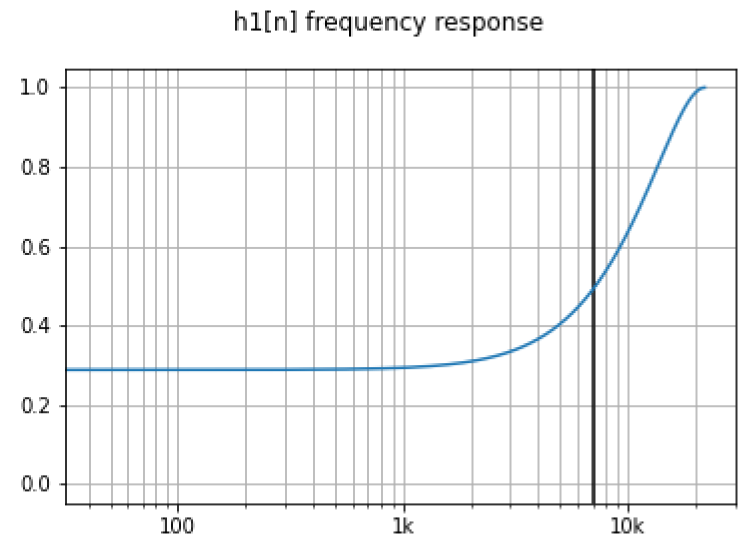
There was no noticable difference between the sound quality of the audio when converting to the different precisions, however the filesize of the saved signals are

