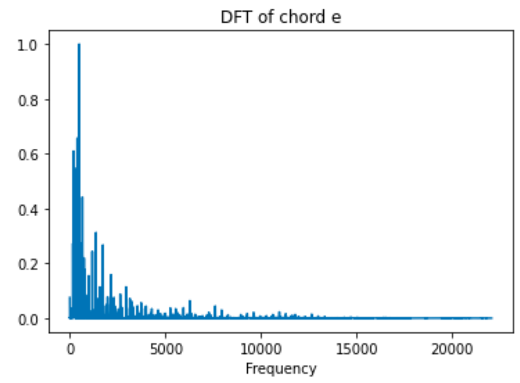
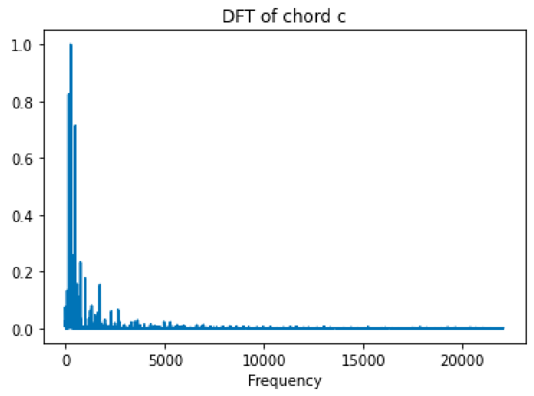
ECE 113: Final Project

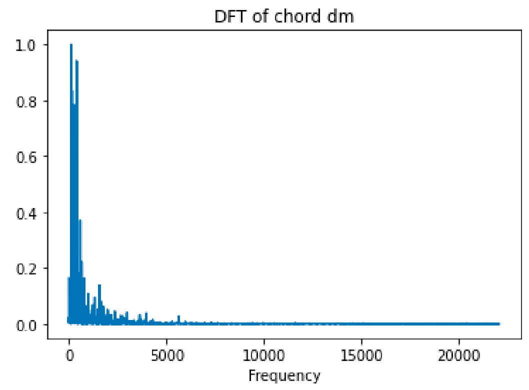
Steven Chu (905-094-800) : Section 1F
Michael Haggenmiller (304-923-247) : Section 1C

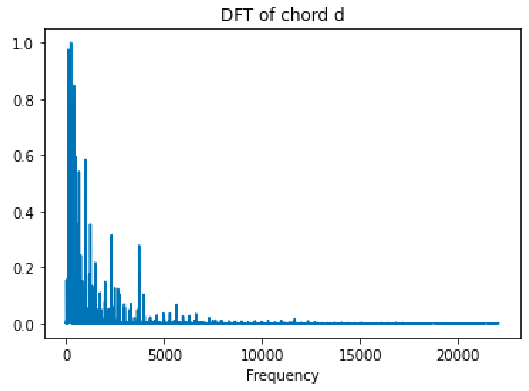
Problem 1

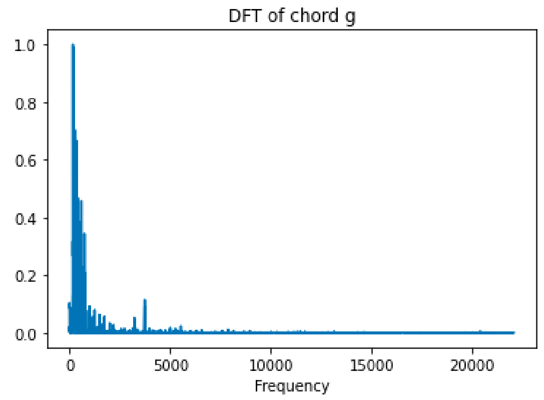
Part A) The plots of the DFT spectra of the 10 chord types are shown below:

