ECE280 Final
Design Project: Fourier Audio Compression

Sanika Gupte

April 30, 2020

Duke University

Instructor: Professor Vahid Tarokh

I have adhered to the Duke Community Standard in completing this assignment.

sgg15

Contents

1 Introduction

Audio compression a quintessential use for information processing today as it allows for faster data transmission. There are many different audio compression techniques, some rooted in software algorithms, such as the Huffman Coding data compression algorithm, and others rooted in signal processing such as the one described here. To compress audio, properties of the Fourier series will be exploited, most notably, the property of conjugate symmetry of the coefficients for a real signal. Because of this property, any file can be losslessly compressed into a file of half its original size. For a slight loss, files can be compressed even more. Comparisons of loss and differences in signal power versus file size will be made to demonstrate the effect of the Fourier compression algorithm executed. Because of floating point errors, the final signal that is produced results in some noise, so a 5-point moving average filter is applied to smooth out the signal. All coding to implement the compression algorithm is done in python.

2 Methods and Results

2.1 Preliminary Data Processing

The file, broke_leaf.wav was read using the wavfile package from scipy.io. To prevent overflow, the data set was normalized by the maximum value of the data set.

```
fs, x = wavfile.read("Broke_leaf.wav")
x = x/max(x)      #normalize magnitude to prevent overflow
N = len(x)        #N=150,000,001
```

2.2 Compression Algorithm

Part A

To execute the Fourier series compression algorithm, the signal compressed must be a real signal of a finite length. If the length is taken as a period, the signal can be periodically extended to the whole real line using equation 1.

$$\tilde{x}[n] = \sum_{l=-\infty}^{+\infty} x[n - lN] \tag{1}$$

For the audio file undergoing compression, the period, N, is 150,000,001. Because it is assumed that the file is periodically extended, the Fourier series can be applied.

Part B

To calculate the Fourier series for a finite discrete signal, the following formulas are typically used:

$$a_k = \frac{1}{N}\sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}, \text{ for k=0,1,2,3,..., N-1} \tag{2}$$

$$x[n] = \sum_{k=0}^{k=N-1} a_k e^{j\frac{2\pi}{N}kn}, \text{ for n=0,1,2,3,...,N-1} \tag{3}$$

To calculate the coefficients directly from equation 2, an O(n^2) operation needs to be performed. As the size of the data set is very large, 150,000,001, calculating the coefficients is extremely time consuming and can be done much faster using the fast fourier transform (fft) algorithm from the scipy package in python (Discrete Fourier Transform, 2017). Scipy implements fft as shown in equation 4.

$$A_k = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}nk}, \text{ for k=0,1,2,...,N-1} \tag{4}$$

This implementation is linearly proportional to equation 2 by 1/N. As a result, the fft from the scipy package can be used to calculate the Fourier coefficients. Therefore, the following line of code were used to calculate the fourier coefficients:

```
a_k = fft(x)/N  #a_k = (1/N)*DFT
```

Part C

To compress the file, the total number of coefficients are incrementally added until the power of the coefficients matches or exceeds the power or a fraction of the original signal, $1-\alpha$. A subroutine in python is implemented to calculate the minimum number of coefficients needed to match the percentage of power of the original signal. Equation 5 shows succinctly shows the algorithm implemented.

$$\sum_{k=-k_\alpha}^{k_\alpha} |a_k|^2 \geq (1-\alpha)\sum_{k=-\frac{N-1}{2}}^{\frac{N-1}{2}} |a_k|^2 \tag{5}$$

When α is 0, the signal is lossless. The greater alpha is, the more loss occurs and less coefficients are needed. Because Fourier coefficients are periodic with N, the maximum value of k$_\alpha$ is N-1/2, 75,000,000 for N=150,000,000. Therefore, the number of coefficients will always be less than than half the original number of data points.

