International Institute of Information Technology Bangalore

Signal Processing
EC 304

Audio Signal Processing

April 17, 2022

Group 15
Abhinav Mahajan (IMT2020553)
Anshul Madurwar(IMT2020554)

Introduction

The aim of this project is to record two audio signals using a microphone, and then convert them into a .wav file format. These files are then merged to form a combined audio signal. In this case, we consider vocals and electric guitar as our two audio sources. But this is just the recursive part of the process, our major task is to use the combined audio signal and extract the input audio signals, and how we do this ?, let us find out...

Dataset

- The dataset consists of two audio files recorded using a phone. The recorded input audio files are uncompressed and with a sample rate of 44100 and 24 bits per sample.
- The same device is to be used to record the two audio signals to maintain a constant sample rate else they have to be re-sampled before combining, and the re-sampled data often leads to loss in information.
- The two input signals are plotted in their time and frequency domain below: -

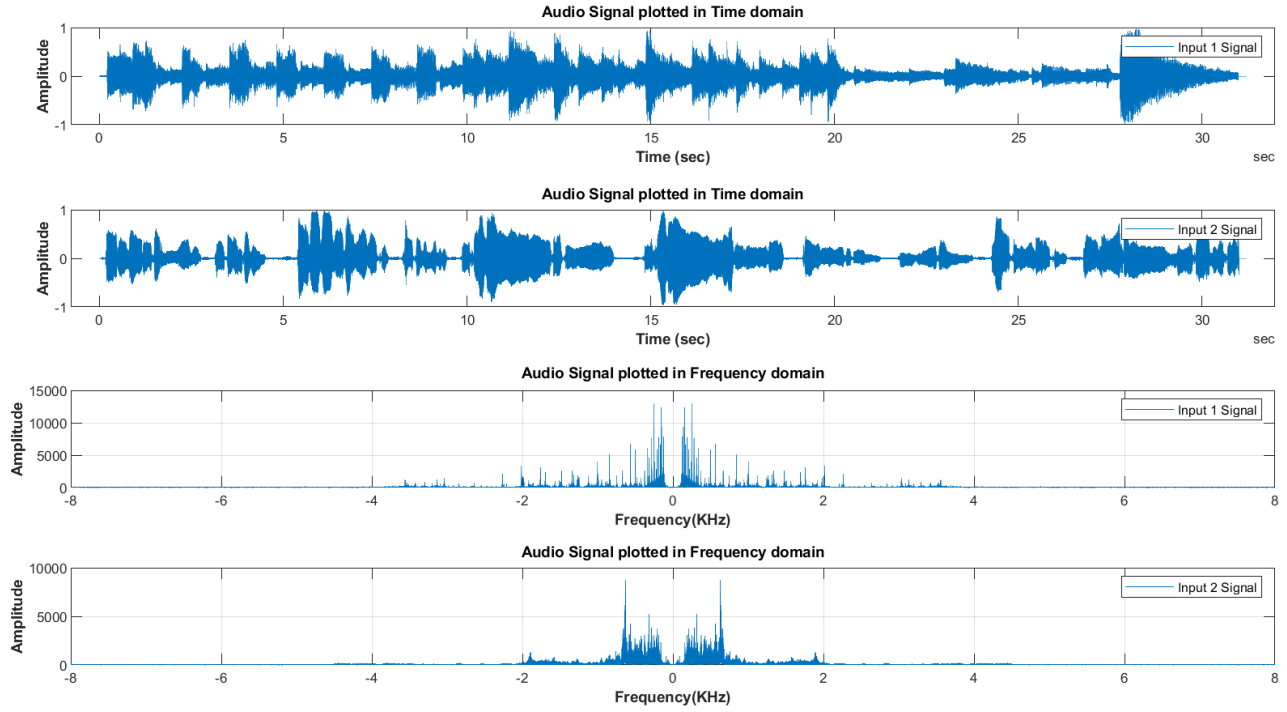

Figure 1: Input Signals in Time and Frequency Domain.

```
Command Window
>> final_matlab_code
            Filename: 'D:\IIIT_Bangalore\4th Semester\Signal Processing\Project\Final\input1_new.wav'
    CompressionMethod: 'Uncompressed'
          NumChannels: 2
           SampleRate: 44100
         TotalSamples: 1376124
             Duration: 31.2046
                Title: []
              Comment: []
               Artist: []
        BitsPerSample: 24
```

Figure 2: Data Set Information.