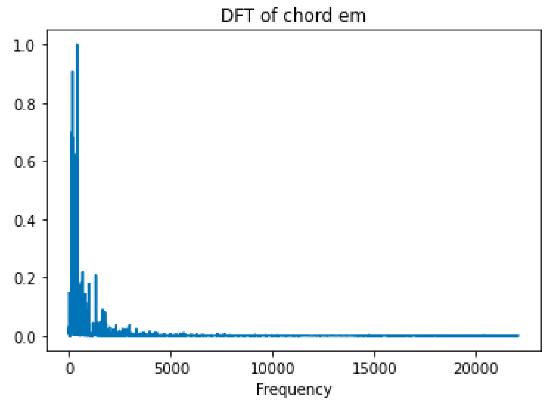


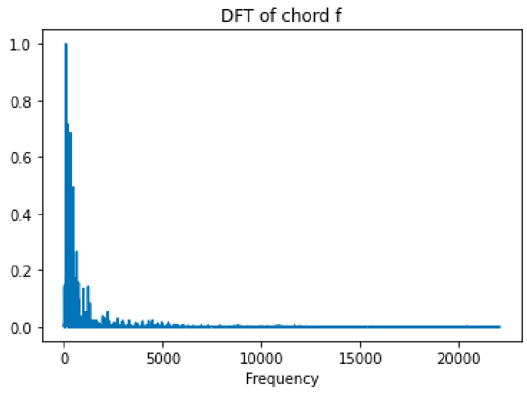


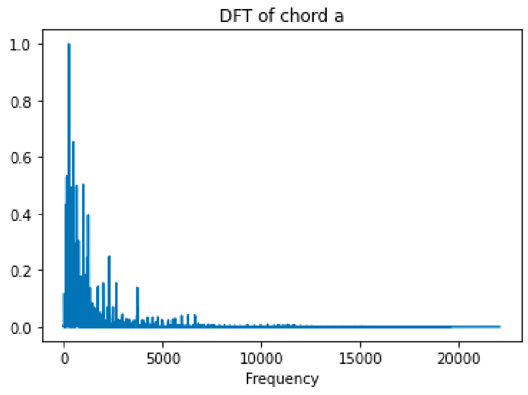


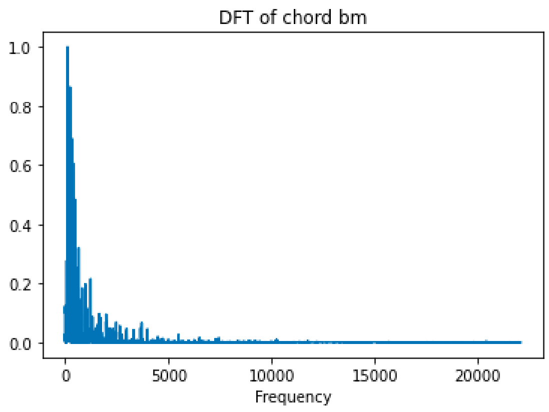


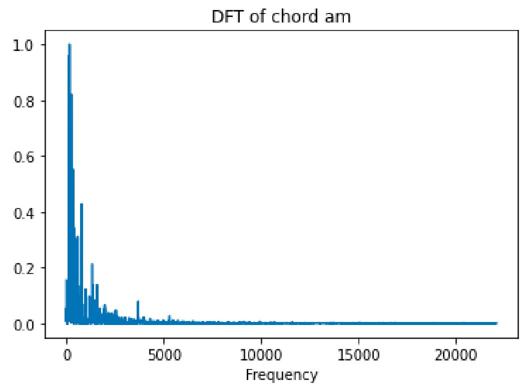












Observations: From these DFT spectra, we can see that the different categories of chords all seem to share the same shape, with the majority of the signals being comprised of the lower base frequencies. However, the most obvious difference between these chords is that they vary in the harmonics, with visibly different peaks in the higher frequencies.

Part B)

Choosing the signals plotted above to be the wavelets, we run the cross correlation comparison with the wavelets against each of the remaining signals in the training data set, in order to classify the remaining signals.

For the sake of efficiency, we opted to carry out this operation as multiplication in the frequency domain using the Fast Fourier Transform, rather than direct convolution. We found that multiplying in the frequency domain and comparing the energies of the Discrete Fourier Transform spectra was much more efficient than calling np.convolve(), and yielded a similar accuracy. This is made possible by Parseval's Theorem, which states that the energies of a signal in its time and frequency domains are proportional.

