# ACOUSTIC BEACONING

Tang Yu Han Brandon, Reiden Ong, Tan Yong Xin[1]

[1]Temasek Junior College, 22 Bedok South Road, 469278

## ABSTRACT

This project compares the effects of different parameters and techniques (i.e. the parameters of Baud rate, start sequence length, frequency used with MP3 compression and bit error threshold.) on the accuracy of acoustic beaconing using binary frequency shift keying. These parameters include the baud rate of data transmitted, the length of a start sequence used to identify the start of the data as well as the frequency of carrier signals used. Testing with lossy compression was also carried out. To accomplish this, we simulated different encoding and decoding methods in a Python script, testing the robustness of the process with simulated white noise of varying powers, measured using power ratio which is the ratio of the Root-Mean-Square of the signal over the Root-Mean-Square of the noise signal. Key results are that MP3 compression does not significantly affect the waveform such that the frequencies cannot be detected, accuracy of decoding increases with decreasing baud rate, the length of the start sequence is proportional to the accuracy of the decoding and proportional to packet loss rate.

## INTRODUCTION

Acoustic beaconing refers to the transmission of data through sound. Data (in this case binary data) is encoded into a sound wave, which is then transmitted by a speaker, received by a microphone, and then decoded again to the get the original data.

With the widespread use of microphone-equipped devices, there is many potential applications of acoustic beaconing for data communication. It can be used for both antagonistic and collaborative ways. For example, in a collaborative use scenario, it can be used to selectively transmit information to visitors of a museum depending on the exhibit they are looking at. This can be done by installing a transmitter at each exhibit to play a sound signal and having a microphone receiver on a hand-held device (which the visitor holds), the device will then decode the message from the encoded signal and display it on the screen. However, this can also be used for nefarious purposes such as secretly taking information from a mobile phone. Information can be exploited without a person's knowledge by transmitting the sound wave at an extremely high or low frequency (beyond the range of frequencies audible to humans). This can also be used to secretly track the location of people by using an application on a mobile phone or tablet that records sound. This could be used in behavioural analysis, allowing shop owners to track how often customers visit a specific aisle. It may also provide more evidence for crime scene investigations, providing information of the locations of suspects which may act as a crucial alibi.

\There are multiple encoding options available for this, including frequency shift keying, phase shift keying and amplitude shift keying.

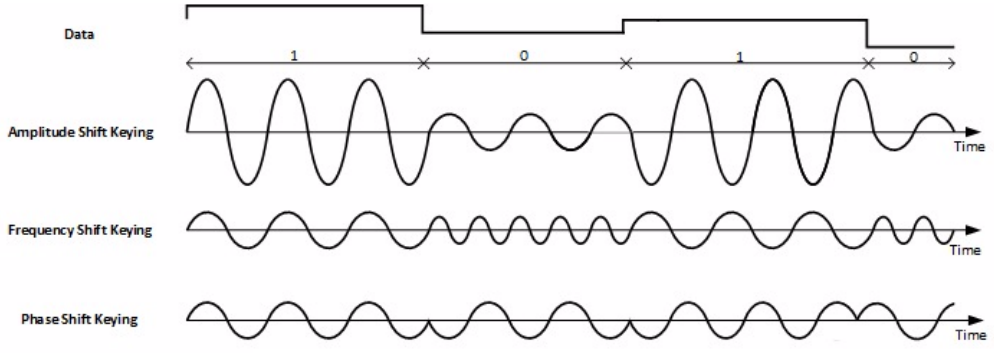



*Figure 1: Diagram of different encoding methods*

Amplitude shift keying is a form of amplitude modulation which represents data as variations of amplitude. This makes it easily disrupted by background noise level, causing the encoded message to have errors.

Phase shift keying transmits data by modulating the phase of the reference signal by varying the sine and cosine inputs and stipulated points of time. This severely limits the number of signals we can use and has a more complicated decoding process. Thus, as both amplitude shift keying and phase shift keying have many drawbacks and are not as suited for implementation in our research project, leading us to select frequency shift keying as our encoding method.