Procedure

- The **first step** in the project was acquiring a Data-set on which we can work. To make our work show an actual real life example where this project is of significance, we decided to perform music mixing and de-mixing. We decided to mix the guitar and vocals components of the song **"Tere Mere"** by **Armaan Malik** and **Abhinav** was on the **guitar** and **Anshul** sang the song. We decided to cap the length to **approximately 30s**. Using **Garage Band**, we made sure that the number of samples and sampling frequency matched and then we made our Sample data-set of inputs.

- The **second step** was to fix the encoding of the input .wav files and we used **SoX** software to correct the encoding so that python and MATLAB can read the files. So our final data-set consisted of **input1_new.wav** and **input2_new.wav**.

- The **third step** was to mix the files. To achieve that, first we have the read the data in the wavfiles and store them in appropriate variables. The implementation can be seen in the python code file attached in the submission folder. For performing ICA, we need at least 2 mixed signals with different weightages of mixture of the 2 audio sources. The weights were stored in an **array A** and after matrix multiplication we save the mixed wavfiles in an **array X**.

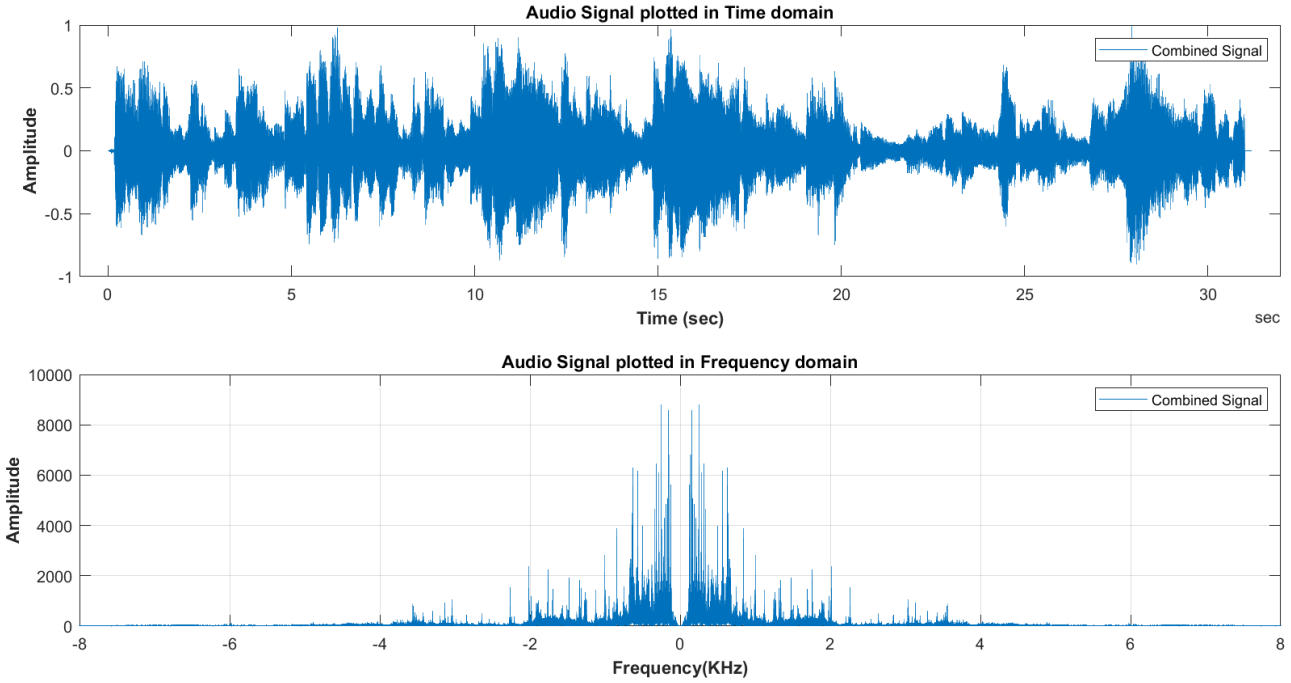

Figure 3: Combined signal in Time and Frequency Domain.

- The **fourth step** was to separate the components using **Independent Component Analysis (ICA)**. Using **numpy** and **scipy** libraries, we were able to do the python implementation of it. We fed the **array X** to the ICA function and the number of iterations was set to 1000, for optimal accuracy. We saved the output in a **matrix S**. From **S** we used a wavfile writing function to write the output to a output wave file.

