Paleokymophony – a novel approach to Audio Compression and Encryption technique

Anirban Datta Gupta
(12614001013)
Neelabja Roy
(12614001058)
Tarashankar Chakraborty
(12614001121)

4th year, Computer Science and Engineering
Heritage Institute of Technology, Kolkata

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY

Guide: **Prof. Reshma Roychoudhuri**

Department of Computer Science and Engineering,
Heritage Institute of Technology, Kolkata

HERITAGE INSTITUTE OF TECHNOLOGY, KOLKATA

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that this project report "Paleokymophony- a novel approach to audio compression and encryption" is the bonafide work of Anirban Datta Gupta, Neelabja Roy, Tarashankar Chakraborty who carried out the project work under my supervision.

SIGNATURE

Dr. Subhashis Majumder
HEAD OF THE DEPARTMENT
Department of Computer Science and Engineering
Heritage Institute of Technology,
Chowbaga Road,Anandapur,
P.O East Kolkata Township,
West Bengal,Kolkata - 700107

SIGNATURE

Prof. Reshma Roychoudhuri
PROJECT GUIDE
Department of Computer Science and Engineering
Heritage Institute of Technology,
Chowbaga Road,Anandapur,
P.O East Kolkata Township,
West Bengal,Kolkata - 700107

SIGNATURE

EXAMINER

Final Year Project for the session 2017

Final Report

.....

Place of Research:

Heritage Institute of Technology, Kolkata

Chowbaga Road, Anandapur,
P.O East Kolkata Township,

Kolkata, West Bengal 700107

Contacts

Anirban Datta Gupta
anirban.dattagupta.cse18@heritageit.edu.in

Neelabja Roy
neelabja.roy.cse18@heritageit.edu.in

Tarashankar Chakraborty
tarashankar.chakraborty.cse18@heritageit.edu.in

Mentor: Prof. Reshma Roychoudhuri
reshma.roychowdhury@heritageit.edu.in

Acknowledgements

We would first like to thank our guide, Professor Reshma Roy Chowdhury of the Department of Computer Science and Engineering at Heritage Institute of Technology, Kolkata for being the greatest help in our efforts. She was always accessible whenever we found ourselves in a spot or had any doubts about our research and writing. She encouraged self-thinking and dedicated exercises during the course of our project and was always open to unending enquiries, as well as inputs and suggestions from our side. She consistently allowed this project to be our own work, but actively steered us in the right direction whenever she thought we needed it. We are and will remain continually indebted to her contributions in our efforts for the project.

We would also like to express our gratitude to all the people who became a part of our subject group open to hearing our ideas, testing and inspecting our modules and components and thereby giving us valuable criticisms and suggestions regarding our progress all of which have helped us in progressing with our project. Without their help, we would not have been able to come this far in continuing to develop our idea.

We would also like to thank our peers and teachers at the Department of Computer Science and Engineering, Heritage Institute of Technology for their valuable inputs towards this project and also for making our duration a priceless and memorable experience.

Contents

Paleospectrophony as a part of Paleokymophony

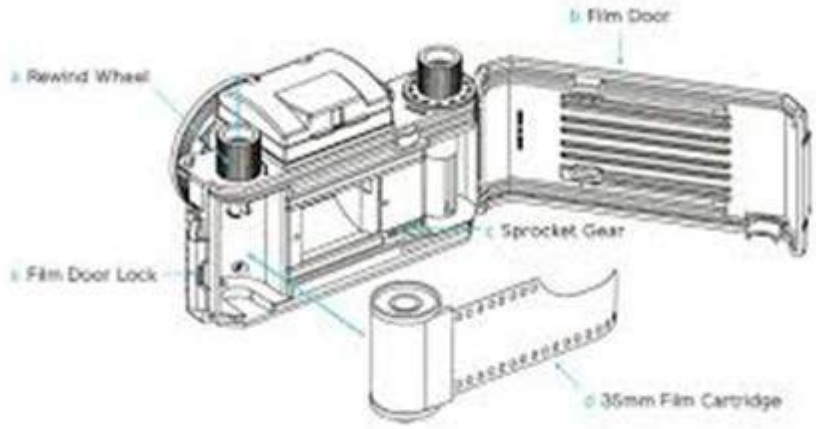
Paleokymophony derives from a latin phrase meaning ‘old world sounding’, or the art of essentially reading sound from an image as a gramophone might from an record. It involves the concept of Spectrophony, which is essentially the study of spectrograms inscribed on digital media and their analysis to reproduce a near-accurate, if not pitch perfect version of the sound it represents. The opportunity for this project came around when it became clear after research that this is not an area of sound analysis that is often frequented, and that transference of sound to visual media might present an opportunity for non-sampling-rate specific compression via image compression tactics such as utilization of PPM and Arithmetic encoding for further changes.

Inspiration behind the idea:

While attempting to come up with an idea that might have a novel application, we researched the concept behind **paleokymophony,** a latin term that deals with “old-wave sounding”, or quite literally- the usage of gramophone like analog principles in a digital setting. This seemed promising and we looked further to find that this is at best a tedious process, involving a high level of understanding in both the fields of sound engineering and software. Softwares failed to generate a faint replication of a sound wave *even if* hundreds of lines of difficult GNU/Java code and numerous libraries, not to mention hours of pixelwise manipulation in photo manipulation softwares manually to remove noise from the etching. Thus we came up with the idea of incorporating an obscure technique with the more modern concepts of encryption, to see if it was possible to build a self sufficient module that could on its own inscribe audio, encode it, freely distribute it, and decode it depending on the availability of a secret key.

ESSENTIAL COMPONENTS, PROCESSES AND TERMS

About digital image manipulation



In analog cameras , the image formation is due to the chemical reaction that takes place on the strip that is used for image formation.

A 35mm strip is used in analog camera. It is denoted in the figure by 35mm film cartridge. This strip is coated with silver halide (a chemical substance).

A 35mm strip is used in analog camera. It is denoted in the figure by 35mm film cartridge. This strip is coated with silver halide (a chemical substance).

When light photons are passed through the camera, it reacts with the silver halide particles on the strip and it results in the silver which is the negative of the image.

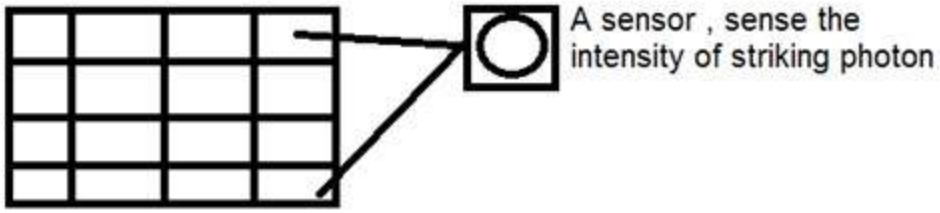
Image formation on digital cameras

In the digital cameras , the image formation is not due to the chemical reaction that takes place . Here , a CCD array of sensors is used for the image formation.

Image formation through CCD array

CCD stands for charge-coupled device. It is an image sensor, and like other sensors it senses the values and converts them into an electric signal. In case of CCD it senses the image and convert it into electric signal e.t.c.

This CCD is actually in the shape of array or a rectangular grid. It is like a matrix with each cell in the matrix contains a censor that senses the intensity of photon.



Like analog cameras , in the case of digital too , when light falls on the object , the light reflects back after striking the object and allowed to enter inside the camera.

Each sensor of the CCD array itself is an analog sensor. When photons of light strike on the chip , it is held as a small electrical charge in each photo sensor. The response of each sensor is directly equal to the amount of light or (photon) energy striked on the surface of the sensor. Since we have already define an image as a two dimensional signal and due to the two dimensional formation of the CCD array , a complete image can be achieved from this CCD array. It has limited number of sensors , and it means a limited detail can be captured by it. Also each sensor can have only one value against the each photon particle that strike on it.

So the number of photons striking(current) are counted and stored. In order to measure accurately these , external CMOS sensors are also attached with CCD array. The value of each sensor of the CCD array refers to each the value of the individual pixel. The number of sensors = number of pixels. It also means that each sensor could have only one and only one value.

Pixel is the smallest element of an image. Each pixel correspond to any one value. In an 8-bit gray scale image, the value of the pixel is between 0 and 255. The value of a pixel at any point correspond to the intensity of the light photons striking at that point. Each pixel store a value proportional to the light intensity at that particular location.

An image is nothing more than a two dimensional signal. It is defined by the mathematical function f(x,y) where x and y are the two co-ordinates horizontally and vertically.The value of f(x,y) at any point is gives the pixel value at that point of an image.

This image is nothing but a two dimensional array of numbers ranging between 0 and 255.

Image storage requirements

The size of an image depends upon three things.

- Number of rows
- Number of columns
- Number of bits per pixel

The formula for calculating the size is given below.

Size of an image = rows * cols * bpp

Assuming it has 1024 rows and it has 1024 columns. And since it is a gray scale image, it has 256 different shades of gray or it has bits per pixel. Then putting these values in the formula, we get

Size of an image = rows * cols * bpp

= 1024 * 1024 * 8

= 8388608 bits.

But since its not a standard answer that we recognize, so will convert it into our format.

Converting it into bytes = 8388608 / 8 = 1048576 bytes.

Converting into kilo bytes = 1048576 / 1024 = 1024kb.

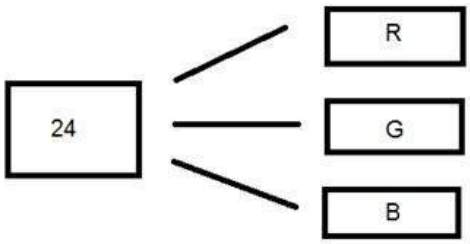
Converting into Mega bytes = 1024 / 1024 = 1 Mb.

The binary image, a 2 bit format

The binary image as it name states, contain only two pixel values *0 and 1* .In our previous tutorial of bits per pixel, we have explained this in detail about the representation of pixel values to their respective colors. Here 0 refers to black color and 1 refers to white color. It is also known as Monochrome. Such an image can be represented in the form of an 8 bit image that has no underlying gray tones or shifts. It is purely duotonic and in black and white, with a particular RGB value corresponding to the two single colours.

The RGB image, 24 bit color format

24 bit color format also known as true color format. Like 16 bit color format, in a 24 bit color format, the 24 bits are again distributed in three different formats of Red, Green and Blue.



Since 24 is equally divided on 8, so it has been distributed equally between three different color channels. Their distribution is like this:

8 bits for R, 8 bits for G, 8 bits for B.

About the Java technologies and functionalities used

Java BufferedImage class is a subclass of Image class. It is used to handle and manipulate the image data. A BufferedImage is made of ColorModel of image data. All BufferedImage objects have an upper left corner coordinate of (0, 0).

Constructors

This class supports three types of constructors.

The first constructor constructs a new BufferedImage with a specified ColorModel and Raster.

```
BufferedImage(ColorModel cm, WritableRaster raster,
boolean isRasterPremultiplied, Hashtable<?,?> properties)
```

The second constructor constructs a BufferedImage of one of the predefined image types.

```
BufferedImage(int width, int height, int imageType)
```

The third constructor constructs a BufferedImage of one of the predefined image types: TYPE_BYTE_BINARY or TYPE_BYTE_INDEXED.

```
BufferedImage(int width, int height, int imageType, IndexColorModel cm)
```

An image contains a two dimensional array of pixels. It is actually the value of those pixels that make up an image. Usually an image could be color or grayscale.

In Java, the BufferedImage class is used to handle images. You need to call **getRGB()** method of the **BufferedImage** class to get the value of the pixel.