file	float16	float32	float64
v0	414 KB	828 KB	1655 KB
v1	414 KB	828 KB	1655 KB

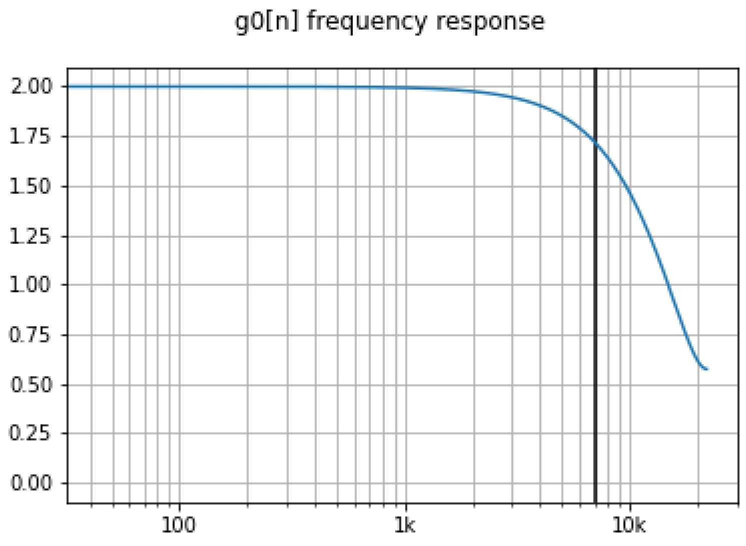
Filter Frequency Responses



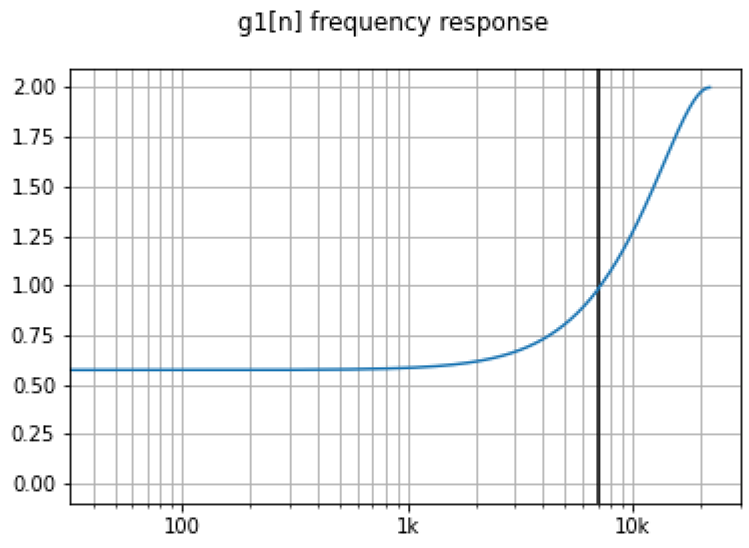
HO.PNG


h1[n] frequency response
1.0
0.8
0.6
0.4
0.2
0.0
100
1k
10k

H1.PNG

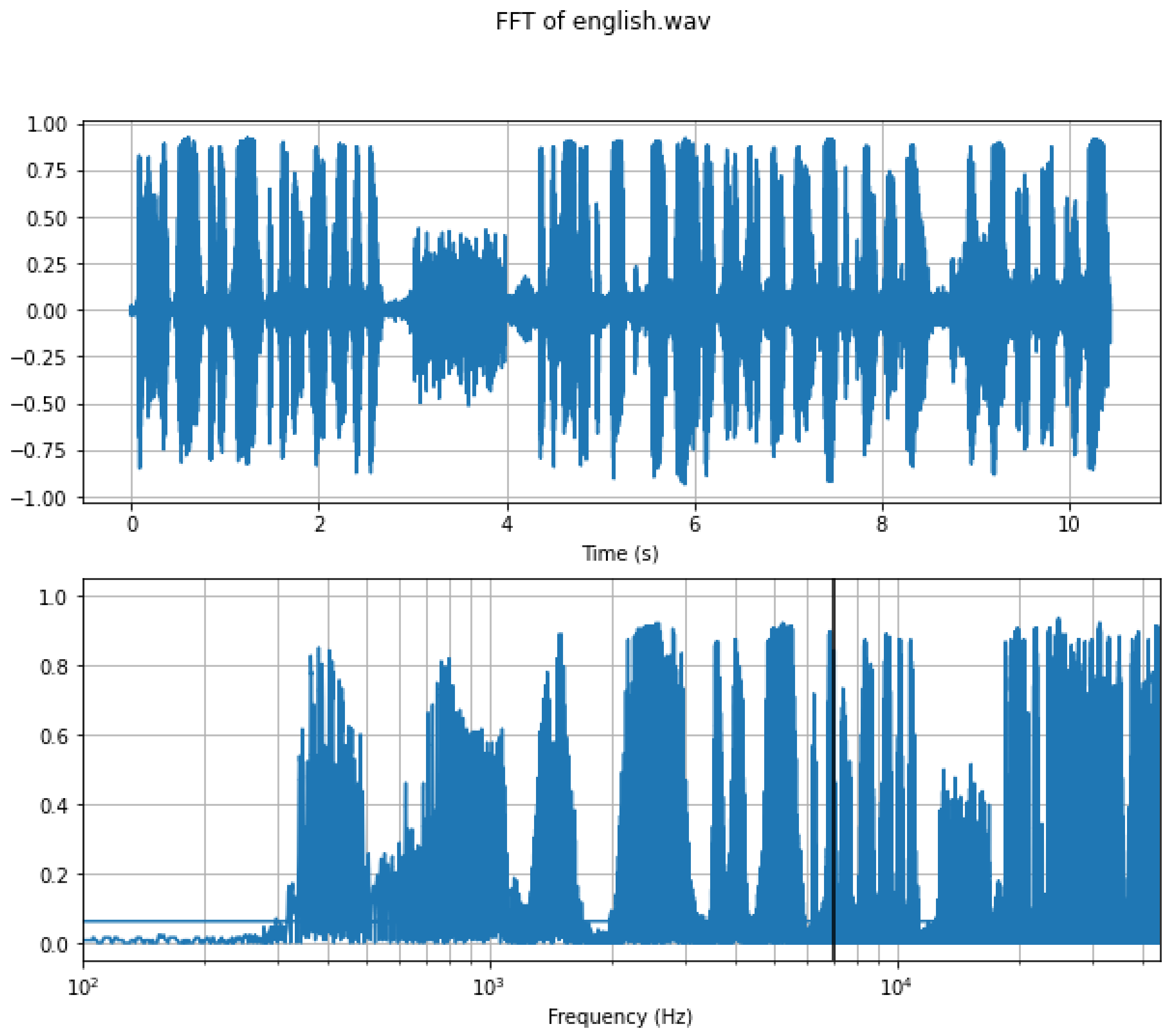

g0[n] frequency response
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
100
1k
10k