Frequency shift keying is a form of frequency modulation. It works by representing different symbols with different frequencies in a carrier signal. The carrier signal can then be sampled for its frequency at different times to obtain the original message sent. The benefits of frequency shift keying are that it can be adapted for a wide variety of scenarios, such as changing the frequencies so that they do not clash with other noises of the same frequency, as well as being able to have multiple frequencies for expansion purposes (E.G. Quadrature Frequency shift keying, which uses 4 frequencies).

## MATERIALS AND METHOD

Software required

- Python 3.6+
- Python libraries: SciPy, matplotlib
- Audio converter: FFmpeg
- Python script (attached to appendix)

### Overview of flow of python script code

For each set of parameters / optimisations (i.e. baud rate, frequency, bit error threshold and start sequence length) we choose to use for encoding and decoding, our code sends 50 packets of data. These parameters included the baud rate, signal carrier frequencies, sampling rate, etc. Each packet consists of a start sequence, meant to identify the starting point of the packet, as well as a message, encoded in bits. A packet is considered “received” when our program is able to identify the start sequence associated with the packet. The bit error rate of the trial is determined by the arithmetic mean of the bit error rates of packets which are received, achieved via bitwise comparison of the original bit message and the decoded bit message.

This would allow for easy comparison of the effects of the optimisations.

### Generating binary data

The data sent and transmitted was a 30-bit long binary bit message, which was generated using a seeded pseudorandom number generator in from a python library. Across various packets set within each bit level, the bit message was varied by changing the random seed, ensuring that the results could be accurately replicated with different bit messages. However, across different trials such as different noise power levels, the original seed remains to ensure fair and accurate testing.

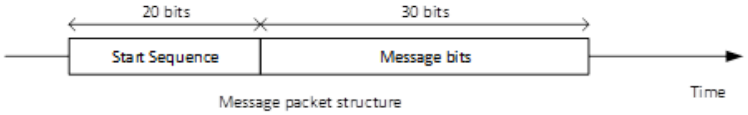



*Figure 2: Diagram of packet structure*

To be able to identify the start of each packet, a start sequence is used. The start sequence is a binary message of adjustable length in a difficult to replicate pattern, which is added on to the start of the bit message to allow for the recognition of the bit message in a soundwave.

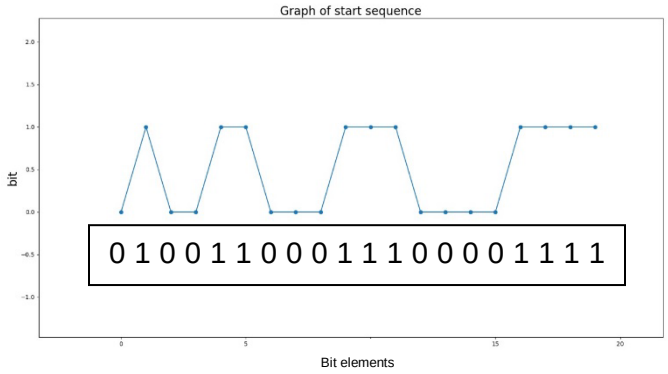



*Figure 3: Graph of an example of the start sequence*

01001100011100001111

The start sequence consists of a specific pattern of 0 and 1 bits. Starting with a 1 bit, an increasing number of 0 or 1 bits are added subsequently for an increasing segment of the code. E.g. 10, to 101100, to 101100111000. This particular pattern was used as it is non-translatable, with increasing effectiveness with increasing length, thus distinguishing it from the bit message by reducing the probability of it clashing with the randomly generated bit message.

The sending of multiple packets instead of an individual long string of bits, increase robustness in asynchronous communication. We test for packet loss, in terms of the percentage of start sequences identified (as being a start sequence) out of the total number of packets sent.