The following code shows the implementation of the subroutine. The full code is shown in the appendix.

```
#%% Subroutine to coumpute the smallest value of k_a
def computeKa(N, alpha, ak):

    #Calculating (1-alpha)*total power of original signal based on
    #Parseval's theorem for DFT
    ak = np.absolute(ak)
    ak = np.square(ak)
    val1 = (1-alpha)*np.sum(ak)


    #If only 1 coefficient is needed
    val2 = ak[0]
    if(val2>=val1):
        return 0

    #Calculate keep incrementing k_alpha by 1 to see if the power of the
    #(1-alpha)*original signal is less then the power from -k_alpha to k_alpha
    ka = 0
    for i in range(1,N):
        #Magnitude of conjugates is the same -->
        #add 2(magnitude(a_k))^2
        ka = ka + 1
        val2 = val2 + 2*ak[ka]

        if(val2 >= val1):       #Check for the condition
            break

    return ka
```

Part D

For each value of alpha, the minimum number of coefficients, k_α, is determined. Two parallel arrays are used to keep track of the values of alpha and the corresponding minimum k_α values. The following code shows the data structures in which the values of alpha and the corresponding values of k_α are stored.

```
# Compute number of terms for each value of alpha given (0 is added)
alphaArray = np.array([0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1,0.05,0.01,0.005,0.001,0])
minKArray = np.zeros((len(alphaArray),1)) #Parallel array to alphaArray

index = 0
for alpha in alphaArray:
    minKArray[index] = computeKa(N, alphaArray[index], a_k)
    index = index+1
```

Part E

The coefficients, from a_0 to a_{k_α} are then stored in a dictionary, that can be indexed by alpha, called alphaDict. The code below shows the implementation.

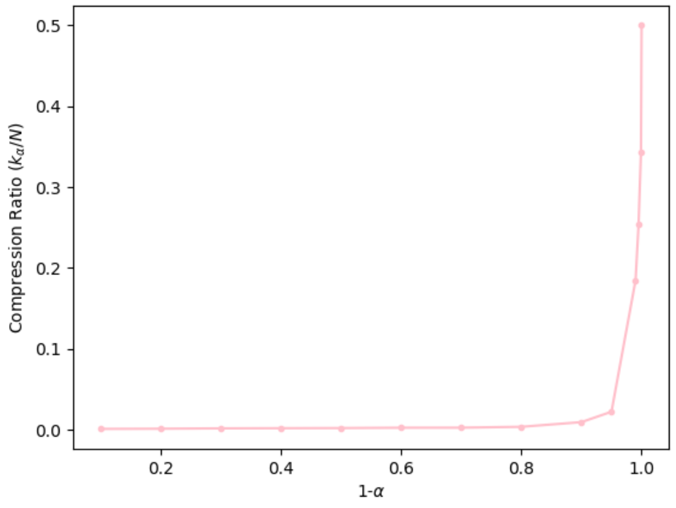
```
# For each value of \alpha and its corresponding ka,
#store the values from a_0 to a_ka in a dictionary

alphaDict = {}  #key:alpha, value:values from a_0 to a_ka per alpha
alphaIndex = 0
for alpha in alphaArray:
    ka = int(minKArray[alphaIndex])
    temp = np.zeros(ka, dtype = "complex_") #retains complex numbers
    for index in range(ka):
        temp[index] = a_k[index]
    alphaDict[alpha] = temp
    alphaIndex = alphaIndex + 1
```

Part F

The relationship between (1-α) and the compression ratio of k_α/N is plotted below and the code used to generate the plots is included in the appendix. For a $(1 - \alpha)$ value of 1, meaning lossless compression, the file will have 50% of the initial power compared to the original file. As the loss increases the number of coefficients needed decreases dramatically. To restore less than or equal to 80% of the original power, less than 0.015% of the original number of coefficients are needed, significantly reducing the size of the file.