Getting Pixel Value

The pixel value can be received using the following syntax:

```
Color c = new Color(image.getRGB(j, i));
```

Getting RGB Values

The method **getRGB()** takes the row and column index as a parameter and returns the appropriate pixel. In case of color image, it returns three values which are (Red, Green, Blue). They can be get as follows:

```
c.getRed();

c.getGreen();

c.getBlue();
```

Getting Width and Height of Image

The height and width of the image can be get by calling the **getWidth()** and **getHeight()** methods of the BufferedImage class. Its syntax is given below:

```
int width = image.getWidth();

int height = image.getHeight();
```

These are the elementary methods we will use initially.

JAVA Cipher class for simple encryption

The *Java Cipher* (`javax.crypto.Cipher`) class represents an encryption algorithm. The term *Cipher* is standard term for an encryption algorithm in the world of cryptography. That is why the Java class is called `Cipher` and not e.g. `Encrypter` / `Decrypter` or something else.

You can use a `Cipher` instance to encrypt and decrypt data in Java. This Java Cipher tutorial will explain how the `Cipher` class of the Java Cryptography API works.

Creating a Cipher

Before you can use a Java `Cipher` you just create an instance of the `Cipher` class. You create a `Cipher`instance by calling its `getInstance()` method with a parameter telling what type of encryption algorithm you want to use. Here is an example of creating a Java `Cipher` instance:

```
Cipher cipher = Cipher.getInstance("AES");
```

This example creates a `Cipher` instance using the encryption algorithm called AES.

Cipher Modes

Some encryption algorithms can work in different modes. An encryption mode specifies details about how the algorithm should encrypt data. Thus, the encryption mode impacts part of the encryption algorithm.

The encryption modes can sometimes be used with multiple different encryption algorithms - like a technique that is appended to the core encryption algorithm. That is why the modes are thought of as separate from the encryption algorithms themselves, and rather "add-ons" to the encryption algorithms. Here are some of the most wellknown cipher modes:

- EBC - Electronic Codebook
- CBC - Cipher Block Chaining
- CFB - Cipher Feedback
- OFB - Output Feedback
- CTR - Counter

When instantiating a cipher you can append its mode to the name of the encryption algorithm. For instance, to create an AES `Cipher` instance using Cipher Block Chaining (CBC) you use this code:

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Since Cipher Block Chaining requires a "padding scheme" too, the padding scheme is appended in the end of the encryption algorithm name string.

Please keep in mind that not all encryption algorithms and modes are supported by the default Java SDK cryptography provider. You might need an external provider like Bouncy Castle installed to create your desired `Cipher` instance with the required mode and padding scheme.

Initializing a Cipher

Before you can use a `Cipher` instance you must initialize it. Initializing a `Cipher` is done by calling its `init()` method. The `init()` method takes two parameters:

- Encryption / decryption cipher operation mode.
- Encryption / decryption key.

Here is an example of initializing a `Cipher` instance in encryption mode:

```
Key key = ... // get / create symmetric encryption key
cipher.init(Cipher.ENCRYPT_MODE, key);
```

Here is an example of initializing a `Cipher` instance in decryption mode:

```
Key key = ... // get / create symmetric encryption key
cipher.init(Cipher.DECRYPT_MODE, key);
```

Encrypting and Decrypting Data

In order encrypt or decrypt data with a `Cipher` instance you call one of these two methods:

- `update()`
- `doFinal()`

There are several overridden versions of both `update()` and `doFinal()` which takes different parameters. I will cover the most commonly used versions here.

If you have to encrypt or decrypt a single block of data, just call the `doFinal()` with the data to encrypt or decrypt. Here is an encryption example:

```
byte[] plainText  = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
byte[] cipherText = cipher.doFinal(plainText);
```

The code actually looks pretty much the same in case of decrypting data. Just keep in mind that the `Cipher`instance must be initialized into decryption mode. Here is how decrypting a single block of cipher text looks:

```
byte[] plainText = cipher.doFinal(cipherText);
```

If you have to encrypt or decrypt multiple blocks of data, e.g. multiple blocks from a large file, you call the `update()` once for each block of data, and finish with a call to `doFinal()` with the last data block. Here is an example of encrypting multiple blocks of data:

```
byte[] data1 = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
byte[] data2 = "zyxwvutsrqponmlkjihgfedcba".getBytes("UTF-8");
byte[] data3 = "01234567890123456789012345".getBytes("UTF-8");

byte[] cipherText1 = cipher.update(data1);
byte[] cipherText2 = cipher.update(data2);
byte[] cipherText3 = cipher.doFinal(data3);
```

The reason a call to `doFinal()` is needed for the last block of data is, that some encryption algorithms need to pad the the data to fit a certain cipher block size (e.g. an 8 byte boundary). But - we do not want to pad the intermediate blocks of data encrypted. Hence the calls to `update()` for intermediate blocks of data, and the call to `doFinal()` for the last block of data.

When decrypting multiple blocks of data you also call the `Cipher` `update()` method for intermediate data blocks, and the `doFinal()` method for the last block. Here is an example of decrypting multiple blocks of data with a Java `Cipher` instance:

```
byte[] plainText1 = cipher.update(cipherText1);
byte[] plainText2 = cipher.update(cipherText2);
byte[] plainText3 = cipher.doFinal(cipherText3);
```

Again, the `Cipher` instance must be initialized into decryption mode for this example to work.

Encrypting / Decrypting Part of a Byte Array

The Java `Cipher` class encryption and decryption methods can encrypt or decrypt part of the data stored in a `byte` array. You simply pass an offset and length to the `update()` and / or `doFinal()` method. Here is an example:

```
int offset = 10;
int length = 24;
byte[] cipherText = cipher.doFinal(data, offset, length);
```

This example will encrypt (or decrypt, depending on the initialization of the `Cipher`) from byte with index 8 and 24 bytes forward.

Encrypting / Decrypting Into an Existing Byte Array

All the encryption and decryption examples shown in this tutorial so far have been returning the encrypted or decrypted data in a new byte array. However, it is also possible to encrypt or decrypt data into an existing byte array. This can be useful to keep the number of created byte arrays down.

You can encrypt or decrypt data into an existing byte array by passing the destination byte array as parameter to the `update()` and / or `doFinal()` method. Here is an example:

```
int offset = 10;
int length = 24;
byte[] dest = new byte[1024];
cipher.doFinal(data, offset, length, dest);
```

This example encrypts the data from the byte with index 10 and 24 bytes forward into the `dest` byte array from offset 0. If you want to set a different offset for the `dest` byte array there is a version of `update()` and `doFinal()` which takes an offset parameter extra. Here is an example of calling the `doFinal()` method with an offset into the `dest` array:

```
int offset = 10;
int length = 24;
byte[] dest = new byte[1024];
int destOffset = 12
cipher.doFinal(data, offset, length, dest, destOffset);
```

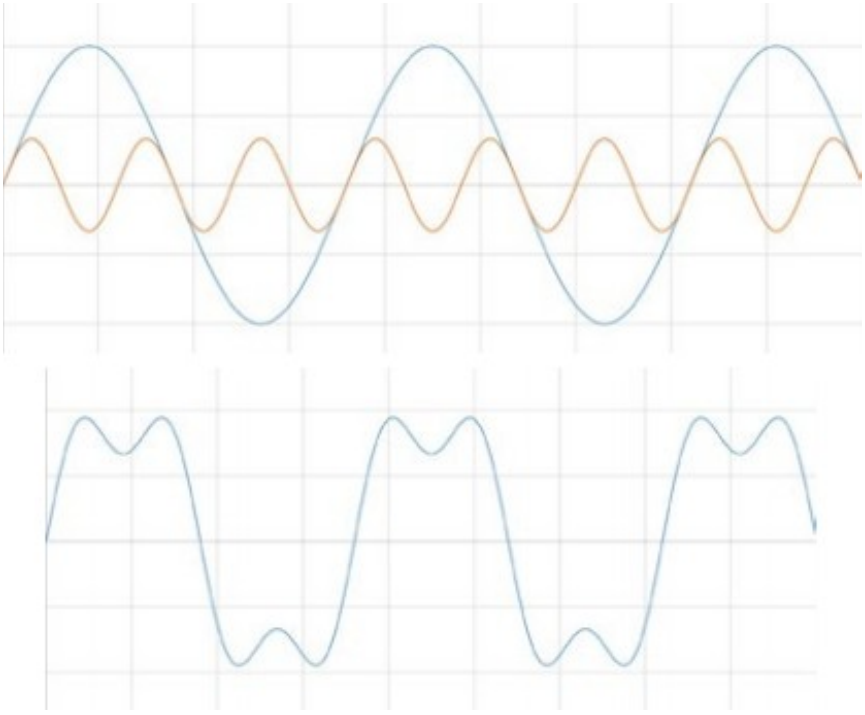
Fast Fourier Transform and Window Functions: An explanation

The Fourier transform can be powerful in understanding everyday signals and troubleshooting errors in signals. Essentially, it takes a signal and breaks it down into sine waves of different amplitudes and frequencies.

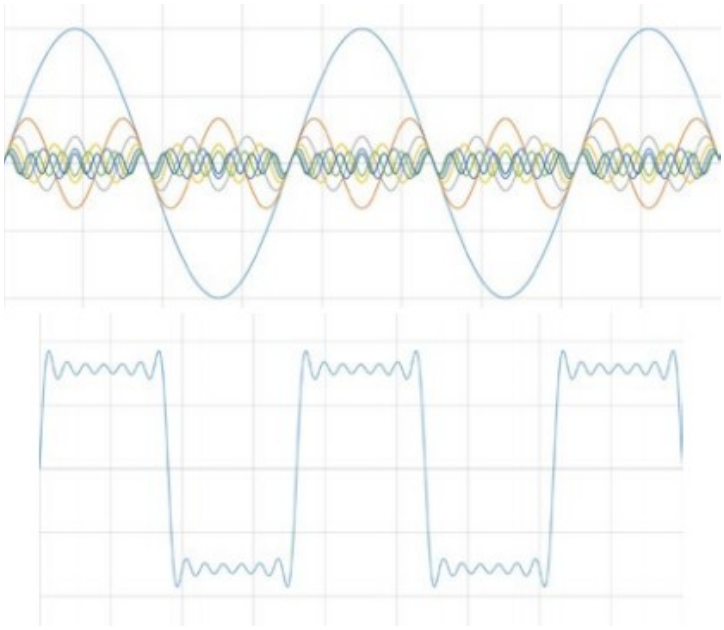
When looking at real-world signals, we usually view them as a amplitude changing over time. This is referred to as the time domain. Fourier's theorem states that any waveform in the time domain can be represented by the weighted sum of sines and cosines.

For example, take two sine waves, where one is three times as fast as the other–or the frequency is 1/3 the first signal. When you add them, you can see you get a different signal.

In the following example, second wave has also 1/3 the amplitude of the first. So just the peaks are affected.



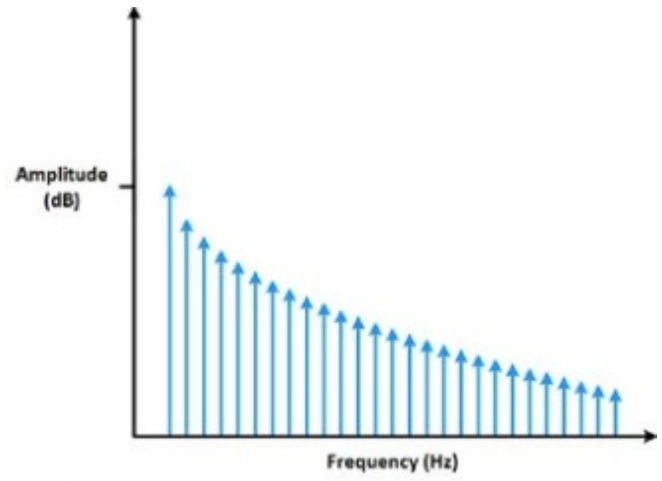
Now if we added a third signal that was 1/5 the amplitude and frequency of the original signal and continued in this fashion until we hit the noise floor, this is the waveform we get.