**Encoding data**

As our code uses multiple carrier frequencies, we used orthogonal frequency digital multiplexing (OFDM). Using this method, the frequencies for the carriers are selected such that they are orthogonal to one another, means that there is no intersymbol interference (ISI) between the 2 different frequencies used. As such, this allows us to perform the fourier transform accurately without the use of an apodization window as each frequency does not contribute to the amplitude when measuring the other frequency. To achieve orthogonality, the difference in frequencies has to be an integer multiple of the baud rate.

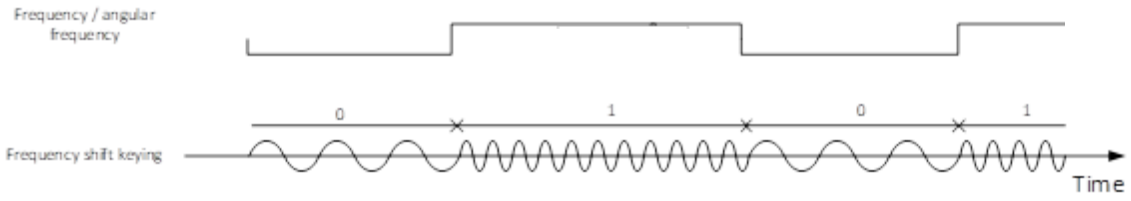



*Figure 4: Diagram of our original signal*

Parameters to consider:

1. Sampling rate (number of samples in signal per second)
2. Baud rate (the number of symbols per second)
3. Frequencies used to represent 1 and 0

For all out testing, we used the same sampling rate of 44100 Hz. Although this is computationally intensive and would increase the time needed to run the code, it allowed us to get a large number of samples per second, increasing the accuracy of our experimentation. By the Nyquist-Shannon sampling theorem, the sampling rate must be greater than twice the maximum frequency of the signal to have enough information to deduce the amplitude and frequency of the wave, thus, choosing this particular sampling rate also allows for our frequencies to range up to 20000 Hz, allowing us wider range of frequencies to experiment with.

To encode the data as a sound wave, we combined carrier sine waves of 2 different frequencies (one for each symbol), each with the time of baud rate seconds, while keeping the phase of the wave continuous.

$$\omega(t) = \frac{d\phi(t)}{dt} \qquad \phi(t) = \int_0^x \omega(t)\, dt = \sum_{t=0}^{T \times sr} \omega(t)\, dt$$

$$\textit{where } T \textit{ is the time in question ,}$$
$$sr \textit{ is the sampling rate,}$$
$$\omega \textit{ is the angular frequency of the wave}$$

Since the angular frequency of the wave is the rate of change of the phase of the wave, the definite integral from 0 to the time in question of the angular frequency of the wave would evaluate to the phase of the wave. For our research purposes, we used discrete data with the angular frequency of the waveform being a step function.

Thus, a Riemann sum of the angular frequency of the wave would result in an exact phase. This results in a soundwave with different frequencies which code for different symbols (1 and 0) in a continuous phase. The waveform is exported as a .wav file using SciPy, this file format is uncompressed and lossless and thus maintains complete accuracy of the data.

To test the effect of lossy audio compression, we converted the generated wav file into an mp3 format (lossy compression) and back into a wav format using FFmpeg, at a bit rate of 64 kbps (default). When converting the .wav file to a .mp3 file, the original signal file is compressed, the accuracy of specific components of sound which are not within hearing capabilities of most people are reduced through approximation. Furthermore, frequencies masked by stronger tones will also be removed, following the psychoacoustic model. This will cause some amount of data to be lost.

White noise is generated using a seeded pseudorandom number generator which generates white noise according to a normal distribution. This noise has a mean of 0 and a standard deviation of 1. The noise wave is then multiplied by an exponential factor for the different powers of white noise and added to the wave. The resultant soundwave is then normalized to a sound wave to have a root-mean-square value equal to the original wave.