From the result, we can see that there is a strong correlation and resemblance to a diagonal matrix, albeit with some error. This matrix was computed by initially initializing the confusion matrix as a 10x10 matrix of all zeros. Each time the classification program makes a decision about a signal's chord type, the element at row i and column j of the confusion matrix is incremented, where i is the actual chord type of the signal, and j is the predicted chord type. Eventually, 2000 increments are made across the matrix, and a trend is generated by observing the size of the elements. The larger the elements across the diagonal are, the more accurate the predictions.

The resulting confusion matrix from the comparison is shown below:

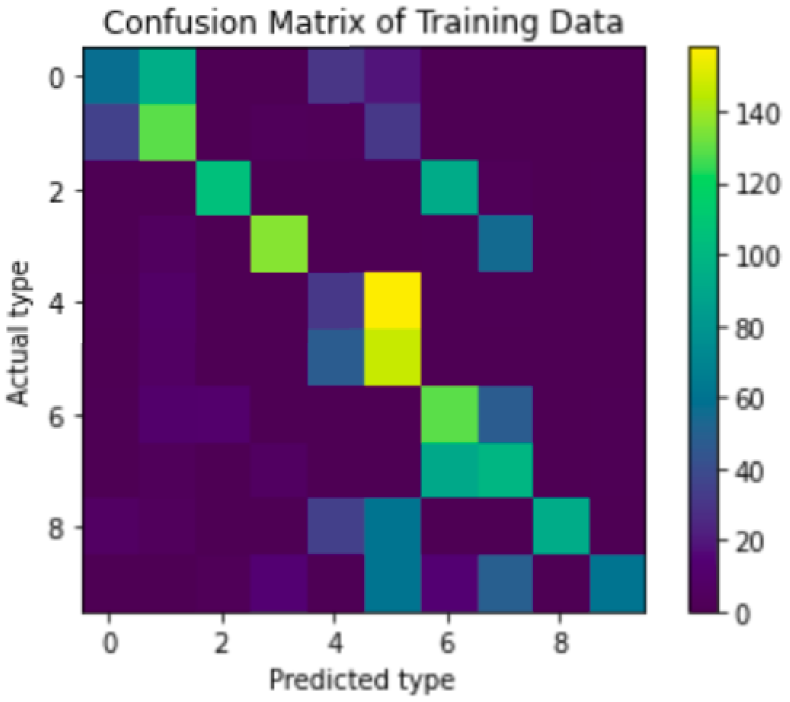


Figure 1: Confusion Matrix generated from Classification in Part B

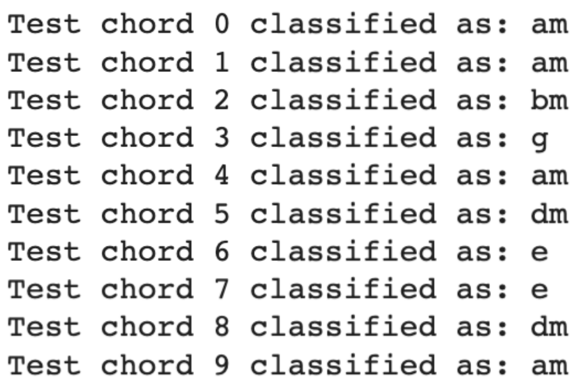
Comments on possible causes of error: It cannot be expected that this method would perfectly classify each chord type. This is because there exists significant variation between signals, even within each class of chord type. Two signals of the same chord can still differ significantly in terms of frequency content. This means that the cross correlation comparison can fail, simply because there is too much unpredictability in these signals. Not only this, but chords can contain the same notes, which means many of the classes will in fact contain the same frequencies as each other. As we saw in the DFT spectra of the classes, each of the DFT spectra of each class still have a very similar general shape, which makes distinguishing between intra-class and inter-class variation even harder.

Part C)

This part of the problem was carried out using the same cross correlation energy function used in Part B. Employing the same wavelets as in Part B, the cross correlation energy comparison was applied to each of the 10 test chords in order to classify them. We wrote Python code to automatically display the classifications chosen.