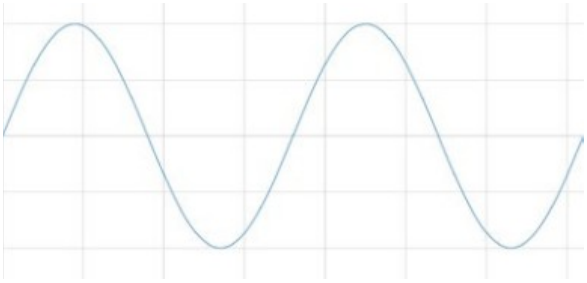
We have now created a square wave. In this way, all signals in the time domain can be represented by a series of sine.

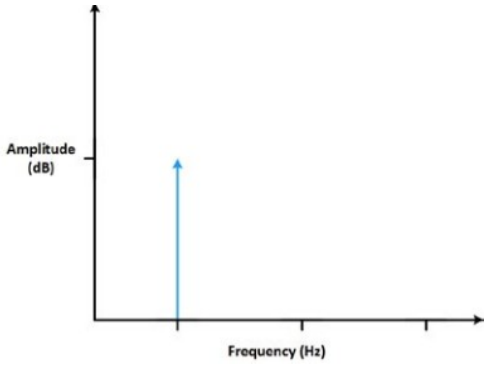
The Fourier transform deconstructs a time domain representation of a signal into the frequency domain representation. The frequency domain shows the amplitudes present at varying frequencies. It is a different way to look at the same signal. A digitizer samples a waveform and transforms it into discrete values. Because of this transformation, the Fourier transform will not work on this data. Instead, the discrete Fourier transform (DFT) is used, which produces as its result the frequency domain components in discrete values, or bins. The fast Fourier (FFT) is an optimized implementation of a DFT that takes less computation to perform but essentially just deconstructs a signal. If we take a look at the first signal from above, there are two signals at two different frequencies; in this case, the signal has two spikes in the frequency domain–one at each of the two frequencies of the sines that composed the signal in the first place.

It can also be helpful to look at the shape of the signal in the frequency domain. For instance, life we take a look at the square wave in the frequency domain, we'll see the following result. We created the square wave using many sine waves at varying frequencies; as such, you would expect many spikes in the signal in the frequency domain —one for each signal added. If you see a nice ramp in the frequency domain, you know the original signal was a square wave.

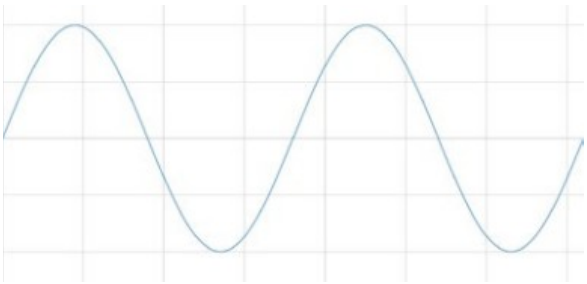


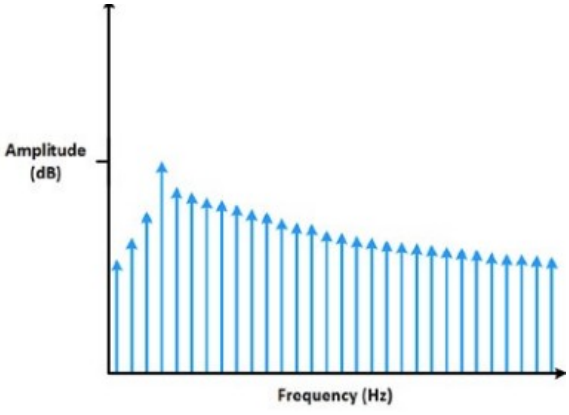
Windowing in FFT : When we use the FFT to measure the frequency component of a signal, you are basing the analysis on a finite set of data. The actual FFT transform assumes that it is a finite data set, a continuous spectrum that is one period of a periodic signal. For the FFT, both the time domain and the frequency domain are circular topologies, so the two endpoints of the time waveform are interpreted as though they were connected together. When the measured signal is periodic and an integer number of periods fill the acquisition time interval, the FFT turns out fine as it matches this assumption



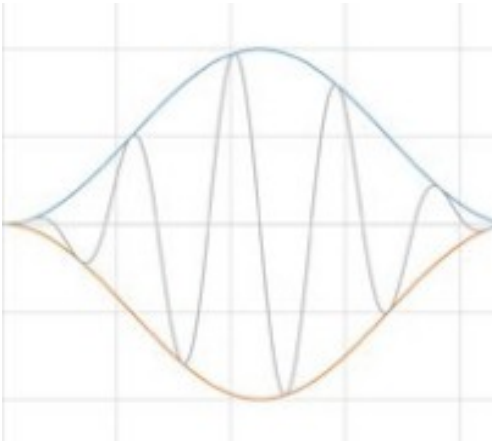


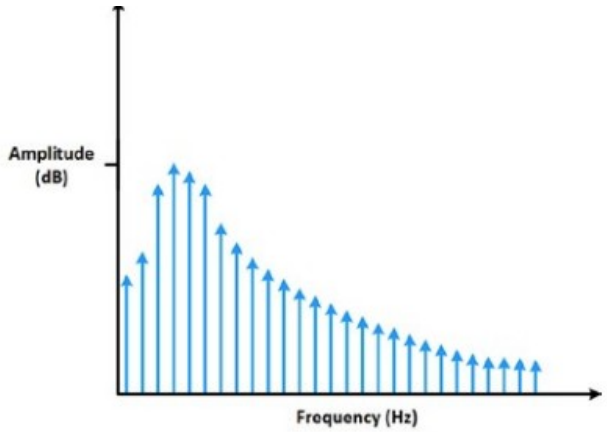
However, many times, the measured signal isn't an integer number of periods. Therefore, the finiteness of the measured signal may result in a truncated waveform with different characteristics from the original continuous-time signal, and the finiteness can introduce sharp transition changes into the measured signal. The sharp transitions are discontinuities. When the number of periods in the acquisition is not an integer, the endpoints are discontinuous. These artificial discontinuities show up in the FFT as high-frequency components not present in the original signal. These frequencies can be much higher than the Nyquist frequency and are aliased between 0 and half of your sampling rate. The spectrum you get by using a FFT, therefore, is not the actual spectrum of the original signal, but a smeared version. It appears as if energy at one frequency leaks into other frequencies. This phenomenon is known as spectral leakage, which causes the fine spectral lines to spread into wider signals.





We can minimise the effects of performing an FFT over a noninteger number of cycles by using a technique called windowing. Windowing reduces the amplitude of the discontinuities at the boundaries of each finite sequence acquired by the digitizer. Windowing consists of multiplying the time record by a finite-length window with an amplitude that varies smoothly and gradually toward zero at the edges. This makes the endpoints of the waveform meet and, therefore, results in a continuous waveform without sharp transitions. This technique is also referred to as applying a window.





Window functions: There are several different types of window functions that you can apply depending on the signal. To understand how a given window affects the frequency spectrum, we need to understand more about the frequency characteristics of windows.

An actual plot of a window shows that the frequency characteristic of a window is a continuous spectrum with a main lobe and several side lobes. The main lobe is centered at each frequency component of the time-domain signal, and the side lobes approach zero. The height of the side lobes indicates the affect the windowing function has on frequencies around main lobes. The side lobe response of a strong sinusoidal signal can overpower the main lobe response of a nearby weak sinusoidal signal. Typically, lower side lobes reduce leakage in the measured FFT but increase the bandwidth of the major lobe. The side lobe roll-off rate is the asymptotic decay rate of the side lobe peaks. By increasing the side lobe roll-off rate, we can reduce spectral leakage.

Selecting a window function is not a simple task. Each window function has its own characteristics and suitability for different applications. To choose a window function, we must estimate the frequency content of the signal.

- If the signal contains strong interfering frequency components distant from the frequency of interest, we choose a smoothing window with a high side lobe roll-off rate.

- If the signal contains strong interfering signals near the frequency of interest, we choose a window function with a low maximum side lobe level.
- If the frequency of interest contains two or more signals very near to each other, spectral resolution is important. In this case, it is best to choose a smoothing window with a very narrow main lobe.
- If the amplitude accuracy of a single frequency component is more important than the exact location of the component in a given frequency bin, choose a window with a wide main lobe.
- If the signal spectrum is rather flat or broadband in frequency content, we use the uniform window, or no window.
- In general, the Hanning (Hann) window is satisfactory in 95 percent of cases. It has good frequency resolution and reduced spectral leakage.

PPM algorithm for compression: Overview

PPM, or prediction by partial matching, is an adaptive statistical modelling technique base on blending together different length context models to predict the next character in the input sequence. The scheme achieves greater compression than Ziv-Lempel (LZ) dictionary based methods which are more widely used because of their simplicity and faster execution speeds.

PPM models are based upon context models of varying length of prior characters.The models record the frequency of characters that have followed each of the contexts. For example, a particular context may be the letters “thei”. All the letters that have followed this context are counted. The next time the letters “thei” occur in the text, the counts are used to estimate the probability for the upcoming character. The PPM technique blends together the context models for the varying lengths (such as “hei” and “ei”) to arrive at an overall probability distribution. Arithmetic coding is then used to optimally encode the character which actually does occur with respect to this distribution.

PPM splits compression into two steps. The first step accumulates a statistical model of the characters seen so far in the input text. As each character is encoded this model is used to generate a probability distribution over those characters which can occur next. PPM’s statistical model is built up from contexts of prior characters of varying lengths. A problem arises in deciding how long the contexts should be, whether there should be an upper bound to the length of the contexts or whether they should be unbounded. Another problem is deciding how much weight should be given to each context. The approach taken by PPM is a “back off” method of selecting a particular context and then backing off or escaping to a shorter context if the chosen context does not predict the upcoming character.

A second problem with probability estimation is how to encode a novel

character, which has not been seen before in the current context. This problem is essentially the zero frequency problem . PPM uses an escape probability to “escape” to another context model, usually of length one shorter than the current context. For novel characters which have never been seen before in any length model, the algorithm escapes down to a default “order -1” context model where all possible characters are equiprobable.

A typical PPM algorithm is as given:

```
begin
       while (not last character) do
       begin
           readSymbol()
           shorten context
           while (context not found and context length not -1) do
            begin
               output(escape sequence)
               shorten context
           end
          output(character)
          while (context length not -1) do
           begin
             increase count of character (create node if nonexistant)
             shorten context
           end
       end
  end
```

PPM functions post Arithmetic encoding as the long generated string of a floating point number is perfect for context extraction. In this project we utilize elements of this algorithm to compress the raw image of the spectrogram losslessly into a png format that reduces the file size drastically, though with some loss of data that we are still working to lessen.

Arithmetic Encoding: A Detailed Explanation

Arithmetic coding is a form of entropy encoding used in lossless data compression.

An *entropy encoding* is a coding scheme that involves assigning codes to symbols so as to match code lengths with the probabilities of the symbols. Typically, entropy encoders are used to compress data by replacing symbols represented by equal-length codes with symbols represented by codes proportional to the negative logarithm of the probability. Therefore, the most common symbols use the shortest codes.