**Decoding data**

The data received comes in the form of a .wav soundwave, which is read through a python library function into an array of amplitude of the wave against time. The wave is then repeatedly processed using the Fourier transform[1] to obtain the amplitude of $\boldsymbol{f_1}$ and $\boldsymbol{f_0}$ of that interval of the wave, resulting in streams of values of $\boldsymbol{f_0}$ and $\boldsymbol{f_1}$, as shown in Figures 5 and 6.

[1] A Fourier transform decomposes a signal into the frequencies that it consists of, and it represents the amplitude of that frequency present in the original function. Thus, using Fourier transforms can allow us to calculate the proportions of different frequencies in the signal and deduce the bit present at a specific time.

$$G(f) = \int_u^v g(t) \times e^{-2\pi i f t}\, dt = \sum_{t=u}^{v} g(t) \times e^{-2\pi i f t}\, dt$$

$$where\ [u, v]\ is\ the\ interval\ for\ the\ transform$$
$$g(t)\ is\ the\ original\ signal,$$
$$f\ is\ the\ frequency\ being\ analysed$$

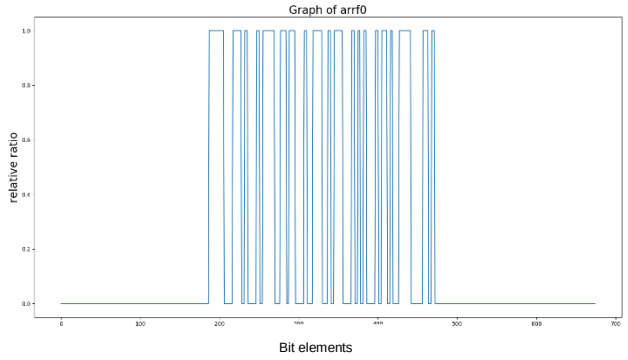



*Figure 5: Graph of array of f0*

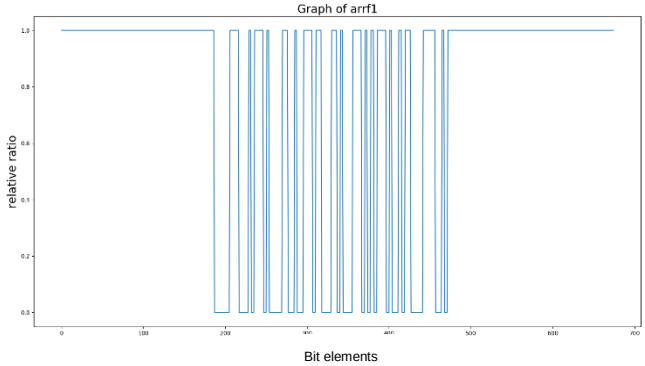



*Figure 6: Graph of array of f1*

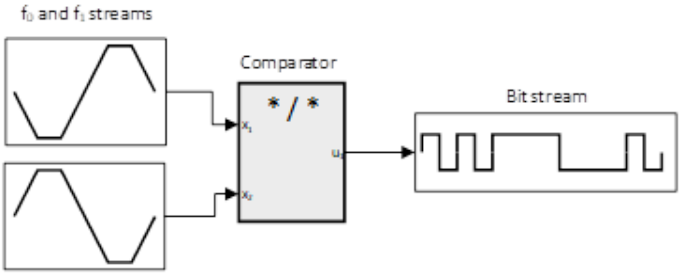



*Figure 7: Visual representation of how our results are compared*

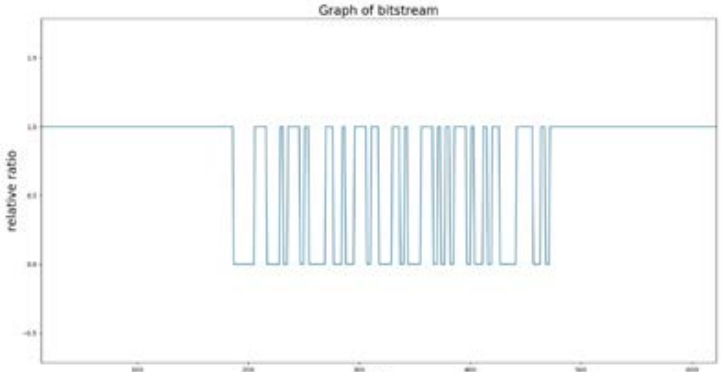