Figure 1: Compression Ratio versus (1-α)



Part G

To reconstruct the signal, the following formula needs to be applied:

$$x[n] = \sum_{k=-k_\alpha}^{k_\alpha} a_k e^{j\frac{2\pi}{N}kn} \tag{6}$$

Since only the positive Fourier coefficients are available, the concept of conjugate symmetry in the Fourier series is used to determine the negative coefficients.

$$x[n] = a_0 + \sum_{k=1}^{k_\alpha} a_k e^{j\frac{2\pi}{N}kn} + \sum_{k=k_\alpha}^{-1} a_k e^{j\frac{2\pi}{N}kn} \tag{7}$$

$$a_k = -a_k^* \tag{8}$$

$$\sum_{k=k_\alpha}^{-1} a_k e^{j\frac{2\pi}{N}kn} = \sum_{k=-1}^{k_\alpha} a_{-k} e^{-j\frac{2\pi}{N}kn} = \sum_{k=-1}^{k_\alpha} a^* k(e^{j\frac{2\pi}{N}kn})^*$$

Therefore

$$x[n] = a_0 + \sum_{k=1}^{k_\alpha} a_k e^{j\frac{2\pi}{N}kn} + \left(\sum_{k=1}^{k_\alpha} a_k e^{j\frac{2\pi}{N}kn}\right)^* \tag{9}$$

The IFFT call in python has the sum running from 0 to N-1. In order to match this convention, x[n] can also be represented as

$$x[n] = -a_0 + \sum_{k=0}^{k_\alpha} a_k e^{j\frac{2\pi}{N}kn} + \left(\sum_{k=0}^{k_\alpha} a_k e^{j\frac{2\pi}{N}kn}\right)^* \tag{10}$$

Note that the only difference between equation 9 and 10 are the lower bounds on the summation and the negative sign of the a_0 coefficient. The lower bounds match to the formula for the inverse fft (ifft) of the scipy package. The negative a_0 coefficient is accounted for because without it, there would be a duplicate value of a_0 in the signal as both sums add the coefficient in the data set. Additionally as the data was originally divided by 1/N when the coefficents were calculated, each coefficient needs to be multiplied by N when the data is restored.

The following code does this operation for every value of alpha inputted and puts the reconstructed signals in a dictionary where the key is the corresponding alpha of the reconstructed signal.

```
#Reconstructing the original signal

reconstructionary = {}                    #key:alpha, value:reconstructed signal
alphaIndex = 0
for alpha in alphaArray:
    coeff = alphaDict[alpha]
    baseifft = ifft(coeff, n=N)
    #Take IFFT of coefficients 0 to ka + conjugate of coefficients from 0 to ka
    #- (extra coefficient at 0)/N
    temp = (baseifft + np.conjugate(baseifft) - coeff[0]/N)*N
    reconstructionary[alpha] = temp
    alphaIndex = alphaIndex + 1
```

Part H

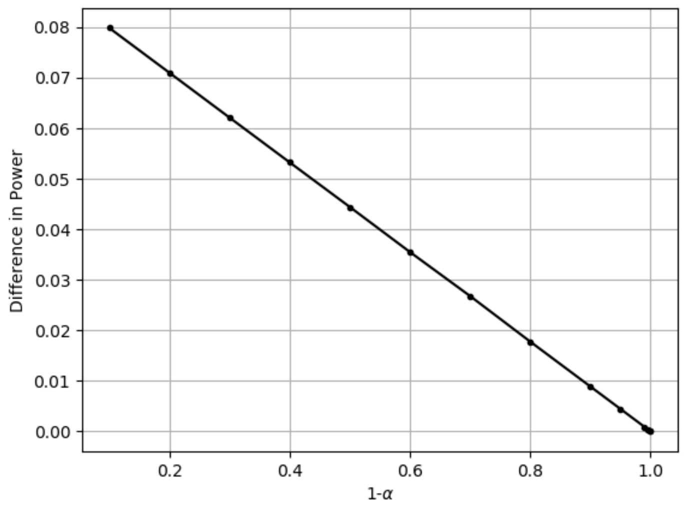
Differences in power between the reconstructed audio and the original can indicate how much of the original audio is restored. As the reconstructed signal should be completely real, the formula for power is shown in equation 11:

$$\text{Average Power} = \sum_{n=0}^{N-1} x[n]^2 \tag{11}$$

Based on the data, there is a linear relationship between the difference in power and (1-α), as expected. As (1-α) increases, the difference in power also decreases. The data shows that even when α is 0, there is still small loss which can likely be attributed to numerical error in python due to rounding.

α	Original Power - Reconstructed Power
0.9	0.0798855
0.8	0.0710165
0.7	0.0621324
0.6	0.0532666
0.5	0.0444331
0.4	0.0355108
0.3	0.0267944
0.2	0.0177688
0.1	0.00887942
0.05	0.00443852
0.01	0.00088761
0.005	0.00044380
0.001	$8.87611234\text{x}10^{-5}$
0	$9.40358902\text{x}10^{-14}$

Figure 2: Power Difference between Reconstructed Signal an Original Signal



The following code was used to calculate the difference in power between the original and the reconstructed signal.

```
#%% Calculate the average power difference between the reconstructed signals versus the origi
originalPower = np.sum(np.square(x))/N
powerDiff = np.zeros(len(alphaArray))

index = 0
for alpha in alphaArray:
    reconPower = np.sum(np.square(np.absolute(reconstructionary[alpha])))/N
    powerDiff[index] = originalPower - reconPower
    index = index + 1
```

2.3 Reducing Noise in Decompressed File through Moving Average Filter

Because of slight power differences between the files due to numerical errors, additional noise was added to the signal that made the amplitudes of the signal vary compared to the original values. To smooth out noise, a 5-point moving average filter was implemented using the following code:

```
ma_filtered = np.zeros(N-5)
for i in range(N-5):
    y_ma=0
    for j in range(5):
        y_ma=y_ma+reconstructionary[0][i+j]
    y_ma=y_ma/5
    ma_filtered[i]=y_ma
```

```
wavfile.write('5ptmovingaveragefiltered.wav',fs,ma_filtered)
#This sounds much better!
```

Although the audio quality sounds better, it does sound slightly different compared to the original signal. The audio sounds more equalized compared to the original file and the accents in the music and percussive instruments are less pronounced. The audio quality is not perfect but it is functional. Decreasing the range of the average from 5 to 2 might sound better are preserve more of the original style of the audio, but it will take longer for that code to run than a 5-point moving average filter.

3 Conclusion

By converting a signal into its Fourier series, the amount of data can be reduced by at least half. If a power loss is allowed, the coefficients can be further reduced from there. The data can then be restored using IFFT by manipulating the negative coefficients through the conjugate symmetry property of Fourier coefficients. The power difference between the original signal and the reconstructed signal depend on the value of α, the percentage of loss in power of the signal. A greater alpha will lead to fewer coefficients needed to reconstruct the signal at the given power. Further noise reduction algorithms can aid in restoring the quality of the original audio.

References

Discrete Fourier Transform (numpy.fft). (2017, June 10).
Retrieved from https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.fft.html