Normally, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so- frequently occurring characters will be stored with more bits, resulting in fewer bits used in total. Arithmetic coding differs from other forms of entropy encoding, such as Huffman coding, in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, an arbitrary-precision fraction q where $0.0 \leq q < 1.0$. It represents the current information as a range, defined by two numbers.

In general, each step of the encoding process, except for the very last, is the same; the encoder has basically just three pieces of data to consider:

- The next symbol that needs to be encoded
- The current interval (at the very start of the encoding process, the interval is set to [0,1], but that will change)
- The probabilities the model assigns to each of the various symbols that are possible at this stage (as mentioned earlier, higher-order or adaptive models mean that these probabilities are not necessarily the same in each step.)

The encoder divides the current interval into sub-intervals, each representing a fraction of the current interval proportional to the probability of that symbol in the current context. Whichever interval corresponds to the actual symbol that is next to be encoded becomes the interval used in the next step.

In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed’to represent it becomes smaller, and the number of bits needed to specify that interval grows. Successive symbols of the message reduce the size of the

interval in accordance with the symbol probabilities generated by the model. The more likely symbols reduce the range by less than the unlikely symbols and hence add fewer bits to the message. Before anything is transmitted, the range for the message is the entire interval [0, l), denoting the half-open interval 0 5 x < 1. As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol.

For example, suppose the alphabet is (a, e, i, O, u, !) , and a fixed model is used with probabilities shown: a=0.2, e=0.3, i=0.1, O=0.2, u=0.1, !=0.1; the message to be transmitted is then provided as *eaii!* . Initially, both encoder and decoder know that the range is [0, 1). After seeing the first symbol, e, the encoder narrows it to [0.2, 04, the range the model allocates to this symbol. The second symbol, a, will narrow this new range to the first one-fifth of it, since a has been allocated [0, 0.2). This produces [0.2, 0.26], since the previous range was 0.3 units long and one-fifth of that is 0.06. The next symbol, i, is allocated [0.5, 0.6), which when applied to [0.2, 0.26) gives the smaller range [0.23, 0.236). Proceeding in this way, the encoded message builds up as follows:

Initially	[0,	1)
After seeing *e*	[0.2,	0.5)
a	[0.2,	0.26)
i	[0.23,	0.236)
i	[0.233,	0.2336)
!	[0.23354,	0.2336)

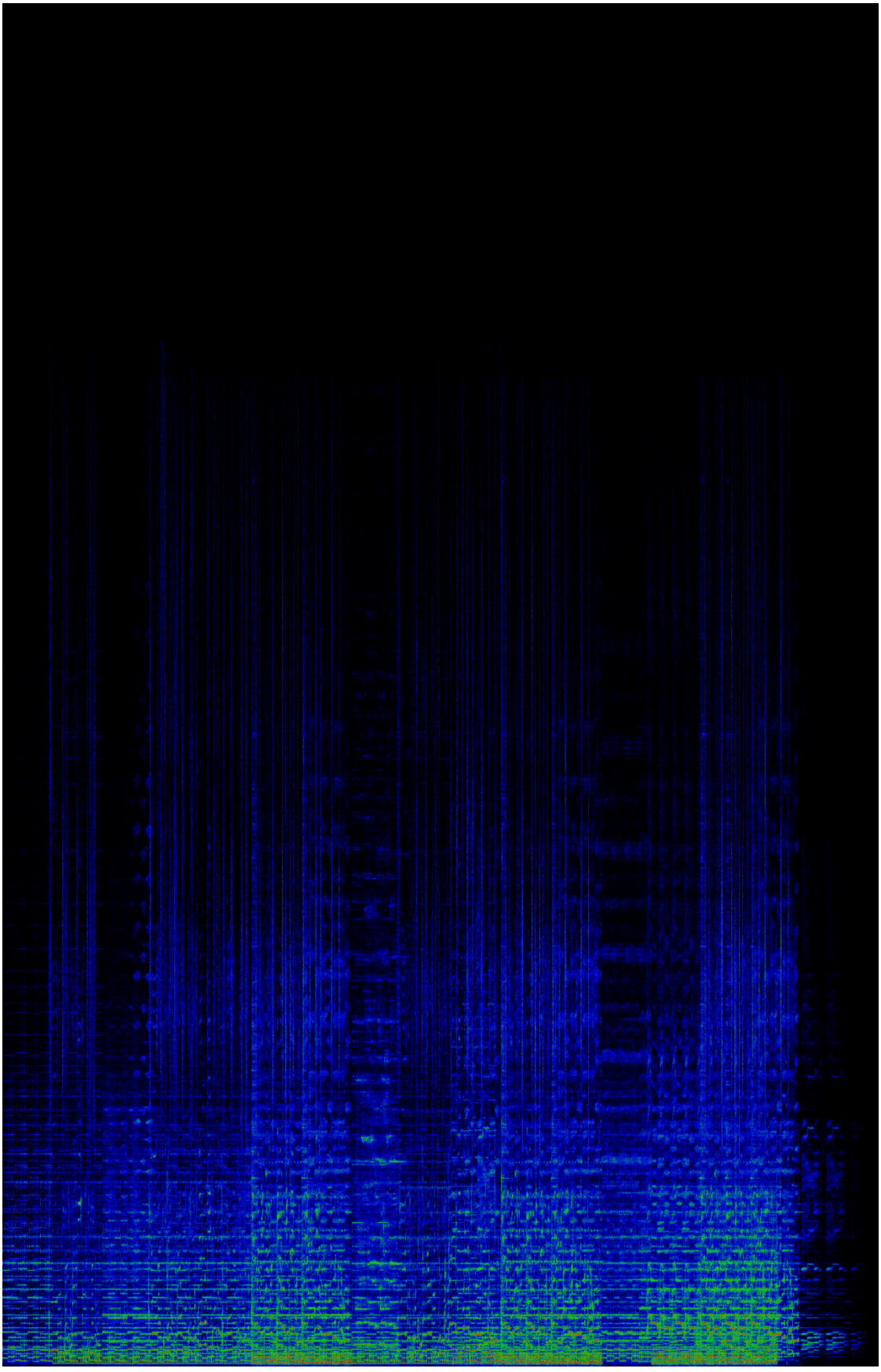
Functioning of Application: Stages

Conversion of sound file to image

Involves analysis of sound file, performing sound analysis via Fast Fourier Transformations and using a simple Hamming Window functions on the oscilloscopic graph of the wave. Code snippets are provided.

```
import javax.sound.sampled.UnsupportedAudioFileException;
import java.io.File;

import java.io.IOException;


public class WavToSpectro {
    public static void main (String[] args) throws IOException,UnsupportedAudioFileException {


        String filename = "/Users/stoiclseuth/Desktop/viva.wav";
        String outFolder = "out";

        convertatos convatos = new convertatos();

        convatos.convert(new File(filename));

    }
}


public class convertatos  {
    public void convert(File file ) throws IOException, UnsupportedAudioFileException {
```

```
        final Spectrogram spectrogram = new Spectrogram(4410,
                0.5,WindowFunction.HAMMING, false, false, false);

        Signal signal= Signal.fromFile(file);


        final String name = file.getName();
        spectrogram.save( new File("/Users/stoiclseuth/Desktop/image.png"), "png", signal);

        };


    }
public void save(File f, String ext, Signal signal) throws IOException {


    int n = signal.getNumSamples();
    int sw = (int)getSpectrumWidth();
    int width = n / sw + 1;


    BufferedImage image = new BufferedImage(width, bins,
            BufferedImage.TYPE_INT_RGB);
    for (int x = 0; x < width; x++) {
      Spectrum spectrum = signal.getSpectrum(x * sw, bins, window, showPhase);
        for (int y = 0; y < bins; y++) {

            int s = bins - (logAxis ? lbins[y] : y) - 1;
            double v = spectrum.get(s);

            Color c = showPhase ? phaseColor(v) : powerColor(v);
            image.setRGB(x, y, c.getRGB());

        }
    }
    ImageIO.write(image, ext, f);
  }
```



A spectrogram version of a popular song of length greater than 4 minutes

Compression and Encryption of image

Arithmetic encoding is utilized `to generate the final compressed and encrypted text file that is to be sent over networks. The file is comparatively lightweight and impossible to read without having a copy of the desktop application to decrypt it. It is also a lossless process and no data is lost during the process. Code snippets are provided.

```
package ColouredFileCompression;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;



public class LockUnlock

{


    public byte[] readFile(String file) throws IOException

    {

     File file1 = new File(file);

        FileInputStream fin = null;


            fin = new FileInputStream(file1);

            byte fileContent[] = new byte[(int)file1.length()];


            fin.read(fileContent);



    return fileContent;


}




    public void encryptor(byte[] str, String k) throws Exception

    {

    Key aesKey = new SecretKeySpec(k.getBytes(), "AES");
```

```
        Cipher cipher = Cipher.getInstance("AES");


        cipher.init(Cipher.ENCRYPT_MODE, aesKey);
        byte[] encrypted = cipher.doFinal(str);
            try (FileOutputStream fos = new FileOutputStream("piou.txt")) {
  fos.write(encrypted);}


}
   public static void main(String[] args) throws Exception
   {
      LockUnlock app = new LockUnlock();
      byte[] fc=app.readFile("popped.txt");
      app.encryptor(fc, "Bar12345Bar12345");


      try
      {byte[] fc1=app.readFile("piou.txt");
      app.decryptor(fc1, "Bar12345Bar12345");
      }
      catch(Exception e)
      {
         System.out.println("Wrong password");
      }
   }
}
public class AdaptiveArithmeticCompress {

        public static void main(String[] args) throws IOException {

                File inputFile  = new File("yel_test.pdf");
                File outputFile = new File("popped.txt");


                try (InputStream in = new BufferedInputStream(new FileInputStream(inputFile));
                                BitOutputStream out = new BitOutputStream(new
BufferedOutputStream(new FileOutputStream(outputFile)))) {
                        compress(in, out);
                }
        }

public abstract class ArithmeticCoderBase {
```

```
	protected final long STATE_SIZE = 32;

	protected final long MAX_RANGE = 1L << STATE_SIZE;

	protected final long MIN_RANGE = (MAX_RANGE >>> 2) + 2;

	protected final long MAX_TOTAL = Math.min(Long.MAX_VALUE / MAX_RANGE,
MIN_RANGE);

	protected final long MASK = MAX_RANGE - 1;

	protected final long TOP_MASK = MAX_RANGE >>> 1;
	protected final long SECOND_MASK = TOP_MASK >>> 1;
	protected long low;
	protected long high;

	public ArithmeticCoderBase() {
		low = 0;
		high = MASK;
	}

	protected void update(CheckedFrequencyTable freqs, int symbol) throws IOException {

		if (low >= high || (low & MASK) != low || (high & MASK) != high)
			throw new AssertionError("Low or high out of range");
		long range = high - low + 1;
		if (range < MIN_RANGE || range > MAX_RANGE)
			throw new AssertionError("Range out of range");
```

```
                        long total = freqs.getTotal();
                long symLow = freqs.getLow(symbol);
                long symHigh = freqs.getHigh(symbol);
                if (symLow == symHigh)
                        throw new IllegalArgumentException("Symbol has zero frequency");
                if (total > MAX_TOTAL)
                        throw new IllegalArgumentException("Cannot code symbol because total is too large");

                long newLow  = low + symLow  * range / total;
                long newHigh = low + symHigh * range / total - 1;
                low = newLow;
                high = newHigh;

                while (((low ^ high) & TOP_MASK) == 0) {
                        shift();
                        low = (low << 1) & MASK;
                        high = ((high << 1) & MASK) | 1;
                }

                while ((low & ~high & SECOND_MASK) != 0) {
                        underflow();
                        low = (low << 1) & (MASK >>> 1);
                        high = ((high << 1) & (MASK >>> 1)) | TOP_MASK | 1;
                }
        }


        protected abstract void shift() throws IOException;
        protected abstract void underflow() throws IOException;

}
```

```
public class ArithmeticCompress {

	public static void main(String[] args) throws IOException {
		if (args.length != 2) {
			System.err.println("Usage: java ArithmeticCompress InputFile
OutputFile");
			System.exit(1);
			return;
		}
		File inputFile  = new File(args[0]);
		File outputFile = new File(args[1]);*/
	 File inputFile  = new File("popped.txt");
		File outputFile = new File("fir.txt");

		FrequencyTable freqs = getFrequencies(inputFile);
		freqs.increment(256);
		try (InputStream in = new BufferedInputStream(new FileInputStream(inputFile));
				BitOutputStream out = new BitOutputStream(new
BufferedOutputStream(new FileOutputStream(outputFile)))) {
			writeFrequencies(out, freqs);
			compress(freqs, in, out);
		}
	}

	private static FrequencyTable getFrequencies(File file) throws IOException {
		FrequencyTable freqs = new SimpleFrequencyTable(new int[257]);
		try (InputStream input = new BufferedInputStream(new FileInputStream(file))) {
			while (true) {
				int b = input.read();
				if (b == -1)
					break;
				freqs.increment(b);
			}
```

```
		}
		return freqs;
	}


	static void writeFrequencies(BitOutputStream out, FrequencyTable freqs) throws
IOException {
		for (int i = 0; i < 256; i++)
			writeInt(out, 32, freqs.get(i));
	}


static void compress(FrequencyTable freqs, InputStream in, BitOutputStream out) throws
IOException {
		ArithmeticEncoder enc = new ArithmeticEncoder(out);
		while (true) {
			int symbol = in.read();
			if (symbol == -1)
				break;
			enc.write(freqs, symbol);
		}
		enc.write(freqs, 256);  // EOF
		enc.finish();
	}


	private static void writeInt(BitOutputStream out, int numBits, int value) throws
IOException {
		if (numBits < 0 || numBits > 32)
			throw new IllegalArgumentException();

		for (int i = numBits - 1; i >= 0; i--)
			out.write((value >>> i) & 1);		}

}
```