Comments on Improving Accuracy: This classification method is far from perfect, as the confusion matrix shows. We believe that because chord classification is the goal, and the base frequencies making up each chord is known, it would perhaps be a better idea to generate "ideal signals" containing only the specific known frequencies comprising each chord. Using these ideal model signals as the wavelets instead of choosing a random signal from each class to be the wavelet would likely result in more accurate results. This is because there is still significant variance between chords of the same class due to extraneous noise and differences in timbre, making it more difficult to distinguish chord types. Just randomly choosing one signal from each class as a wavelet to compare with every other signal is inviting error. A benefit to choosing these ideal signals that only contain very specific frequencies, is that computing the cross correlation function will serve to zero out all extraneous frequencies, making the classifications much clearer to distinguish.

The output of the program we wrote for Part C is as follows:



```
Test chord 0 classified as: am
Test chord 1 classified as: am
Test chord 2 classified as: bm
Test chord 3 classified as: g
Test chord 4 classified as: am
Test chord 5 classified as: dm
Test chord 6 classified as: e
Test chord 7 classified as: e
Test chord 8 classified as: dm
Test chord 9 classified as: am
```

Figure 2: Output generated from Classification in Part C

Python Code:

```
######## SETUP ########

from google.colab import drive
drive.mount('/content/drive')
%cd "drive/'My Drive'/ECE113"

from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os
from glob import glob

PATH= './Problem_1_data/training_data_1'

audio_file_paths = [y for x in os.walk(PATH) for y in glob(os.path.join(x[0], '*.wav'))]

index = [i for i in range(len(audio_file_paths))]
columns = ['data', 'label']
df_train = pd.DataFrame(index=index, columns=columns)
for i, file_path in enumerate(audio_file_paths):
    fs, data = wavfile.read(file_path)
    # label assigned to each chord is the name of the folder it is placed inside
    label = os.path.dirname(file_path).split("/")[-1]
    df_train.loc[i] = [data, label]

y = df_train.iloc[:, 1].values
X = df_train.iloc[:, :-1].values
X = np.squeeze(X)
#X = np.stack(X, axis=0)

from sklearn import preprocessing
labelencoder_y = preprocessing.LabelEncoder()
y = labelencoder_y.fit_transform(y)
```

```
######## PART A #########

indices = list(range(0,10))
tracking_indices = list()
y_indices = list()
# generate list of indices in the X data array to find one of each type of chord
for i in range(len(y)):
  if y[i] in indices:
    indices.remove(y[i])
    tracking_indices.append(y[i])
    y_indices.append(i)
chord_types = list()

# generate list of chord types corresponding to the order of the generated indices above
for i in range(len(tracking_indices)):
  chord_types.append(labelencoder_y.classes_[tracking_indices[i]])

# loop through the 10 sampled chords and plot the DFT spectra for each
for i in range(len(y_indices)):
  signal = X[i]
  N = len(signal)
  freq = np.linspace(0, 44100, num=N)
  DFT = np.fft.fft(signal)/N
  DFT = DFT[range(int(N/2))]
  DFT = np.abs(DFT)
  DFT = DFT/np.amax(DFT)
  values = np.arange(int(N/2))
  timePeriod = N/44100
  freq = values/timePeriod
  plt.plot(freq, DFT)
  plt.title("DFT of chord " + chord_types[i])
  plt.xlabel("Frequency")
  plt.show()
```

```
######## PART B #########

# compute the DFT of the crosscorrelation and find its *relative* energy
def crosscorr_energy(signal_1, signal_2):
  L = 0
  length1 = len(signal_1)
  length2 = len(signal_2)
  if(length1 > length2):
    L = length1
  else:
    L = length2
  f1 = np.fft.fft(signal_1, L)
  f2 = np.fft.fft(signal_2, L)
  crosscorr = np.multiply(f1, np.conj(f2))
  crosscorr = np.multiply(crosscorr, np.conj(crosscorr))
  energy = sum(crosscorr)
  return energy


# normalizing all the wavelets for fair comparison, and generating a set of wavelets
X_norm = []
X_wavelets = []
for i in range(len(X)):
  norm_x = X[i]/np.linalg.norm(X[i])
  X_norm.append(norm_x)
for j in y_indices:
  X_wavelets.append(X_norm[j])


###### STILL PART B #######

# generate a confusion matrix
confusion_m = np.zeros(shape=(10,10))
# for each signal, carry out the cross correlation operation and comparison + update confusion matrix
for i in range(len(X_norm)):
  signal_1 = X_norm[i]
  actual_type = y[i]
  energies = []
  for j in range(len(X_wavelets)):
    signal_2 = X_wavelets[j]
    energy = crosscorr_energy(signal_1, signal_2)
    energies.append(energy)
  maxIndex = energies.index(np.max(energies))
  confusion_m[actual_type][tracking_indices[maxIndex]]
  = confusion_m[actual_type][tracking_indices[maxIndex]] + 1

# plot the confusion matrix
plt.imshow(confusion_m)
plt.colorbar()
plt.title("Confusion Matrix of Training Data")
plt.xlabel("Predicted type")
plt.ylabel("Actual type")
plt.show()
```

```
###### PART C #######

# extracting the test files data
PATH= './Problem_1_data/test_data_1'

# max_length of audiofile
index = [i for i in range(10)]
columns = ['data']
df_test_1 = pd.DataFrame(index=index, columns=columns)
for i, file_path in enumerate(glob(os.path.join(PATH, '*.wav'))):
    fs, data = wavfile.read(file_path)
    df_test_1.loc[i] = [data]
# creating a list of the test signals
X_test = df_test_1.iloc[:, :].values
X_test = np.squeeze(X_test)


# normalize signals in X_test
x_test_norm = []
for i in range(len(X_test)):
  x_t_norm = X_test[i]/np.linalg.norm(X_test[i])
  x_test_norm.append(x_t_norm)

# run same cross correlation comparison with the test data signals using the same chosen wavelets
for i in range(len(x_test_norm)):
  signal_1 = x_test_norm[i]
  energies = []
  for j in range(len(X_wavelets)):
    signal_2 = X_wavelets[j]
    energy = crosscorr_energy(signal_1, signal_2)
    energies.append(energy)
  maxIndex = energies.index(np.max(energies))
  decision = "Test chord " + str(i) + " classified as: " +
  str(labelencoder_y.classes_[tracking_indices[maxIndex]])
  print(decision)
```