GO.PNG

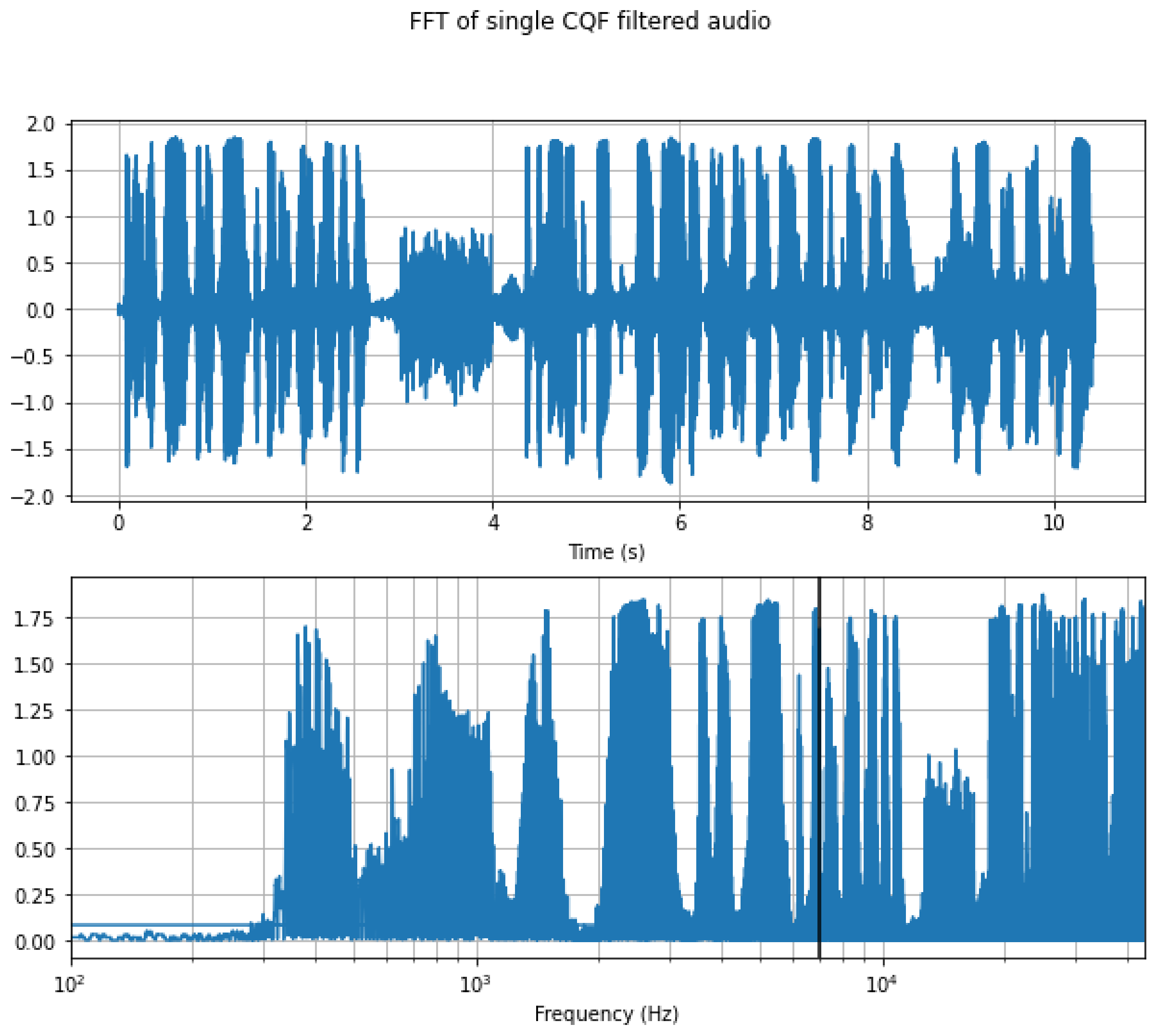


G1.PNG

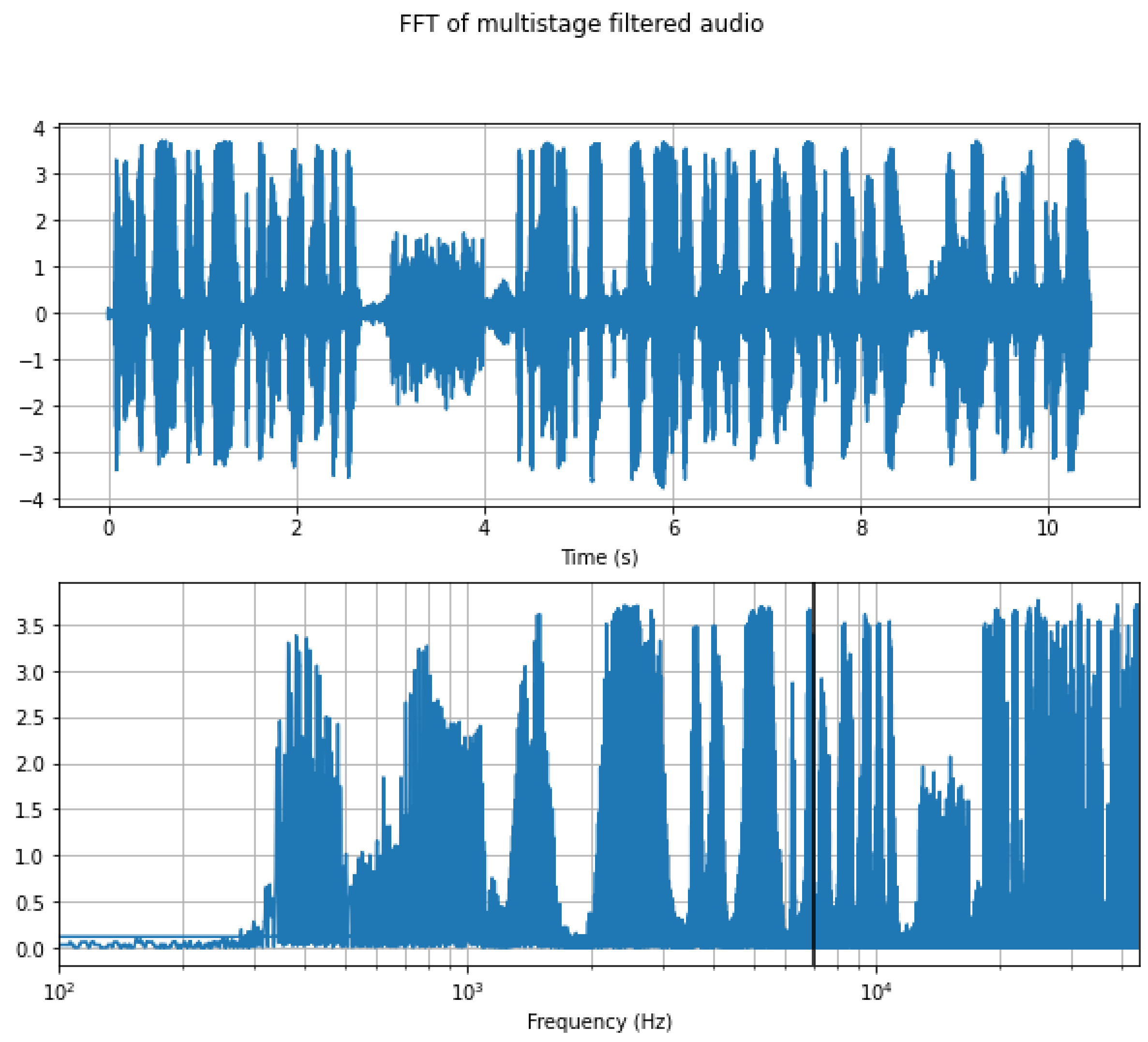
Signal FFTs



Original Signal (english.wav)



Output of Single CQF



Output of Multi CQF

Code

Python 3.8.6 was used to run this code. Most packages used are provided by Anaconda, but additional packages such as `soundfile` and `sounddevice` must be installed manually using `pip install <package>`.