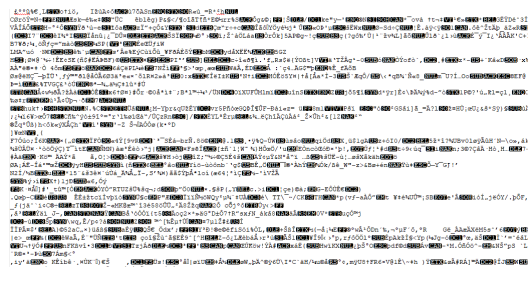
```
public final class CheckedFrequencyTable implements FrequencyTable {


	private FrequencyTable freqTable;

	public CheckedFrequencyTable(FrequencyTable freq) {
		Objects.requireNonNull(freq);
		freqTable = freq;
	}


	public int getSymbolLimit() {
		int result = freqTable.getSymbolLimit();
		if (result <= 0)
			throw new AssertionError("Non-positive symbol limit");
		return result;
	}


	public int get(int symbol) {
		int result = freqTable.get(symbol);
		if (!isSymbolInRange(symbol))
			throw new AssertionError("IllegalArgumentException expected");
		if (result < 0)
			throw new AssertionError("Negative symbol frequency");
		return result;
	}


	public int getTotal() {
		int result = freqTable.getTotal();
		if (result < 0)
			throw new AssertionError("Negative total frequency");
		return result;
```

```
	}


	public int getLow(int symbol) {
		if (isSymbolInRange(symbol)) {
			int low   = freqTable.getLow (symbol);
			int high  = freqTable.getHigh(symbol);
			if (!(0 <= low && low <= high && high <= freqTable.getTotal()))
				throw new AssertionError("Symbol low cumulative frequency out of
range");
			return low;
		} else {
			freqTable.getLow(symbol);
			throw new AssertionError("IllegalArgumentException expected");
		}
	}


	public int getHigh(int symbol) {
		if (isSymbolInRange(symbol)) {
			int low   = freqTable.getLow (symbol);
			int high  = freqTable.getHigh(symbol);
			if (!(0 <= low && low <= high && high <= freqTable.getTotal()))
				throw new AssertionError("Symbol high cumulative frequency out
of range");
			return high;
		} else {
			freqTable.getHigh(symbol);
			throw new AssertionError("IllegalArgumentException expected");
		}
	}


	public String toString() {
		return "CheckFrequencyTable (" + freqTable.toString() + ")";
	}
```

```
	public void set(int symbol, int freq) {
		freqTable.set(symbol, freq);
		if (!isSymbolInRange(symbol) || freq < 0)
			throw new AssertionError("IllegalArgumentException expected");
	}


	public void increment(int symbol) {
		freqTable.increment(symbol);
		if (!isSymbolInRange(symbol))
			throw new AssertionError("IllegalArgumentException expected");
	}


	private boolean isSymbolInRange(int symbol) {
		return 0 <= symbol && symbol < getSymbolLimit();
	}

}

public final class PpmCompress {

	private static final int MODEL_ORDER = 3;


	public static void main(String[] args) throws IOException {
		if (args.length != 2) {
			System.err.println("Usage: java PpmCompress InputFile OutputFile");
			System.exit(1);
			return;
		}
		File inputFile  = new File(args[0]);
		File outputFile = new File(args[1]);*/
```

```
        File inputFile  = new File("yel_test.pdf");
                File outputFile = new File("whoop.txt");



                try (InputStream in = new BufferedInputStream(new FileInputStream(inputFile)))
{
                        try (BitOutputStream out = new BitOutputStream(new
BufferedOutputStream(new FileOutputStream(outputFile)))) {
                                compress(in, out);
                        }
                }
        }



                static void compress(InputStream in, BitOutputStream out) throws IOException {
                ArithmeticEncoder enc = new ArithmeticEncoder(out);
                PpmModel model = new PpmModel(MODEL_ORDER, 257, 256);
                int[] history = new int[0];

                while (true) {
                        int symbol = in.read();
                        if (symbol == -1)
                                break;
                        encodeSymbol(model, history, symbol, enc);
                        model.incrementContexts(history, symbol);

                        if (model.modelOrder >= 1) {
                                if (history.length < model.modelOrder)
                                        history = Arrays.copyOf(history, history.length + 1);
                                else
                                        System.arraycopy(history, 1, history, 0, history.length - 1);
                                history[history.length - 1] = symbol;
                        }
                }

                encodeSymbol(model, history, 256, enc);
```

```
            enc.finish();
        }


        private static void encodeSymbol(PpmModel model, int[] history, int symbol,
ArithmeticEncoder enc) throws IOException {
            outer:
            for (int order = Math.min(history.length, model.modelOrder); order >= 0; order--)
{
                PpmModel.Context ctx = model.rootContext;
                for (int i = history.length - order; i < history.length; i++) {
                    if (ctx.subcontexts == null)
                        throw new AssertionError();
                    ctx = ctx.subcontexts[history[i]];
                    if (ctx == null)
                        continue outer;
                }
                if (symbol != 256 && ctx.frequencies.get(symbol) > 0) {
                    enc.write(ctx.frequencies, symbol);
                    return;
                }

                enc.write(ctx.frequencies, 256);
            }

            enc.write(model.orderMinus1Freqs, symbol);
        }
}
```



An encrypted and compressed text file created at the end of this phase

Decompression and Decryption of text

The above process is reversed.
Code snippets are provided.

```
package ColouredFileCompression;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;

public class LockUnlock
{

   public byte[] readFile(String file) throws IOException
   {
    File file1 = new File(file);
      FileInputStream fin = null;

          fin = new FileInputStream(file1);
          byte fileContent[] = new byte[(int)file1.length()];

          fin.read(fileContent);


   return fileContent;

}



}
public void decryptor(byte[] str, String k) throws Exception
   {

      Key aesKey = new SecretKeySpec(k.getBytes(), "AES");
         Cipher cipher = Cipher.getInstance("AES");
   cipher.init(Cipher.DECRYPT_MODE, aesKey);
          String decrypted = new String(cipher.doFinal(str));

         System.out.println(decrypted);

}
   public static void main(String[] args) throws Exception
```

```
   {
      LockUnlock app = new LockUnlock();
      byte[] fc=app.readFile("popped.txt");


      try
      {byte[] fc1=app.readFile("piou.txt");
      app.decryptor(fc1, "Bar12345Bar12345");
      }
      catch(Exception e)
      {
         System.out.println("Wrong password");
      }
   }
}

public class AdaptiveArithmeticDecompress {

        public static void main(String[] args) throws IOException {

                if (args.length != 2) {
                        System.err.println("Usage: java AdaptiveArithmeticDecompress InputFile
OutputFile");
                        System.exit(1);
                        return;
                }
                File inputFile  = new File("PPPP.txt");
                File outputFile = new File("OUTPUT.png");

                // Perform file decompression
                try (BitInputStream in = new BitInputStream(new BufferedInputStream(new
FileInputStream(inputFile)));
                                OutputStream out = new BufferedOutputStream(new
FileOutputStream(outputFile))) {
                        decompress(in, out);
                }
        }


        .
        static void decompress(BitInputStream in, OutputStream out) throws IOException {
                FlatFrequencyTable initFreqs = new FlatFrequencyTable(257);
                FrequencyTable freqs = new SimpleFrequencyTable(initFreqs);
                ArithmeticDecoder dec = new ArithmeticDecoder(in);
                while (true) {
                        // Decode and write one byte
                        int symbol = dec.read(freqs);
                        if (symbol == 256)
                                break;
                        out.write(symbol);
                        freqs.increment(symbol);
                }
        }

}



public final class ArithmeticDecoder extends ArithmeticCoderBase {

        private BitInputStream input;
```

```
	private long code;



	public ArithmeticDecoder(BitInputStream in) throws IOException {
		super();
		Objects.requireNonNull(in);
		input = in;
		code = 0;
		for (int i = 0; i < STATE_SIZE; i++)
			code = code << 1 | readCodeBit();
	}



	public int read(FrequencyTable freqs) throws IOException {
		return read(new CheckedFrequencyTable(freqs));
	}



	public int read(CheckedFrequencyTable freqs) throws IOException {
		long total = freqs.getTotal();
		if (total > MAX_TOTAL)
			throw new IllegalArgumentException("Cannot decode symbol because
total is too large");
		long range = high - low + 1;
		long offset = code - low;
		long value = ((offset + 1) * total - 1) / range;
		if (value * range / total > offset)
			throw new AssertionError();
		if (value < 0 || value >= total)
			throw new AssertionError();


		.
		int start = 0;
		int end = freqs.getSymbolLimit();
		while (end - start > 1) {
			int middle = (start + end) >>> 1;
			if (freqs.getLow(middle) > value)
				end = middle;
			else
				start = middle;
		}
		if (start + 1 != end)
			throw new AssertionError();

		int symbol = start;
		if (offset < freqs.getLow(symbol) * range / total || freqs.getHigh(symbol) *
range / total <= offset)
			throw new AssertionError();
		update(freqs, symbol);
		if (code < low || code > high)
			throw new AssertionError("Code out of range");
		return symbol;
	}


	protected void shift() throws IOException {
```

```
            code = ((code << 1) & MASK) | readCodeBit();
        }

        protected void underflow() throws IOException {
            code = (code & TOP_MASK) | ((code << 1) & (MASK >>> 1)) | readCodeBit();
        }


        private int readCodeBit() throws IOException {
            int temp = input.read();
            if (temp == -1)
                temp = 0;
            return temp;
        }

}

public class ArithmeticDecompress {


        public static void main(String[] args) throws IOException {


            File inputFile  = new File(args[0]);
            File outputFile = new File(args[1]);



            try (BitInputStream in = new BitInputStream(new BufferedInputStream(new
FileInputStream(inputFile)));
                        OutputStream out = new BufferedOutputStream(new
FileOutputStream(outputFile))) {
                FrequencyTable freqs = readFrequencies(in);
                decompress(freqs, in, out);
            }
        }



            static FrequencyTable readFrequencies(BitInputStream in) throws IOException {
            int[] freqs = new int[257];
            for (int i = 0; i < 256; i++)
                freqs[i] = readInt(in, 32);
            freqs[256] = 1;  // EOF symbol
            return new SimpleFrequencyTable(freqs);
        }
```

```
static void decompress(FrequencyTable freqs, BitInputStream in, OutputStream out) throws
IOException {
		ArithmeticDecoder dec = new ArithmeticDecoder(in);
		while (true) {
			int symbol = dec.read(freqs);
			if (symbol == 256)  // EOF symbol
				break;
			out.write(symbol);
		}
	}


private static int readInt(BitInputStream in, int numBits) throws IOException {
		if (numBits < 0 || numBits > 32)
			throw new IllegalArgumentException();

		int result = 0;
		for (int i = 0; i < numBits; i++)
			result = (result << 1) | in.readNoEof();  // Big endian
		return result;
	}

}
public final class PpmDecompress {

	private static final int MODEL_ORDER = 3;
	public static void main(String[] args) throws IOException {

		if (args.length != 2) {
			System.err.println("Usage: java PpmDecompress InputFile OutputFile");
			System.exit(1);
			return;
		}
		File inputFile  = new File(args[0]);
		File outputFile = new File(args[1]);*/
```

```
        File inputFile  = new File("music.mp3");
                File outputFile = new File("whoop1.txt");

                try (BitInputStream in = new BitInputStream(new BufferedInputStream(new
FileInputStream(inputFile)))) {
                        try (OutputStream out = new BufferedOutputStream(new
FileOutputStream(outputFile))) {
                                decompress(in, out);
                        }
                }
        }

static void decompress(BitInputStream in, OutputStream out) throws IOException {
                                ArithmeticDecoder dec = new ArithmeticDecoder(in);
                PpmModel model = new PpmModel(MODEL_ORDER, 257, 256);
                int[] history = new int[0];

                while (true) {
                        int symbol = decodeSymbol(dec, model, history);
                        if (symbol == 256)  // EOF symbol
                                break;
                        out.write(symbol);
                        model.incrementContexts(history, symbol);

                        if (model.modelOrder >= 1) {
                                if (history.length < model.modelOrder)
                                        history = Arrays.copyOf(history, history.length + 1);
                                else
                                        System.arraycopy(history, 1, history, 0, history.length - 1);
                                history[history.length - 1] = symbol;
                        }
                }
        }


        private static int decodeSymbol(ArithmeticDecoder dec, PpmModel model, int[] history)
throws IOException {
```

```
                outer:
                for (int order = Math.min(history.length, model.modelOrder); order >= 0;
order--) {
                        PpmModel.Context ctx = model.rootContext;
                        for (int i = history.length - order; i < history.length; i++) {
                                if (ctx.subcontexts == null)
                                        throw new AssertionError();
                                ctx = ctx.subcontexts[history[i]];
                                if (ctx == null)
                                        continue outer;
                        }
                        int symbol = dec.read(ctx.frequencies);
                        if (symbol < 256)
                                return symbol;

                }
                return dec.read(model.orderMinus1Freqs);
        }

}
```