- It is noteworthy that our data-set had **2 columns of audio data**, the left and right headphone components. So for each component (**left and right**) we had to perform separate mixing and de-mixing for accuracy of the result. After de-mixing both the left and right headphone components, we did column addition to generate a 2 column wavefile format and saved them in our output files (**out1.wav** and **out2.wav**) so that our de-mixed data set exactly resembles our input dataset (**input1_new.wav** and **input2_new.wav**) to minimise the loss of data.

Theory

Now we would like to explain the theory behind **Independent Component Analysis, ICA**.

- In signal processing, ICA is a **computational method** for separating a **multivariate signal** into **additive sub-components**. This is done by assuming that at most one sub-component is a **non-Gaussian signal** and that the sub-components are **statistically independent** from each other. This is a derivative of Blind Source Separation with a specialised data-set. **Source separation**, **blind signal separation (BSS)** or **blind source separation**, is the separation of a set of source signals from a set of mixed signals, without the aid of information about the source signals or the mixing process.
- Before we delve into how ICA works, the two assumptions are: -
  1. The source signals are independent of each other.
  2. The values in each source signal have non-Gaussian distributions.
- ICA finds the **independent components** (also called **factors**, **latent variables** or **sources**) by **maximizing the statistical independence** of the estimated components. We may choose one of many ways to define a proxy for independence, and this choice governs the form of the ICA algorithm.
- The two broadest definitions of independence for ICA are: -
  1. Minimization of mutual information.
  2. Maximization of non-Gaussianity.
- The **Minimization-of-Mutual information (MMI)** family of ICA algorithms uses measures like **Kullback-Leibler Divergence** and **maximum entropy**. The **non-Gaussianity family** of ICA algorithms, motivated by the central limit theorem, uses **kurtosis** and **negentropy**.
- Typical algorithms for ICA use **centering** (subtract the mean to create a zero mean signal), **whitening** (usually with the eigenvalue decomposition), and **dimensionality reduction** as preprocessing steps in order to simplify and reduce the complexity of the problem for the **actual iterative algorithm**.
- Whitening and dimension reduction can be achieved with **principal component analysis** or **singular value decomposition**. Whitening ensures that all dimensions are treated equally a priori before the algorithm is run. In general, ICA **cannot** identify the actual number of source signals, a uniquely correct ordering of the source signals, nor the proper scaling (including sign) of the source signals. In our **demixed signals** you can see there is a **significant change in amplitude**, though the waveform is retained. After conversion to a **float32 data type** in the wavefile, the **quality of signal is unchanged**.
- Our implementation of ICA can be seen in the python file attached in the submission folder.

Data Compression

- During the project, we had to use a dataset which comprised of .wav files and only then we truly understood its flexibility.
- Wavefiles are just **container classes** for audio files, which can comprise of any audio files either compressed with **mp3**, **mp4** or **AAC**, etc or also consist of **raw, uncompressed audio data**.
- The header contains all the information relevant to the data and using it, any audio player can play these wavefiles. During the project, we had to make a lot of intermediate files, when we extracted the **left and right headphone signals** for the input files, making the sample mixed signals and their de-mixed signals. After performing arithmetic operations, we seldom exceeded the maximum amplitude supported by the datatype of the wavefiles and it reflected in inaccurate audio files with inaccurate audio.

- To fix this we had to find a way out to store the dataset, and **float32** was found to be the **winner**, after we divided the entire dataset by the maximum value. Then **scipy.io.wavfile.write()** function in python added the **audio codec** in the header required to make a playable wavefile, otherwise audio players would give an **encoding not supported error** or would store an audio file which will generate no sound on playing.

Results

- The procedure was successful and the de-mixed signals had extremely accurate waveforms which was identical to the input data files. The number of samples and sampling frequency was retained and upon hearing the the wav files, we can see the extent of the accuracy.
- The following graphs portray the signals obtained after **Independent Component Analysis**.
- The **first graph** show the output signals, that are separated from the combined signal in their Time and Frequency Domain.
- The **second graph** shows the comparison between input signals and obtained output signals, We closely observe that the length of the signal is unaltered but the amplitude is changed after the process.
- The obtained output .wav signals are included in the zip file and can be set side-by-side with the input signals to have a clearer picture.