Problem 2

Task 1

The Python code for Task 1 is as follows:

```
# importing relevant libraries
from scipy.io import wavfile
from scipy import signal
from matplotlib import pyplot as plt
import numpy as np
import math

# update this with the path to inception folder
PATH = '/content/drive/My Drive/ECE 113 Project/inception_sound_track.wav'
# load in inception data
inception_fs, inception_data = wavfile.read(PATH)

######### Task 1 #########

# taking the initial DFT of one channel of the inception signal
inception_fft = np.fft.fft(inception_data[:,0])                      #part (a)
# taking the magnitude of the DFT
inception_fft_mag = abs(inception_fft)                               #part (b)
# initial placeholder for signal we will write to the new file
inception_data_new = inception_data[:,0]

# initialize a phase signal following normal distribution bounded at (0, 2*pi)
norm_dist = np.random.randn(inception_fft.size)+1                    #part (c)
dist = ((2 * math.pi) * (norm_dist +1)) % (2 * math.pi)
dist = abs(dist)

# perform 100 iterations
for i in range(100):                                                 #part(i)
  #combine original magnitude with generated phase
  dist = np.exp(1j * dist)
  inception_fft_2 = np.multiply(inception_fft_mag, dist)             #part(d)
  #take IDFT of newly combined Fourier Transform
  inception_data_ifft = np.fft.ifft(inception_fft_2)                 #part(e)
  #take only real part of newly created signal
  inception_data_ifft = inception_data_ifft.real                     #part(f)
  #copy the signal over so we can write it to a file when the loop ends
  inception_data_new = inception_data_ifft
  #carry out DFT on new signal and take its phase
  inception_fft_2 = np.fft.fft(inception_data_ifft)
  dist = np.angle(inception_fft_2)                                   #part(g)

# write obtained signal to new wavfile
wavfile.write('/content/drive/My Drive/ECE 113 Project/inception_sound_track_new_fft.wav',
inception_fs, inception_data_new)
```