Conversion of image to sound file

This is reverse of the first step, in which the spectrogram is converted back to the audio, to reconstruct the original audio signal.

```
public class SpectroToAudio {
   public static void main (String[] args) throws IOException {


      String filename = “/Users/stoiclseuth/Desktop/image.png";
      String outFolder = "/Users/stoiclseuth/Desktop";
```

```
        convertstoa convsa = new convertstoa();


        convsa.convert(new File(filename), outFolder+"/antinational.wav");
    }
}
public class convertstoa {
    public void convert(File filename, String outfile) throws IOException {
        int width = 963;    //width of the image
        int height = 640; //height of the image


        BufferedImage image = null;


                File input_file = new File("G:\\intellij_java\\SpectrogramToAudio\\spec_pic\\ok_wav2.jpg"); //
image file path


            image = new BufferedImage(width, height,
                    BufferedImage.TYPE_INT_ARGB);


            image = ImageIO.read(filename);


            System.out.println("Reading complete.");


        int hertz = 44100;
        double overlap = 0.5;


        final int hz = hertz;
        final double lap = overlap;


        Spectrogram.imageToAudio( image, new File(outfile),hz, lap);
```

```
}

public static void imageToAudio( BufferedImage image,
                        File f, int hertz, double overlap) throws IOException {
    String name = f.getName();
    int cols = image.getWidth();
    int bins = image.getHeight();


    double maxPower = bins * bins / 4.0;


    int sw = (int)(bins * 2 * overlap);
    double[] samples = new double[sw * cols];
    double[] col = new double[bins * 2];

    DoubleFFT_1D fft = new DoubleFFT_1D(col.length);
    int index = 0;

    for (int c = 0; c < cols; c++) {


        for (int r = 0; r < bins; r++) {
            int rgb = image.getRGB(c, bins - r - 1);
            double power = getPower(rgb, maxPower);
            double amplitude = Math.sqrt(power);

            double phase = Math.random() * TAU - Math.PI;
            col[2*r] = amplitude * Math.cos(phase);
            col[2*r+1] = amplitude * Math.sin(phase);

        }
        fft.realInverse(col, true);
        int start = 0; // keep beginning samples
        int end = start + sw;

        for (int s = start; s < end; s++)
            { samples[index++] = col[s] *
            2;

        }
    }
```

```
        SampledMemoryData data = new SampledMemoryData(samples);
        AudioInputStream stream = data.toStream(hertz);
        AudioSystem.write(stream, AudioFileFormat.Type.WAVE, f);

    }

public class Spectrogram {
    private static final long serialVersionUID = -5442088111430822270L;

    private static final double TAU = Math.PI * 2;
    private static final int MAX_COLOR = 0xFF;

    private boolean logAxis, showPhase;
    private int bins, lbins[];
    private double overlap, maxPower, step, cf;
    //private WindowFunction window;

    private static double getPower(int rgb, double maxPower) {
        double step = Math.log1p(maxPower) / 4;
        double cf = MAX_COLOR / step;
        int r = (rgb >> 16) & 0xFF;
        int g = (rgb >> 8) & 0xFF;
        int b = rgb & 0xFF;
        double k = 0.0;
        if (r > 0 && g > 0 && b > 0)
            k = g / cf + step * 3;
        else if (r > 0)
            k = r / cf + step * 2;
        else if (g > 0)
            k = g / cf + step;
        else
            k = b / cf;
        return Math.expm1(k);
    }

    public static void imageToAudio( BufferedImage image,
                                     File f, int hertz, double overlap) throws IOException {
        String name = f.getName();
        int cols = image.getWidth();
```

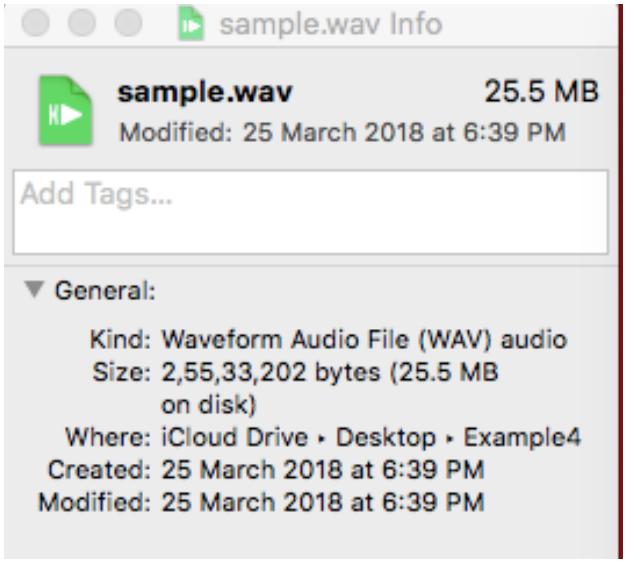
```
        int bins = image.getHeight();

        double maxPower = bins * bins / 4.0;
        //double freqF = (double)hertz / bins / 2; // unused
        int sw = (int)(bins * 2 * overlap);
        double[] samples = new double[sw * cols];
        double[] col = new double[bins * 2];
        DoubleFFT_1D fft = new DoubleFFT_1D(col.length);
        int index = 0;
        for (int c = 0; c < cols; c++) {

            for (int r = 0; r < bins; r++) {
                int rgb = image.getRGB(c, bins - r - 1);
                double power = getPower(rgb, maxPower);
                double amplitude = Math.sqrt(power);
                //double frequency = r * freqF; // unused
                double phase = Math.random() * TAU - Math.PI;
                col[2*r] = amplitude * Math.cos(phase);
                col[2*r+1] = amplitude * Math.sin(phase);
            }
            fft.realInverse(col, true);
            int start = 0; // keep beginning samples
            int end = start + sw;
            for (int s = start; s < end; s++) {
                samples[index++] = col[s] * 2;
            }
        }

        SampledMemoryData data = new SampledMemoryData(samples);
        AudioInputStream stream = data.toStream(hertz);
        AudioSystem.write(stream, AudioFileFormat.Type.WAVE, f);

    }


}
```