*Figure 8: Graph of bitstream*

These results are compared, with the optional optimization of a hysteresis which we will not explore in this project. This results in a "stream" of bits collected through the repeated sampling (which we call the bitstream).

The bitstream is then analysed by sampling at fixed intervals with a sliding window to check for the start sequence. A popularity voting mechanism is implemented here.

This works by calculating the arithmetic mean of the bit stream bits across the time for a bit message bit. If the mean is greater than 0.5, it would mean that majority of the stream bits vote that the message bit codes for a 1, thus the program treats that sequence as a 1 in the decoded bit message, and vice versa for a mean less than 0.5.

We did not require the detected start sequence to exactly match the generated start sequence, but rather has a variable bit error threshold of 20% which allows room for error in the detection of the start sequence. This is such that only 80% of the start sequence has to match,

increasing the margin of error and allowing a few bits to be lost when decoding, without preventing the start sequence from being detected.

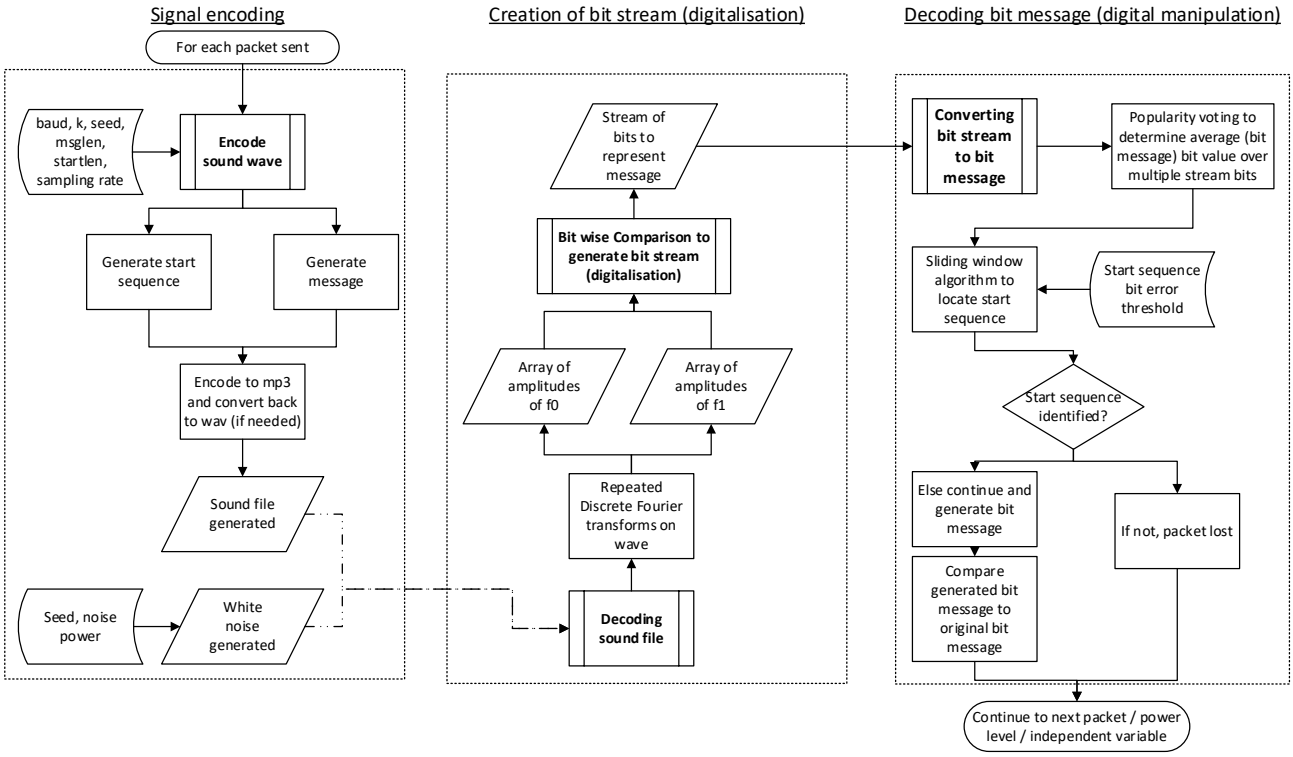
Once the start sequence is identified, repeated sampling continues for the next message length of message bits.