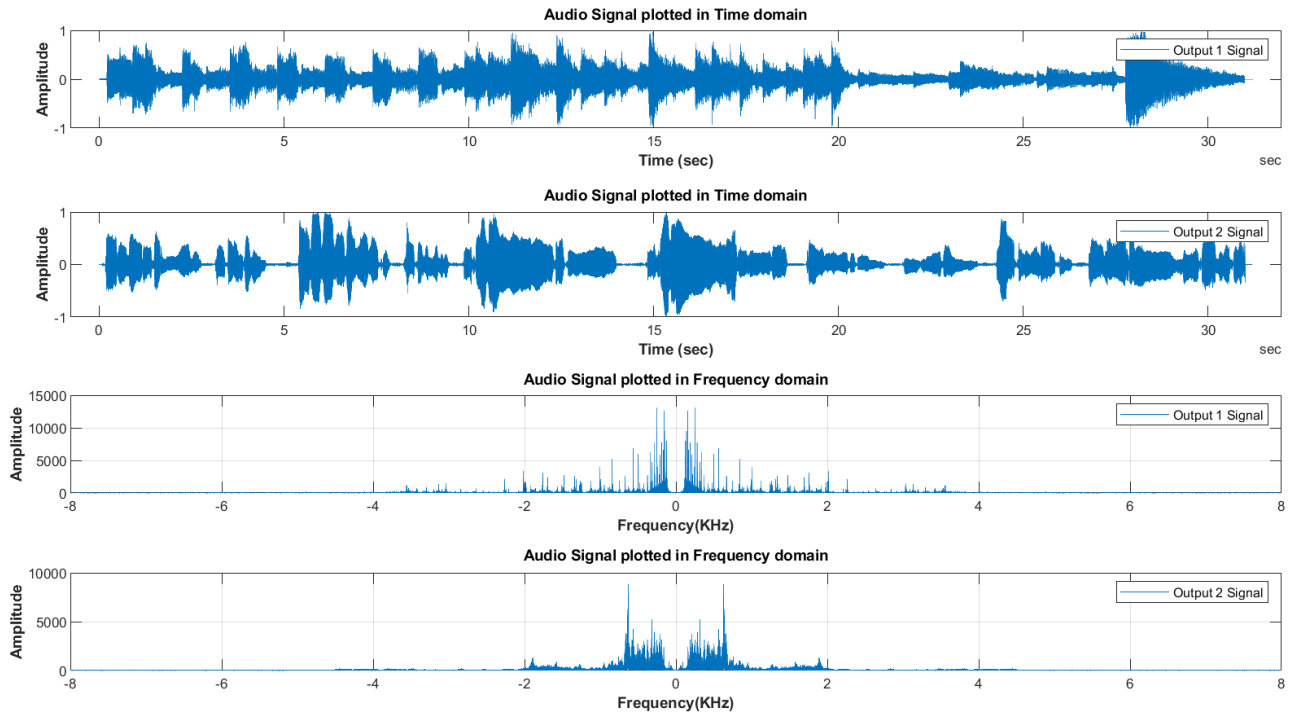

Figure 4: Output signals in Time and Frequency Domain

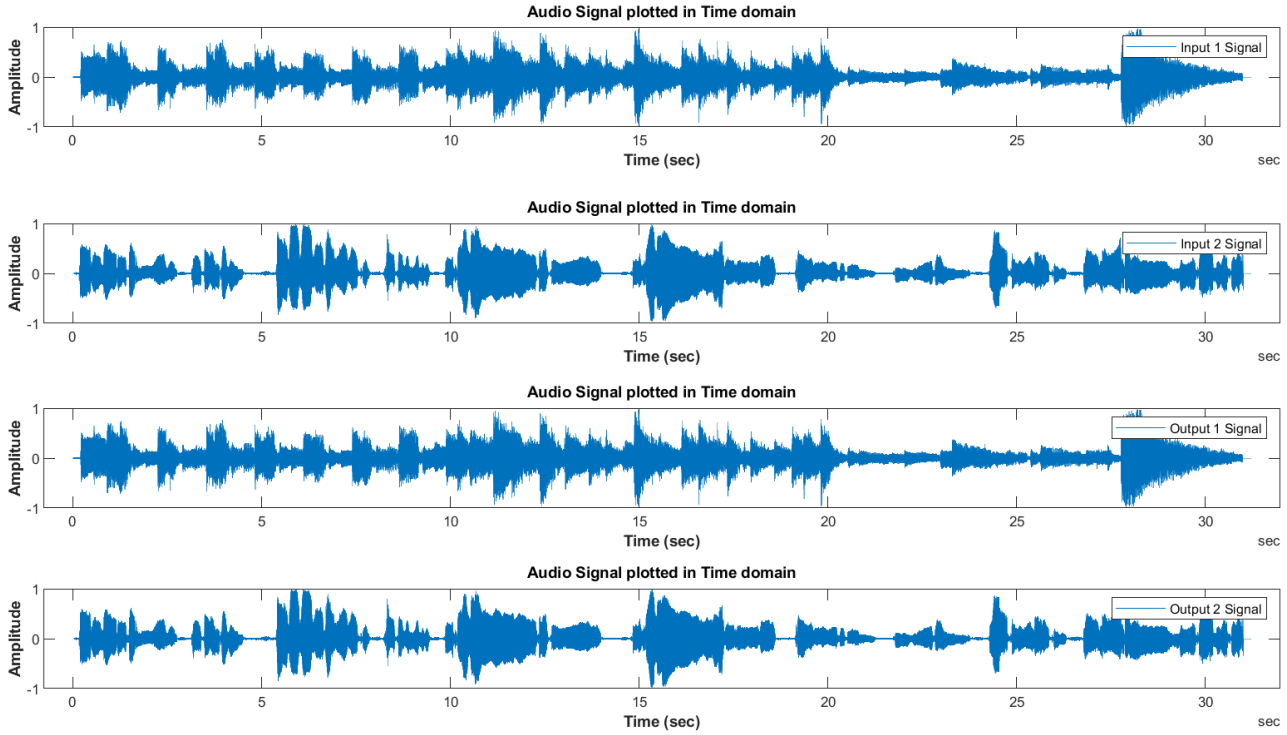

Figure 5: Input Signal Vs. Output Signals obtained after ICA.

Future Work

Independent Component Analysis has extensive use in a lot a fields. Here are a few: -

1. **Professional music de-mixing** can be performed with ICA. We showed how vocals and guitar occupying the same frequency bandwidth can be separated but the extensibility of this can be reached to extracting **bass**, **snare drums**, **hi-hat**, **kick-drums**, **violin**, **different singers**, and so much more. De-Mixing is already a huge field in the **music industry** and also for those who learn music. They can isolate the instrument they wish to learn. Also this can be used for generating **backing tracks** for those learning music and musical instruments and also can be used for the set up of a **karaoke system**!

2. ICA can be used to solve the **cocktail party problem**. Suppose in a **cocktail party**, there are multiple people speaking, there is the sound of glass, there is music in the background, there is the sound of a fan or an AC, and so many more sources of noise. How do we filter out the sources? That was the inspiration of the problem. The gist of it is when there are multiple sound producing sources, how can we filter out the separate sources and isolate them. Human brains have a mechanism to focus on some sound producing sources and blur out the rest, to some extent at least. Machines and computers have found it historically very hard to do the same. **Independent Component Analysis** offers a very good solution for it and can be used to solve this problem.

3. ICA does not end with sound de-mixing. This can also be applied to **image processing**. This can be used to distinguish between **different people**, **animals** and **every other category** which has a visual distinction. This branch in image processing can be extensively used for **security** and **surveillance**. Footage tapes can identify the human who needs to be identified, and also the process can be automated by triggering an alarm the moment a person is caught on a surveillance camera in the act of violating a law.

GitHub Link

Please visit this for an example based explanation.
https://github.com/McLucifer2646/Independent-Componenet-Analysis