Encoded text file

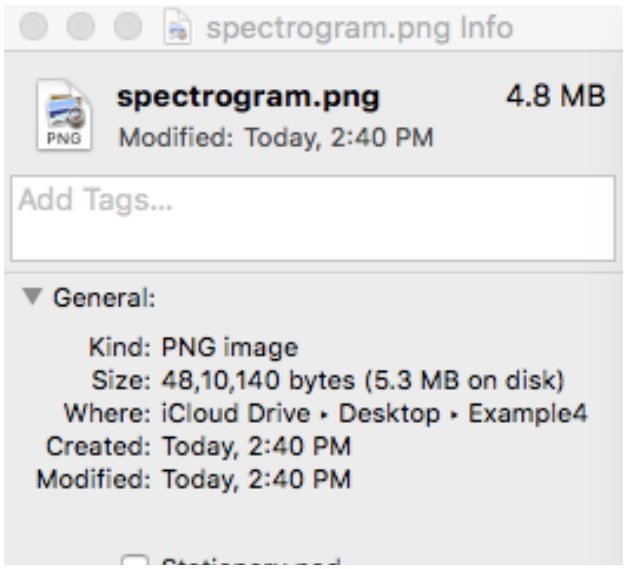
Test Cases

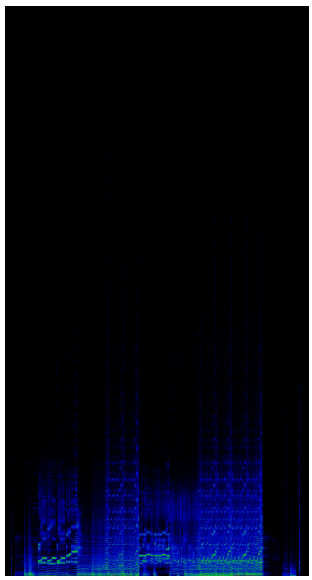
1. Test Case I

Original File

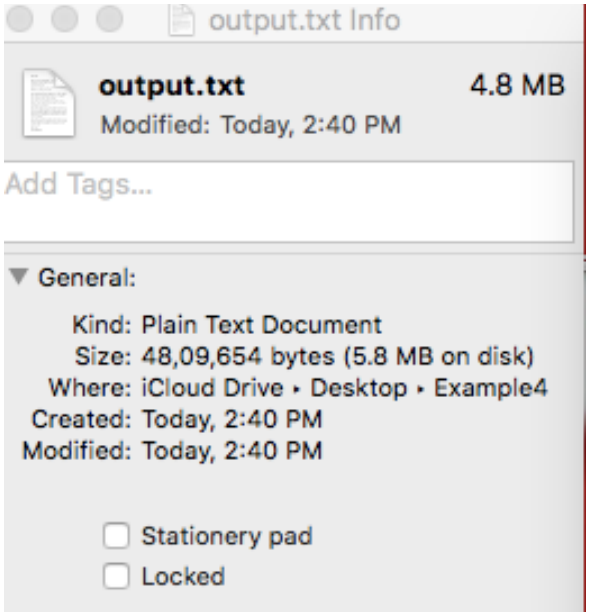


Generated Spectrogram

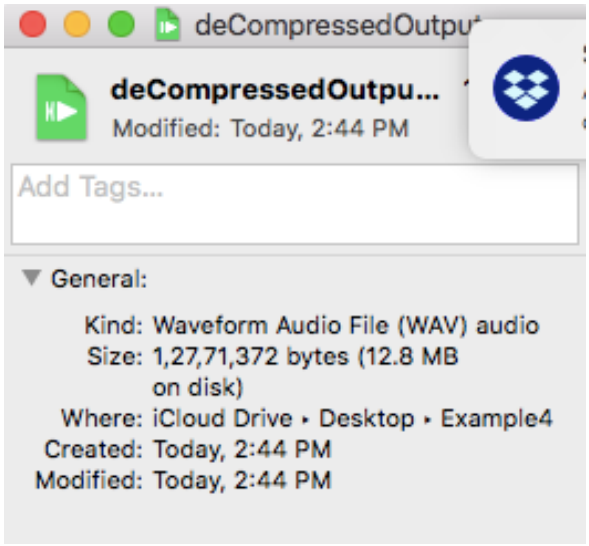




Encoded Text Size



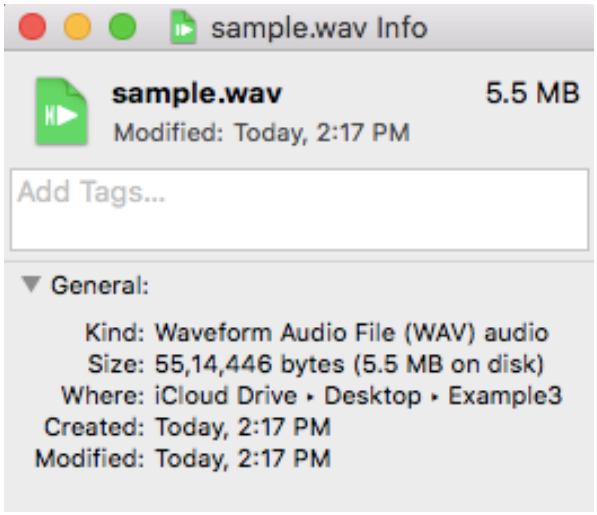
Uncompressed Audio File Size



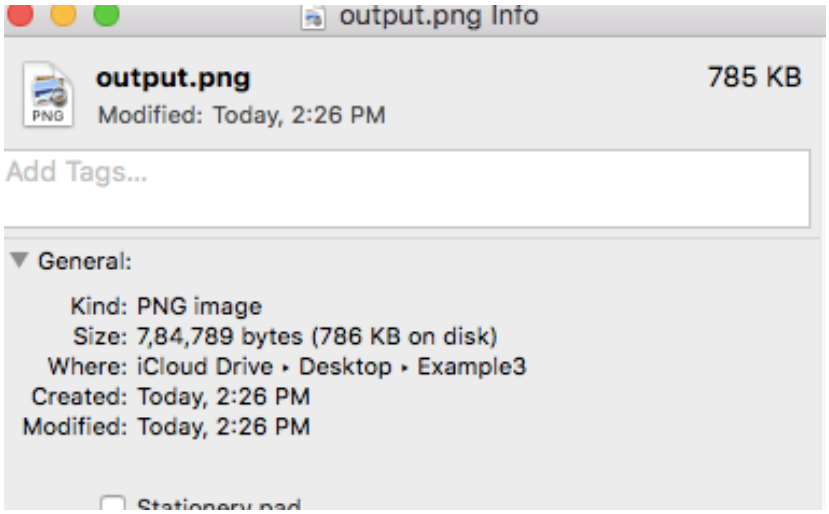
Test Cases

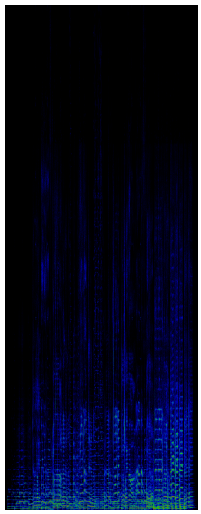
Test Case II

Original File

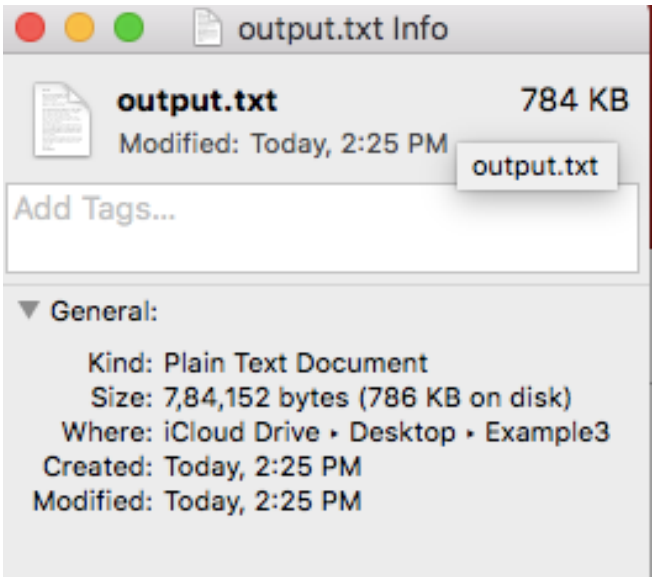


Generated Spectrogram

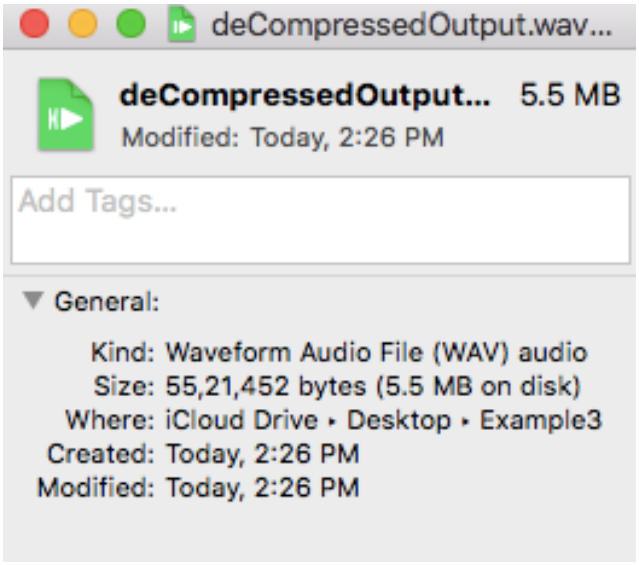




Encoded Text Size



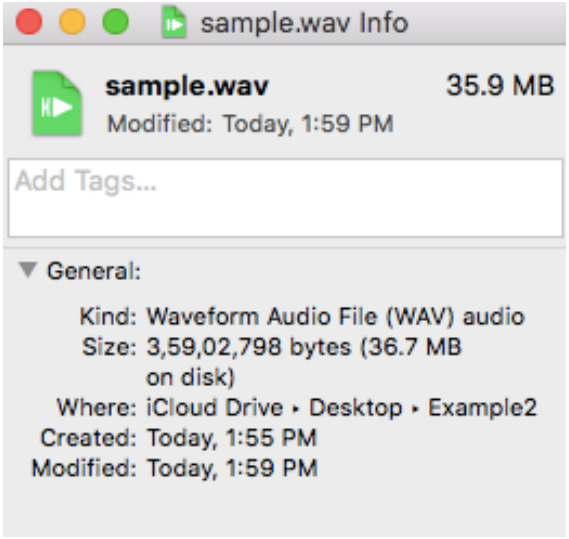
Uncompressed Audio File Size



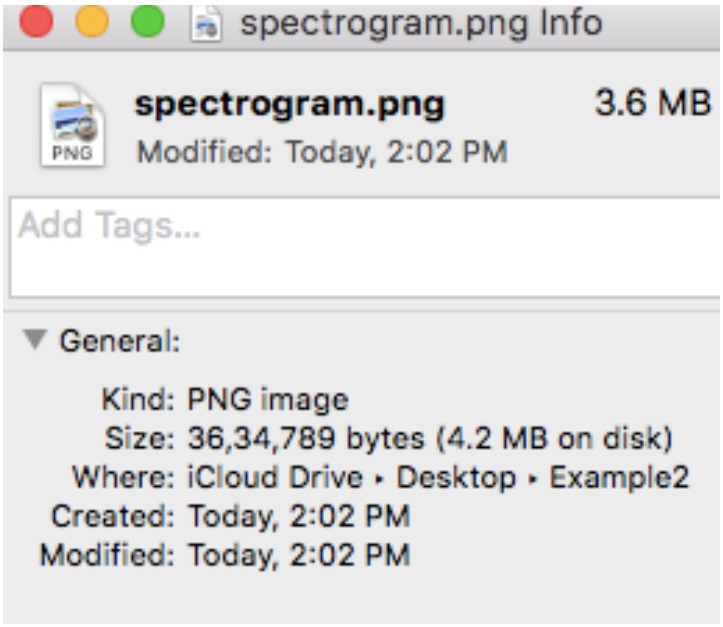
Test Cases

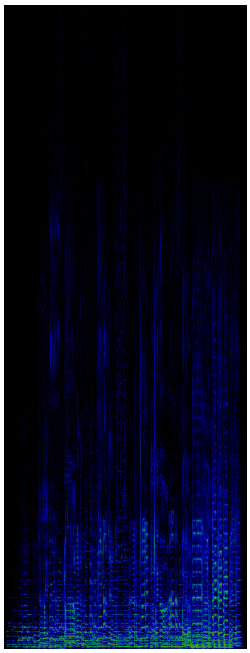
Test Case III

Original File

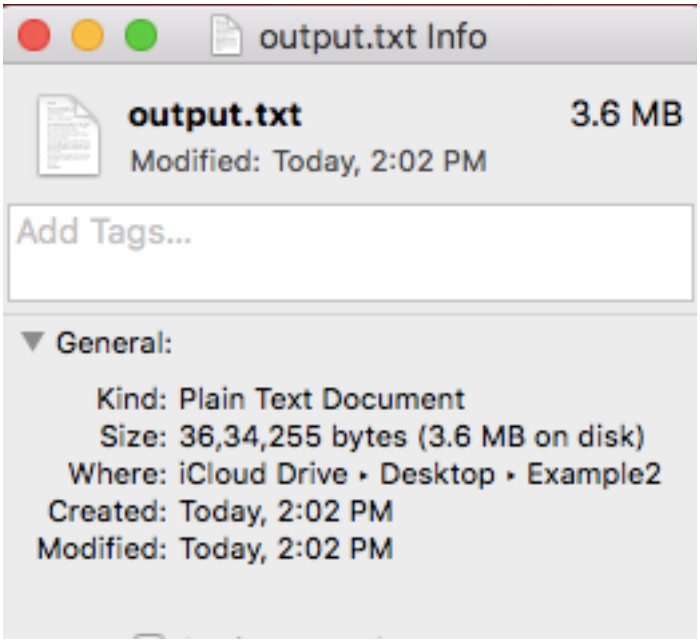


Generated Spectrogram

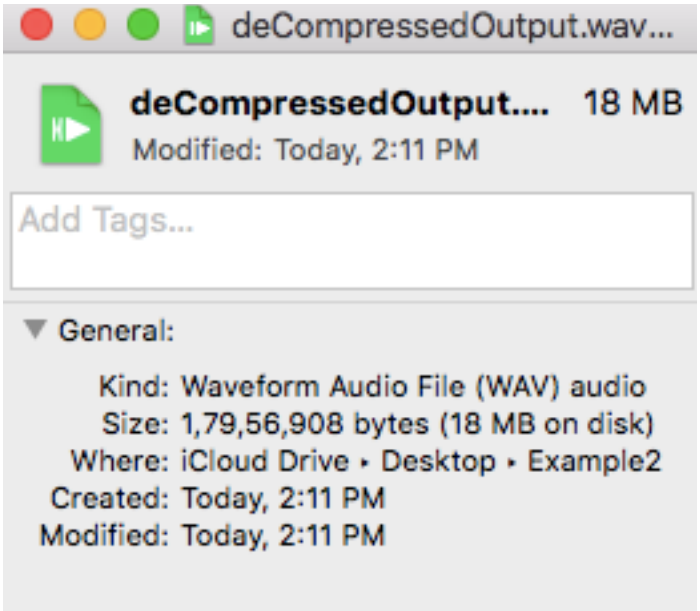




Encoded Text Size

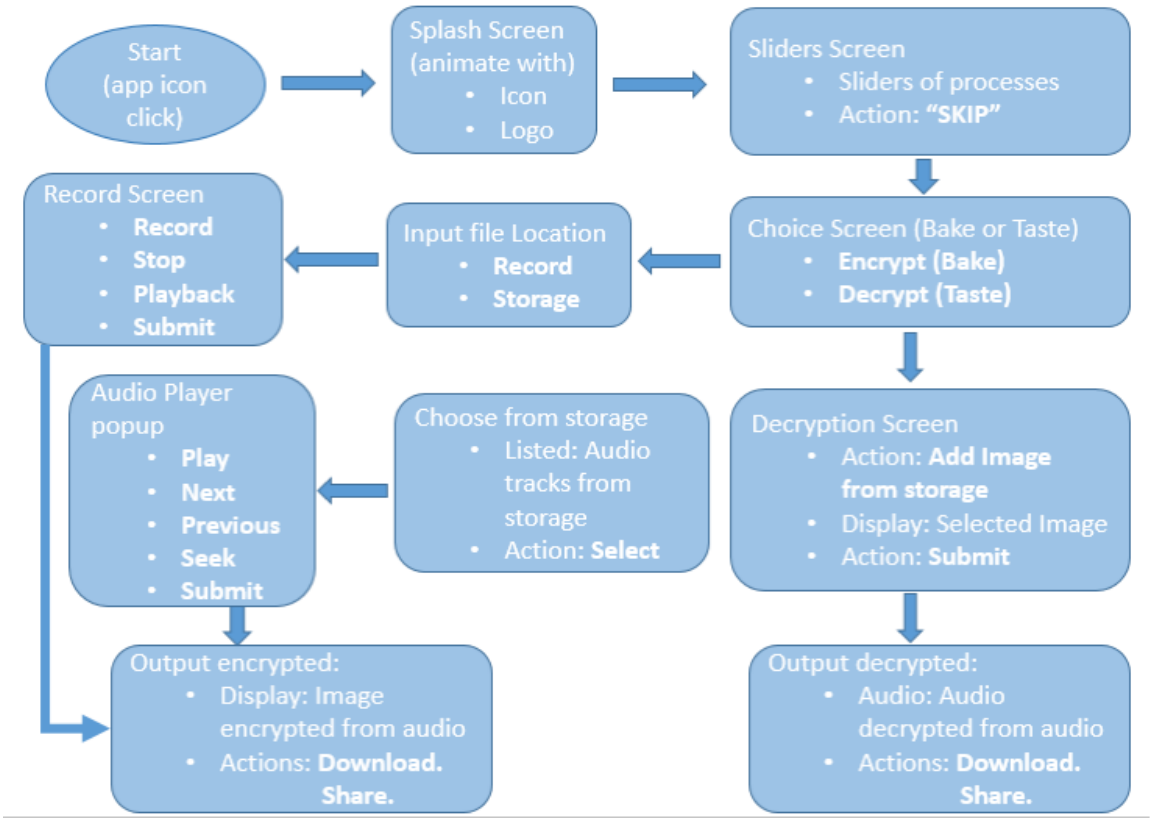


Uncompressed Audio File Size

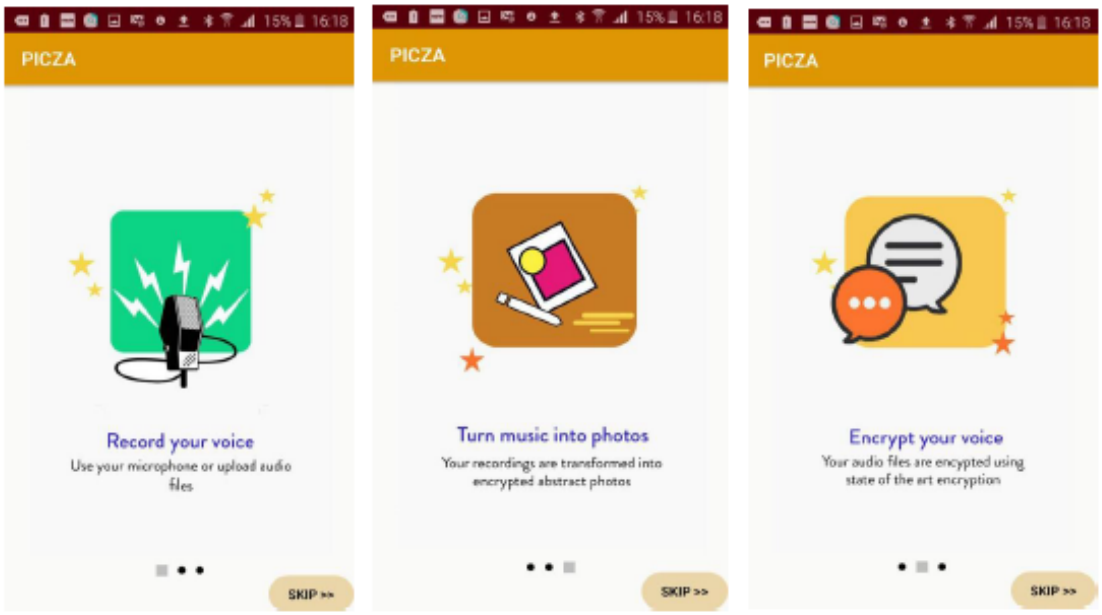


Potential Future Developments

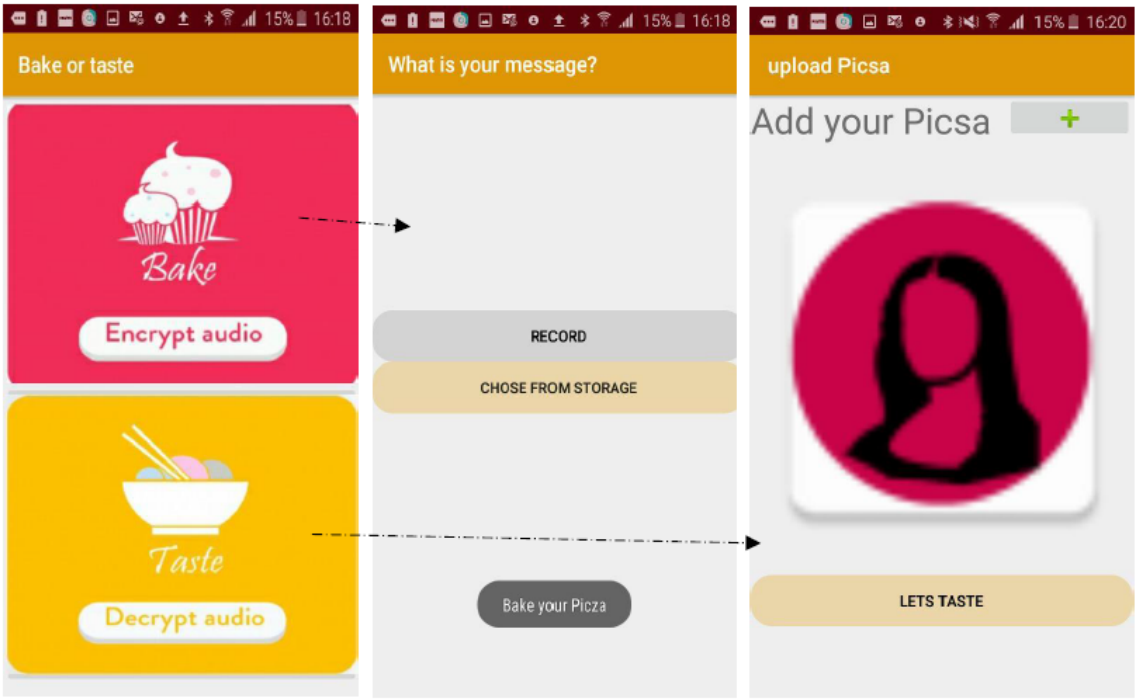
1. Now that we've completed the console application, the next logical step seems like having an application for the same reason. We've been since working on an android application of which the front-end is completed. As of now, we are implementing the back-end and fixing a few edge cases before publishing the same.



These sliders inform the users about exact processes a user needs to follow for application. The Sliders are currently kept active for each opening due to need for process reinforcement