A Code

```
from scipy.io import wavfile
from scipy import fft, ifft
import numpy as np
import matplotlib.pyplot as plt

#%% Subroutine to coumpute the smallest value of k_a
def computeKa(N, alpha, ak):

    #Calculating (1-alpha)*total power of original signal based on Parseval's theorem
    #for DFT
    ak = np.absolute(ak)
    ak = np.square(ak)
    val1 = (1-alpha)*np.sum(ak)

    #If only 1 coefficient is needed
    val2 = ak[0]
    if(val2>=val1):
        return 0

    #Calculate keep incrementing k_alpha by 1 to see if the power of the
    #(1-alpha)*original signal is less then the power from -k_alpha to k_alpha
    ka = 0
    for i in range(1,N):
        ka = ka + 1
        val2 = val2 + 2*ak[ka]  #Magnitude of conjugates is the same
                                #--> add 2(magnitude(a_k))^2
        if(val2 >= val1):       #Check for the condition
            break

    return ka

#%% Read the file, calculate the fourier coefficients
fs, x = wavfile.read("Broke_leaf.wav")
x = x/max(x)                    #normalize magnitude to prevent overflow
N = len(x)                      #N=150,000,001

a_k = fft(x)/N                  #a_k = (1/N)*DFT
```

```
#%% Compute number of terms for each value of alpha given (0 is added)
alphaArray = np.array([0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1,0.05,0.01,0.005,0.001,0])
minKArray = np.zeros((len(alphaArray),1)) #Parallel array to alphaArray

index = 0
for alpha in alphaArray:
    minKArray[index] = computeKa(N, alphaArray[index], a_k)
    index = index+1

#%% For each value of \alpha and its corresponding ka, store the values from a_0
#to a_ka in a dictionary

alphaDict = {}  #key:alpha, value:values from a_0 to a_ka per alpha
alphaIndex = 0
for alpha in alphaArray:
    ka = int(minKArray[alphaIndex])
    temp = np.zeros(ka, dtype = "complex_") #retains complex numbers
    for index in range(ka):
        temp[index] = a_k[index]
    alphaDict[alpha] = temp
    alphaIndex = alphaIndex + 1

#%% Modifying the Data for Visualization

compressionRatio = minKArray/N                 #calculating the compression ratio
oneMinusAlpha = np.subtract(1,alphaArray)   #calculating 1-alpha

#%% Plotting (1-\alpha) versus Compression Ratio

plt.figure(1)
plt.plot(oneMinusAlpha, compressionRatio, marker = ".", color = "pink")
plt.ylabel(r"Compression Ratio ($k_\alpha/N$)")
plt.xlabel(r"1-$\alpha$")
#plt.title(r"Compression Ratio versus ($1-\alpha$)")

#%% Reconstructing the original signal
```

```
reconstructionary = {}                          #key:alpha, value:reconstructed signal
alphaIndex = 0
for alpha in alphaArray:
    coeff = alphaDict[alpha]
    baseifft = ifft(coeff, n=N)
    #Take IFFT of coefficients 0 to ka + conjugate of coefficients from 0 to
    #ka - (extra coefficient at 0)/N
    temp = (baseifft + np.conjugate(baseifft) - coeff[0]/N)*N
    reconstructionary[alpha] = temp
    alphaIndex = alphaIndex + 1

#%% Calculate the average power difference between the reconstructed signals versus
# the original
originalPower = np.sum(np.square(x))/N
powerDiff = np.zeros(len(alphaArray))

index = 0
for alpha in alphaArray:
    reconPower = np.sum(np.square(np.absolute(reconstructionary[alpha])))/N
    powerDiff[index] = originalPower - reconPower
    index = index + 1

#%% plot (1-alpha) versus powerDiff
plt.figure(2)
plt.plot(oneMinusAlpha, powerDiff, marker = ".", color = "black")
plt.ylabel(r"Difference in Power")
plt.xlabel(r"1-$\alpha$")
plt.grid()

#%% Write to wav file to see if this really works...

wavfile.write('Reconstructed_broke_leaf_lossless.wav', fs,np.abs(reconstructionary[0]))

#%% Let's make a 5 point moving average filter to smoothen out the sound and reduce the noise

ma_filtered = np.zeros(N-5)
for i in range(N-5):
    y_ma=0
```

```
        for j in range(5):
            y_ma=y_ma+reconstructionary[0][i+j]
        y_ma=y_ma/5
        ma_filtered[i]=y_ma

wavfile.write('5ptmovingaveragefiltered.wav',fs,ma_filtered) #This sounds much better!
```