This program expects the audio files to be in the same location as the two scripts.

`cqf.py`

```
# %%



import numpy as np
```

```
from scipy.signal import zpk2sos

from scipy import signal



class CQFFilterbank:

 """A 2-channel conjugate quadrature filter (CQF) filterbank"""


 def __init__(self, fs, M, cutoff):

 # NOTE: here, M must be odd

 self.M = M


 MM = M * 2 + 1 # 2M


 # FIR low-pass filter taps with order 2M

 self.r0_n = signal.firwin(MM, cutoff, fs=fs, pass_zero="lowpass")


 # coefficients of h0[n]

 self.h0_n = signal.minimum_phase(self.r0_n)


 # coefficients of h1[n]
 self.h1_n = [((-1) ** n) * self.h0_n[M - n] for n in
range(len(self.h0_n))]



 # coefficients of g0[n]

 self.g0_n = [2 * self.h0_n[M - n] for n in range(len(self.h0_n))]



 # coefficients of g1[n]

 self.g1_n = [2 * ((-1) ** n) * self.h0_n[n] for n in
range(len(self.h0_n))]



 print("h0 = %s" % self.h0_n)

 print("h1 = %s" % self.h1_n)

 print("g0 = %s" % self.g0_n)

 print("g1 = %s" % self.g1_n)



 # zero-pole-gain representations of our filters

 self.r0_zpk = signal.tf2zpk(self.r0_n, [1])

 self.h0_zpk = signal.tf2zpk(self.h0_n, [1])

 self.h1_zpk = signal.tf2zpk(self.h1_n, [1])

 self.g0_zpk = signal.tf2zpk(self.g0_n, [1])

 self.g1_zpk = signal.tf2zpk(self.g1_n, [1])
 self.r0_zpk = self.r0_zpk[0], self.r0_zpk[1], 1

 self.h0_zpk = self.h0_zpk[0], self.h0_zpk[1], 1

 self.h1_zpk = self.h1_zpk[0], self.h1_zpk[1], 1

 self.g0_zpk = self.g0_zpk[0], self.g0_zpk[1], 1

 self.g1_zpk = self.g1_zpk[0], self.g1_zpk[1], 1


 # # plot zero-poles

 # plot_zplane(*r0_zpk)

 # plot_zplane(*h0_zpk)

 # plot_zplane(*h1_zpk)

 # plot_zplane(*g0_zpk)

 # plot_zplane(*g1_zpk)


 # # plot freq-response of r0[n]

 # w, r0 = signal.freqz(self.r0_n, 1.0)

 # figure()

 # yticks([0, 0.5, 1])

 # xticks([0, 0.5, 1])

 # grid(True)
```

```
 # plot(w / np.pi, abs(r0))



 def analyze(self, x, dtype=np.float64):

 self.v0 = signal.resample(signal.sosfilt(zpk2sos(*self.h0_zpk), x),
len(x) // 2)

 self.v1 = signal.resample(signal.sosfilt(zpk2sos(*self.h1_zpk), x),
len(x) // 2)



 self.v0 = self.v0.astype(dtype)

 self.v1 = self.v1.astype(dtype)



 return self.v0, self.v1



 def analyze_save(self, x, v0_name="v0", v1_name="v1"):

 v0, v1 = self.analyze(x)

 # save files

 print("saving v0 and v1...")

 np.save(v0_name, v0)

 np.save(v1_name, v1)



 return v0, v1
```

```python
 def synthesize(self, v0, v1):

 w0 = signal.sosfilt(zpk2sos(*self.g0_zpk), signal.resample(v0,
len(v0) * 2))

 w1 = signal.sosfilt(zpk2sos(*self.g1_zpk), signal.resample(v1,
len(v1) * 2))

 return w0 + w1



 def synthesize_load(self, v0_name="v0", v1_name="v1"):

 # read files

 print("loading v0 and v1...")

 v0 = np.load("%s.npy" % v0_name)

 v1 = np.load("%s.npy" % v1_name)

 return self.synthesize(v0, v1)
```
Python