Upon "SKIP", "Choice" activity comes up, from which we have to choose Encrypt/ Decrypt



Probable Applications

1.Practising industrial grade compression for fast data transfer.

2.To provide data encryption.

3.To prevent lossy compression via third party applications.

4.To implement PPM compression for the use of average end-user.

Evolution of the project

Over time as we researched further into the project area and gleaned new facts and better methodology, our approach changed too.

These involved:

- Substituting compression and encryption into a single module via Arithmetic Encryption.
- Using text files as the transferrable media instead of images as they are unlikely to be lossily compressed over applications.
- Shifting to generalized compression techniques instead of aiming to only reduce redundancy in specific cases.

Bibliography

1. The works of Patrick Feaster who has dealt with elements of paleomykophony for a decade, especially spectrographic treating
2. Analysis and Resynthesis Sound Spectrograph that once again deals solely in spectrographic manipulation
3. GNU Octave, a powerful language with built in graphical waveform visualization options
4. MIT OCW for Fast Fourier Transformation
5. Matt Mahoney's Data Compression Explained ebook
5. https://www.gnu.org/software/octave/
6. https://developer.android.com/training/index.html
7. https://nodejs.org/en/docs/
8. COMPARISON OF LOSSLESS DATA COMPRESSION ALGORITHMS FOR TEXT DATA (S.R. Kodituwakku et. al. / Indian Journal of Computer Science and Engineering Vol 1 No 4 416-425)
9. Research Paper on Text Data Compression Algorithm using Hybrid Approach (JCSMC,Vol. 3,Issue.12,December2014,pg.01–10)