Playing the .wav file yielded from this task, we found that while the clip seemed to contain the correct frequencies (i.e. the notes playing in the clip were the same as that of the original), there was a great deal of noise. Furthermore, the obtained clip seemed to lack the structure and rhythm of the original clip entirely. The clip sounded more like a constant, held out noise that resembled the chord being played in the original clip.

With this clip, we are able to observe the analogy presented in the problem statement: the magnitude spectrum of the DFT contains information about which frequency components are present in the signal, while the phase indicates their relative positions in the time domain signal. The fact that this clip contains the correct frequencies but not the proper structure indicates that the phase was not properly reconstructed in the iterations performed. Indeed, when a print statement was added to the loop to print the phase array in each iteration, we found that the phase was not changing at all across the iterations. Ultimately, this resulted in a random phase in the obtained signal that did not correspond to a reasonable sounding audio clip. We can attribute this to the fact that we cannot accurately reconstruct the phase or make it converge to a reasonable value when taking the magnitude across the very large number of samples (in this case, the entire length of the signal).

Task 2

In this task, we utilize the Short Time Fourier Transform in place of the Discrete Fourier Transform to attempt the same procedure. The problem with using the DFT to attempt to reconstruct the phase of a signal from its magnitude is that, across a large segment, the magnitude of a signal provides no information about its original phase. Therefore, repeatedly transforming the signal will yield no new information, and the obtained phase will remain largely constant.

However, the reasoning behind using the Short Time Fourier Transform is that we are able to break the large signal down into smaller segments, where the hope is that the variation in the phase of these segments remains negligible. This due to the fact that, with very short segments of an audio clip, the probability of there being a large change in sound quality is relatively small. Since we are more accurately able to determine the phase of these smaller segments, we can call the DFT on each of these segments individually and eventually piece together each of their phase values to have the overall phase converge to a reasonable value.

The Python code for Task 2 is as follows:

```
######### Task 2 ##########

# taking the initial STFT of one channel of the inception signal
f, t, inception_stft = signal.stft(inception_data[:,0], fs=inception_fs, window='bartlett',
nperseg=64, noverlap=32)

# taking the magnitude of the STFT
inception_stft_mag = abs(inception_stft)

# initial placeholder for signal we will write to the new file
inception_data_new = inception_data[:,0]

# generating a phase matrix with a normal distribution bounded by (0, 2*pi)
rows = int(inception_stft.size/inception_stft[0].size)
columns = inception_stft[0].size
normal_dist = abs(np.random.randn(rows, columns))+1
phase = ((2 * math.pi) * normal_dist) % (2*math.pi)

# taking phase of original STFT and generate empty list to store Frobenius norms
original_phase = np.angle(inception_stft)
```

```
frob_norms = list()

# perform 500 iterations
for i in range(500):
  #combine original magnitude with generated phase
  mult = np.exp(1j * phase)
  inception_stft_new = np.multiply(inception_stft_mag, mult)
  #take ISTFT of newly combined ST Fourier Transform
  f, inception_data_stft = signal.istft(inception_stft_new, fs=inception_fs, window='bartlett',
  nperseg=64, noverlap=32)
  #take only real part of newly created signal
  inception_data_stft = inception_data_stft.real
  #copy the signal over so we can write it to a file when the loop ends
  inception_data_new = inception_data_stft
  #carry out STFT on new signal and take its phase
  f, t, inception_stft_new = signal.stft(inception_data_stft, fs=inception_fs, window='bartlett',
  nperseg=64, noverlap=32)
  phase = np.angle(inception_stft_new)
  #on iterations 1, 50, 100, 250, and 500, take the Frobenius norm of the phase difference
  if i == 0 or i == 49 or i == 99 or i == 249 or i == 499:
    phase_diff = np.subtract(original_phase, phase)
    norm = np.linalg.norm(phase_diff)
    frob_norms.append(norm)

# write obtained signal to new wavfile
wavfile.write('/content/drive/My Drive/ECE 113 Project/inception_sound_track_new_stft1.wav',
 inception_fs, inception_data_new)

# plot the obtained Frobenius norms with respect to number of iterations
iterations = [1, 50, 100, 250, 500]
plt.plot(iterations, frob_norms)
plt.xlabel("Number of Iterations")
plt.ylabel("Frobenius Norm of Phase Difference")
plt.show()
```