## Calculating and displaying results

For each packet received where the start sequence is identified, the accuracy of the decoded bit message is determined through the bit error rate, from a bit error function. The derived bit message is compared bitwise with the original bit message, and is terminated at the end of the message, and expresses the bit error rate as the percentage of wrong bits over the total bits.

A graph is then plotted of the average bit error rate across all the packets for each power level against the logarithmic scale of loud noise in decibels.

### Flowchart of program for each packet tested




## RESULTS & DISCUSSION

## Baud results (Fig 9, Fig 10)

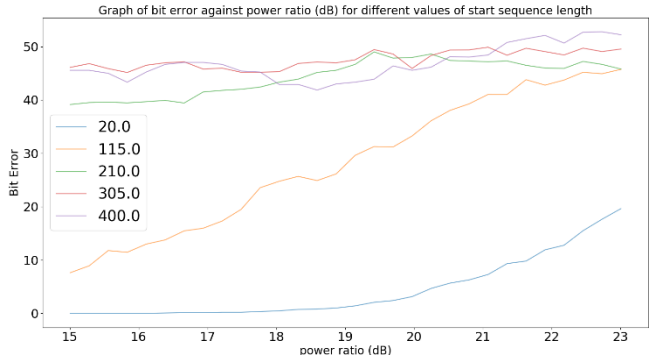



*Figure 9: Graph of bit error rate against power ratio for different values of baud rate*

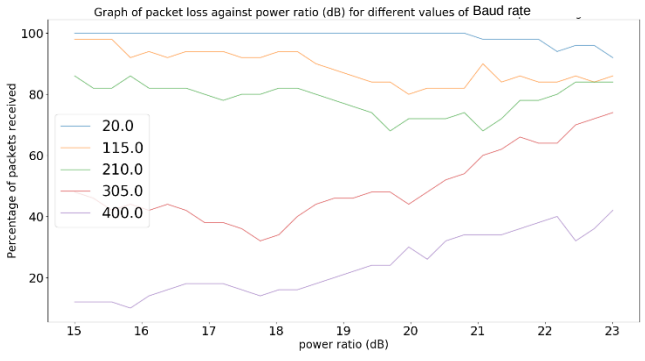



*Figure 10: Graph of packet loss against power ratio for different values of baud rate*

In terms of the value of Baud rate, data shows (Fig 9, Fig 10) that a lower value would result in more accurate decoding as the popularity vote used in the decoding could help with ensuring that most of the message would be decoded accurately. Furthermore, a lower baud rate greatly helps to ensure that the start sequence is decoded correctly. This is especially true is a high bit error threshold (greater than 5) is used since wrongly decoded start sequences can easily be considered as being correct, hence robust decoding of the start sequence is required.

However, a lower baud rate would also result in a longer time required to send the message and would make decoding the message more computationally intensive due to the increased number of operations (particularly fourier transforms) needed. Furthermore, in the case of intermittent pulses of interference, a low baud rate results in lesser accuracy whereas a signal of high baud rate could be used to transfer data in the time between the pulses. Thus, when choosing a value of baud, one would have to consider the practical usage of their code and find a middle ground between speed and computational complexity and accuracy of results as well as considering any special types or patterns of interference.