`multicqf.py`

```python
# %%



import numpy as np

from scipy.signal import zpk2sos

from scipy import signal
```

```
class CQFFilterbank:

 """A 2-channel conjugate quadrature filter (CQF) filterbank"""



 def __init__(self, fs, M, cutoff):

 # NOTE: here, M must be odd

 self.M = M



 MM = M * 2 + 1 # 2M



 # FIR low-pass filter taps with order 2M

 self.r0_n = signal.firwin(MM, cutoff, fs=fs, pass_zero="lowpass")



 # coefficients of h0[n]

 self.h0_n = signal.minimum_phase(self.r0_n)



 # coefficients of h1[n]

 self.h1_n = [((-1) ** n) * self.h0_n[M - n] for n in
range(len(self.h0_n))]



 # coefficients of g0[n]
 self.g0_n = [2 * self.h0_n[M - n] for n in range(len(self.h0_n))]



 # coefficients of g1[n]

 self.g1_n = [2 * ((-1) ** n) * self.h0_n[n] for n in
range(len(self.h0_n))]



 print("h0 = %s" % self.h0_n)

 print("h1 = %s" % self.h1_n)

 print("g0 = %s" % self.g0_n)

 print("g1 = %s" % self.g1_n)



 # zero-pole-gain representations of our filters

 self.r0_zpk = signal.tf2zpk(self.r0_n, [1])

 self.h0_zpk = signal.tf2zpk(self.h0_n, [1])

 self.h1_zpk = signal.tf2zpk(self.h1_n, [1])

 self.g0_zpk = signal.tf2zpk(self.g0_n, [1])

 self.g1_zpk = signal.tf2zpk(self.g1_n, [1])



 self.r0_zpk = self.r0_zpk[0], self.r0_zpk[1], 1

 self.h0_zpk = self.h0_zpk[0], self.h0_zpk[1], 1

 self.h1_zpk = self.h1_zpk[0], self.h1_zpk[1], 1
 self.g0_zpk = self.g0_zpk[0], self.g0_zpk[1], 1

 self.g1_zpk = self.g1_zpk[0], self.g1_zpk[1], 1



 # # plot zero-poles

 # plot_zplane(*r0_zpk)

 # plot_zplane(*h0_zpk)

 # plot_zplane(*h1_zpk)

 # plot_zplane(*g0_zpk)

 # plot_zplane(*g1_zpk)



 # # plot freq-response of r0[n]

 # w, r0 = signal.freqz(self.r0_n, 1.0)

 # figure()

 # yticks([0, 0.5, 1])

 # xticks([0, 0.5, 1])

 # grid(True)

 # plot(w / np.pi, abs(r0))



 def analyze(self, x, dtype=np.float64):

 self.v0 = signal.resample(signal.sosfilt(zpk2sos(*self.h0_zpk), x),
len(x) // 2)

 self.v1 = signal.resample(signal.sosfilt(zpk2sos(*self.h1_zpk), x),
len(x) // 2)



 self.v0 = self.v0.astype(dtype)

 self.v1 = self.v1.astype(dtype)



 return self.v0, self.v1



 def analyze_save(self, x, v0_name="v0", v1_name="v1"):

 v0, v1 = self.analyze(x)

 # save files

 print("saving v0 and v1...")

 np.save(v0_name, v0)

 np.save(v1_name, v1)



 return v0, v1



 def synthesize(self, v0, v1):

 w0 = signal.sosfilt(zpk2sos(*self.g0_zpk), signal.resample(v0,
len(v0) * 2))
```

```python
 w1 = signal.sosfilt(zpk2sos(*self.g1_zpk), signal.resample(v1,
len(v1) * 2))

 return w0 + w1



 def synthesize_load(self, v0_name="v0", v1_name="v1"):

 # read files

 print("loading v0 and v1...")

 v0 = np.load("%s.npy" % v0_name)

 v1 = np.load("%s.npy" % v1_name)

 return self.synthesize(v0, v1)
```
Python