Python Code

Python code for performing ICA

```
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.io import wavfile
from scipy import signal
import numpy as np
np.random.seed(0)
sns.set(rc={'figure.figsize': (11.7, 8.27)})


def g(x):
    return np.tanh(x)


def g_der(x):
    return 1 - g(x) * g(x)


def center(X):
    X = np.array(X)

    mean = X.mean(axis=1, keepdims=True)

    return X - mean


def whitening(X):
    cov = np.cov(X)
    d, E = np.linalg.eigh(cov)
    D = np.diag(d)
    D_inv = np.sqrt(np.linalg.inv(D))
    X_whiten = np.dot(E, np.dot(D_inv, np.dot(E.T, X)))
    return X_whiten


def calculate_new_w(w, X):
    w_new = (X * g(np.dot(w.T, X))).mean(axis=1) - \
        g_der(np.dot(w.T, X)).mean() * w
    w_new /= np.sqrt((w_new ** 2).sum())
    return w_new


def ica(X, iterations, tolerance=1e-5):
    X = center(X)

    X = whitening(X)

    components_nr = X.shape[0]

    W = np.zeros((components_nr, components_nr), dtype=X.dtype)

    for i in range(components_nr):

        w = np.random.rand(components_nr)

        for j in range(iterations):

            w_new = calculate_new_w(w, X)

            if i ≥ 1:
                w_new -= np.dot(np.dot(w_new, W[:i].T), W[:i])

            distance = np.abs(np.abs((w * w_new).sum()) - 1)

            w = w_new

            if distance < tolerance:
                break

```

```
        W[i, :] = w

    S = np.dot(W, X)

    return S


def plot_mixture_sources_predictions(X, original_sources, S):
    fig = plt.figure()
    c = 1
    for s in original_sources:
        plt.subplot(4, 1, c)
        c += 1
        plt.plot(s)
    plt.title("real sources")
    for s in S:
        plt.subplot(4, 1, c)
        c += 1
        plt.plot(s)
    plt.title("predicted sources")

    fig.tight_layout()
    plt.show()


def mix_sources(mixtures, apply_noise=False):
    for i in range(len(mixtures)):

        max_val = np.max(mixtures[i])

        if max_val > 1 or np.min(mixtures[i]) < 1:

            mixtures[i] = mixtures[i] / (max_val / 2) - 0.5

    X = np.c_[[mix for mix in mixtures]]

    if apply_noise:

        X += 0.02 * np.random.normal(size=X.shape)

    return X


sampling_rate, source1 = wavfile.read('input1_new.wav')
sampling_rate, source2 = wavfile.read('input2_new.wav')
source1_left_headphone = np.asarray([row[0] for row in source1])
source2_left_headphone = np.asarray([row[0] for row in source2])
source1_right_headphone = np.asarray([row[1] for row in source1])
source2_right_headphone = np.asarray([row[1] for row in source2])

X = np.c_[source1_left_headphone, source2_left_headphone]
A = np.array(([[1, 1], [0.5, 2]]))

X = np.dot(X, A.T)
X = X.T

max_mix1 = np.max(np.abs(np.asarray(X[0])))
mix1 = (np.asarray(X[0]/max_mix1)).astype(np.float)
wavfile.write('Sample_Mixed_Wave_1.wav', int(sampling_rate),
              np.asarray(mix1, dtype=np.float32))
max_mix2 = np.max(np.abs(np.asarray(X[1])))
mix2 = (np.asarray(X[1]/max_mix2)).astype(np.float)
wavfile.write('Sample_Mixed_Wave_2.wav', int(sampling_rate),
              np.asarray(mix2, dtype=np.float32))

S = ica(X, iterations=1000)

wavfile.write('inp1_left_headphone.wav', sampling_rate, source1_left_headphone)
wavfile.write('inp2_left_headphone.wav', sampling_rate, source2_left_headphone)

max_sig_1 = np.max(np.abs(np.asarray(S[0])))
sig_1_32_left = (np.asarray(S[0])/max_sig_1).astype(np.float)
wavfile.write('out1_left_headphone.wav', int(sampling_rate),
```

```
                np.asarray(sig_1_32_left, dtype=np.float32))
max_sig_2 = np.max(np.abs(np.asarray(S[1])))
sig_2_32_left = (np.asarray(S[1])/max_sig_2).astype(np.float)
wavfile.write('out2_left_headphone.wav', int(sampling_rate),
                np.asarray(sig_2_32_left, dtype=np.float32))

plot_mixture_sources_predictions(
    X, [source1_left_headphone, source2_left_headphone], [sig_1_32_left, sig_2_32_left])


X = np.c_[source1_right_headphone, source2_right_headphone]
A = np.array(([[1, 1], [0.5, 2]]))

X = np.dot(X, A.T)
X = X.T
S = ica(X, iterations=1000)

wavfile.write('inp1_right_headphone.wav',
                sampling_rate, source1_right_headphone)
wavfile.write('inp2_right_headphone.wav',
                sampling_rate, source2_right_headphone)

max_sig_1 = np.max(np.abs(np.asarray(S[0])))
sig_1_32_right = (np.asarray(S[0])/max_sig_1).astype(np.float)
wavfile.write('out1_right_headphone.wav', int(sampling_rate),
                np.asarray(sig_1_32_right, dtype=np.float32))
max_sig_2 = np.max(np.abs(np.asarray(S[1])))
sig_2_32_right = (np.asarray(S[1])/max_sig_2).astype(np.float)
wavfile.write('out2_right_headphone.wav', int(sampling_rate),
                np.asarray(sig_2_32_right, dtype=np.float32))

plot_mixture_sources_predictions(
    X, [source1_right_headphone, source2_right_headphone], [sig_1_32_right, sig_2_32_right])

wavfile.write('out1.wav', int(sampling_rate), np.asarray(
    [[sig_1_32_left[i], sig_1_32_right[i]] for i in range(0, len(sig_1_32_left))], ...
        dtype=np.float32))
wavfile.write('out2.wav', int(sampling_rate), np.asarray(
    [[sig_2_32_left[i], sig_2_32_right[i]] for i in range(0, len(sig_2_32_left))], ...
        dtype=np.float32))
```