**Start sequence length results (Fig11, Fig 12)**

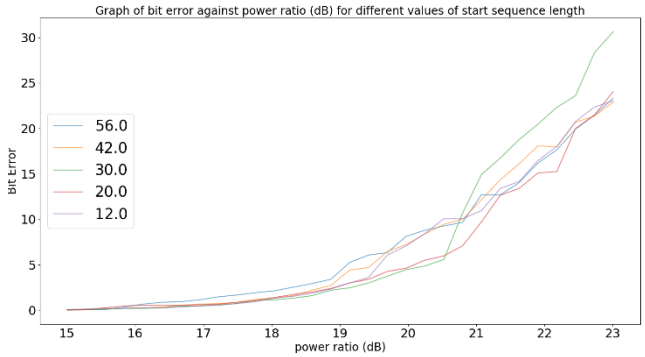



*Figure 11: Graph of bit error rate against power ratio for different values of start sequence length*

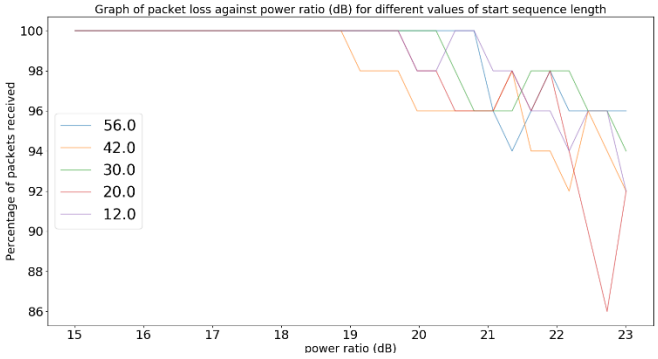



*Figure 12: Graph of packet loss against power ratio for different values of start sequence length*

In terms of the start sequence length, a longer start sequence length results in more accurate decoding, as the longer start sequence is more unique and harder to be replicated in other parts of the bit message, and the exact start location is more accurate, thus the information detected is more of the information of the bit message, and less of random bits created by the white noise interference. However, this may be required in the case where the data received in a packet must be very accurate.

However, it can be seen that the higher start sequence length also results in greater packet loss rates. This is expected and can be explained through how since the start sequence is longer, there is a greater probability of having a wrong bit, which leads to a greater probability of not fulfilling the bit error threshold and thus not recognising the start sequence.

Thus, various start sequence lengths would be ideal for different situations. Longer start sequences could be ideal in scenarios where the accuracy of the information received is of utmost importance, and that packet loss can be neglected in favour of more accurate information. Shorter start sequences would thus be ideal for cases where packet loss has to be kept to a minimum, and the accuracy of the data is not strictly necessary. Thus, to choose an optimal value of baud, the accuracy against the packet loss of the data must be considered per the specific needs of the acoustic beacon.

## MP3 compression with Frequency results (Fig13, 14, 17, 18)

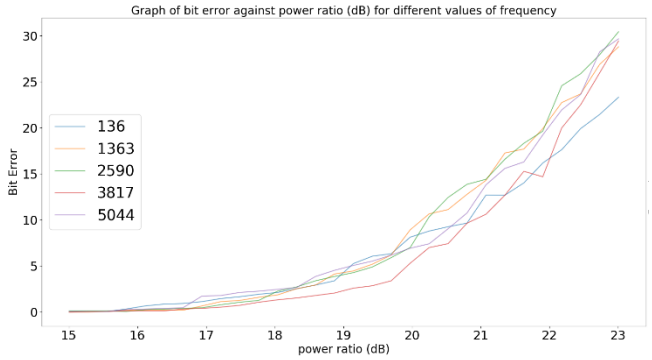



*Figure 13: Graph of bit error rate against power ratio for different values of frequencies (ranging from 136-5044)*

