**CMPT 365 Project 1 Report**

For this project, I am using the Swing, AWT, Sound.Sampled, ImageIO, and File IO libraries for Java. The first two, Swing and AWT, are used for creating the UI of the application, and the other three are used for reading .wav and .png files respectively. For both parts, I am first reading the file into a Java File object, then passing that along to separate methods for processing.

**Q1: Reading .wav Files and Plotting Waveforms**

To read the .wav files, I am using the javax.sound.sampled.AudioSystem library, and the

getAudioInputStream function. This takes the file and turns it into an InputStream variable. I am then using the .read function to convert the InputStream to a byte array. This array is then converted to an integer array by one of two nested for-loops. If the bits per sample of the file is 16, then I am separating out the high and low bytes of each sample, then shifting the high sample left by 8. If the bits per sample is 8, I am simply storing every value into the integer array. I am also accounting for the number of channels by using an inner for loop to switch channels every 16 or 8 bits. In order to obtain the sampling rate, I am simply using the .getSampleRate() function associated with AudioInputStream. However, to calculate the total samples, I am using the following equation.

In order to plot the waveforms, I am using the paintComponent function of JPanel. I have created a new class which inherits JPanel. I am then normalizing both the amplitude values and the total samples of each .wav file to fit within a 1000 by 384 area. Then, I am using the Graphics.drawLine() function, along with a for loop, to iterate through the samples and drawing lines with the following coordinates:

int x1 = (int) Math.ceil(j \* normalizationFactorX);

int y1 = (int) Math.ceil(-1\*audioValues[i][j] \* normalizationFactorY + offset +

dimensionY/2/channels);

int x2 = (int) Math.ceil((j + 1) \* normalizationFactorX);

int y2 = (int) Math.ceil(-1\*audioValues[i][j+1] \* normalizationFactorY + offset +

dimensionY/2/channels);

where normalization factors X and Y are defined by width/samples and height/max amplitude respectively. In the case of multi-channel files, I am further dividing the y coordinates and the normalization factor by the number of channels. The results are as follows, I have included all three of the new samples, and three of the old ones:

bass\_1.wav

Chart

Description automatically generated

cartoon\_2.wav

A picture containing antenna

Description automatically generated

HP\_2.wav

Chart

Description automatically generated

car+horn+x.wav

Chart, histogram

Description automatically generated

Explosion+1.wav

Graphical user interface

Description automatically generated

pianoStereo.wav

Chart, line chart

Description automatically generated

**Q2. Reading Uncompressed .png Images and Plotting RGB Histograms and Dithering**

To read the image file, I am first converting the File object to a BufferedImage object with the ImageIO library. Then, I am taking that object and passing it into a method which iterates through every pixel of the image and taking the RGB bytes with the BufferedImage.getRGB function. This returns the RGB values, along with the alpha value in a byte format, and stores it into a 2D integer array with the same dimensions as the original image. I am then passing this array into a method to convert the byte values into an array of Color objects from java.awt with the following code:

int R = (input[x][y] >> 16) & 0xFF;

int G = (input[x][y] >> 8) & 0xFF;

int B = (input[x][y]) & 0xFF;

output[x][y] = new Color(R, G, B);

The reason there is shifting is because all the values are stored in one segment of 32 bits, and I do not need to store the alpha value, so that is discarded. Each R, G, and B combo is then used to create a new Color object, and that is stored in to a 2D array with the same dimensions as the original image.