MATLAB Code

MATLAB code for plotting

```
filename_inp_1 = 'input1_new.wav';
filename_inp_2 = 'input2_new.wav';
filename_combine = 'Sample_Mixed_Wave_1.wav';

filename_out_1 = 'out2.wav';
filename_out_2 = 'out1.wav';

[y,Fs] = audioread(filename_combine);
[y1,Fs1] = audioread(filename_inp_1);
[y2,Fs2] = audioread(filename_inp_2);
[y3,Fs3] = audioread(filename_out_1);
[y4,Fs4] = audioread(filename_out_2);

info1 = audioinfo(filename_inp_1);
disp(info1);

SampFreq = info1.SampleRate;
t = 0:seconds(1/Fs):seconds(info1.Duration);
t = t(1:end-1);

subplot(4, 1, 1);
plotter(t, y1, 'Input 1 Signal');

subplot(4, 1, 2);
plotter(t, y2, 'Input 2 Signal');

subplot(4, 1, 3);
plotter(t, y3, 'Output 1 Signal');

subplot(4, 1, 4);
plotter(t, y4, 'Output 2 Signal');

%================================================================
figure;
subplot(4, 1, 1);
plotter(t, y3, 'Output 1 Signal');

subplot(4, 1, 2);
plotter(t, y4, 'Output 2 Signal');

subplot(4, 1, 3);
my_fft(y3, Fs3, 'Output 1 Signal');

subplot(4, 1, 4);
my_fft(y4, Fs4, 'Output 2 Signal');

%================================================================
figure;
subplot(2, 1, 1);
plotter(t, y, 'Combined Signal');

subplot(2, 1, 2);
my_fft(y, Fs, 'Combined Signal');

%================================================================

function my_fft(y_new, Fs, legend_label)
    n = length(y_new)-1;
    f = -Fs/2:Fs/n:Fs/2;
    y_fft = abs(fftshift(fft(y_new)));
    plot(f/1000, y_fft);
    xlim([-8 8]);
    xlabel ('Frequency(KHz)','fontweight','bold');
    ylabel ('Amplitude','fontweight','bold');
    legend (legend_label)
    title ('Audio Signal plotted in Frequency domain');
    grid on
end
```

```
69
70 function plotter(t, y_var, legend_label)
71     plot(t, y_var);
72     xlabel('Time (sec)','fontweight','bold');
73     ylabel('Amplitude','fontweight','bold');
74     legend(legend_label);
75     title('Audio Signal plotted in Time domain');
76 end
```

References

1. Stone, J.V., 2004. Independent component analysis: a tutorial introduction.

2. Cao, X.R. and Liu, R.W., 1996. General approach to blind source separation. IEEE Transactions on signal Processing, 44(3), pp.562-571.

3. https://towardsdatascience.com/independent-component-analysis-ica-in-python-a0ef0db0955e
 - For Inependent Component Analysis in Python.

4. https://www.uaudio.com/blog/understanding-audio-data-compression/
 - For Data Compression.

5. https://builtin.com/data-science/step-step-explanation-principal-component-analysis
 - For step-by-step ecplanation of Principal Component Analysis.