Upon playing the audio clip yielded from this task, we found that the reverse problem had occurred from Task 1. It appeared that the rhythm and structure of the clip had been captured very well, indicating that the phase content of the signal had converged to a reasonable value. However, the frequency content had been distorted beyond recognition, and the clip overall sounded very noisy to the point where the original notes could not be distinguished.

Upon further investigation, we found that the ISTFT did not reconstruct the time domain signal perfectly from the STFT. In fact, we found that simply taking the STFT of the inception signal a single time and directly converting the signal back using the ISTFT resulted in a very noisy, albeit recognizable, version of the original audio clip. This was a very confounding problem, because we tried multiple combinations of windows and overlaps that met the Constant Overlap Add constraint necessary for perfect reconstruction of signals using the ISTFT, such as the rectangular window with no overlap, and the cosine, Hann, and Bartlett windows with 50% overlap. Despite this, we found that this distortion occurred similarly for all of them. Ultimately we decided to use the Bartlett window with 50% overlap, but let it be noted that we achieved very similar results with the Hann and cosine windows.

The plot showing the relationship between the norm phase difference and number of iterations is shown below:

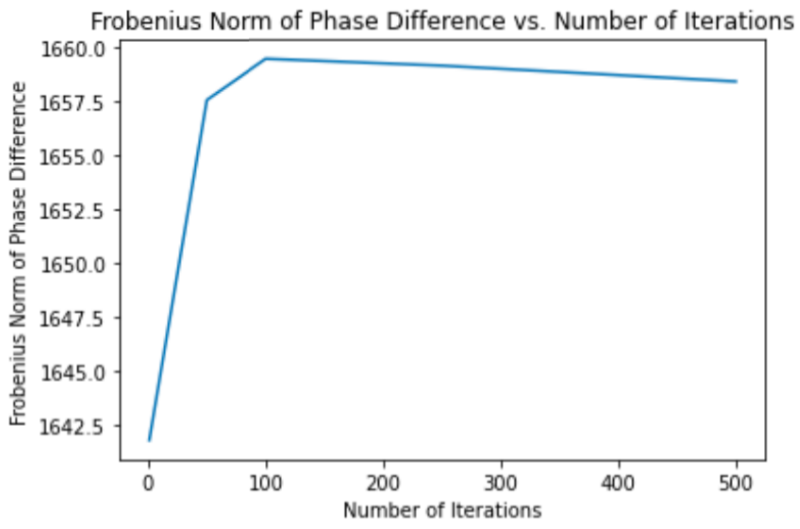


Figure 3: Plot showing Frobenius Norms vs. Number of Iterations

We can see from this plot that the norm of the phase difference between the generated phase and the original phase changes rapidly between the first 50-100 iterations. Between iterations 100-500, it appears that the phase difference changes more slowly, and appears to be converging to a particular value. From this plot, it appears that the phase difference is increasing, and therefore that should mean the generated phase of the 500th iteration is worse than that of the first iteration. However, listening to the audio clips generated by both indicates that the clip generated by the 500th iteration captures the structure and phase of the original clip much more effectively (both do not contain the right frequencies however).

This observation can be explained using a fact introduced in the problem statement: Because our hearing cannot distinguish global shifts in phase, the generated phase of the audio clip does not necessarily need to converge to the original phase to sound plausible. It is likely that past the 100th iteration, the generated phase was converging to a globally shifted version of the original phase that happened to have a larger phase difference from the original than the randomly generated phase in the first iteration.