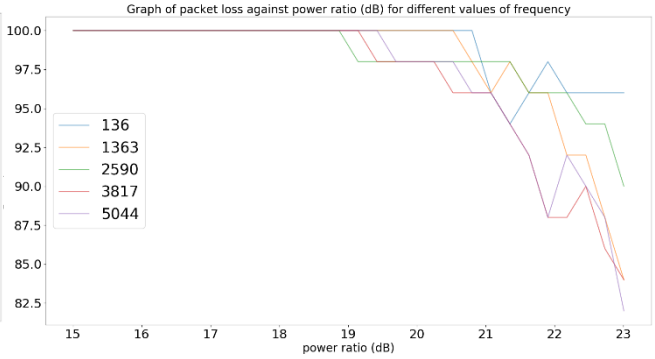



*Figure 14: Graph of packet loss against power ratio for different values of frequencies (ranging from 136-5044)*

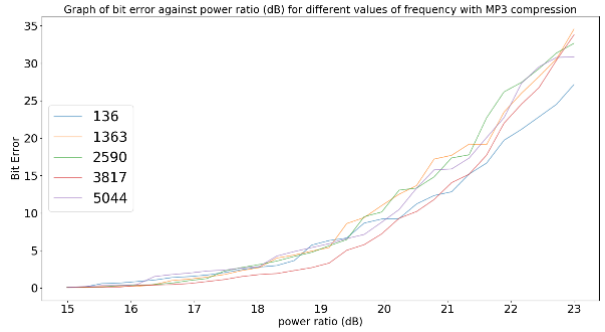



*Figure 17: Graph of bit error rate against power ratio for different values of frequencies (ranging from 136-5044) with MP3 compression*

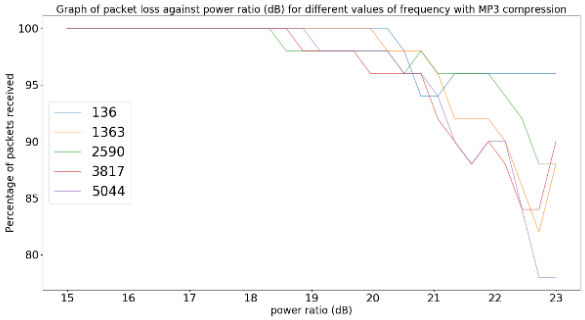



*Figure 18: Graph of packet loss against power ratio for different values of frequencies (ranging from 136-5044) with MP3 compression*

The hypothesis was that frequencies masked by stronger tones will also be removed, following the psychoacoustic model, where tones at frequencies outside of the audible range of humans are removed to save space.

This would cause some amount of data to be lost, especially for extremely high and low frequencies, removing the data encoded in these high frequencies. However, this data lost is not significant, as seen in the similarities between Fig 17 and 18 with Fig 13 and 14 respectively, where MP3 compression only results in a slightly steeper curve which ends at a bit error rate of 35%, as compared with the normal 30% without MP3 compression.

Thus, MP3 compression does not affect much of the frequency and clarity of the signal, such that even when passed through MP3 encoding, the waveform is able to retain enough of the original shape such that data can still be accurately decoded.

## Bit error threshold results (Fig 18 and Fig 19)

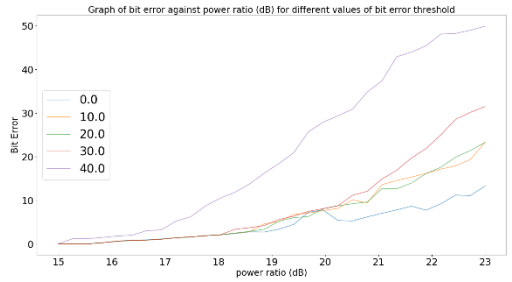



*Figure 18: Graph of bit error rate against bit error threshold for start sequence detection (ranging from 0 to 40%)*

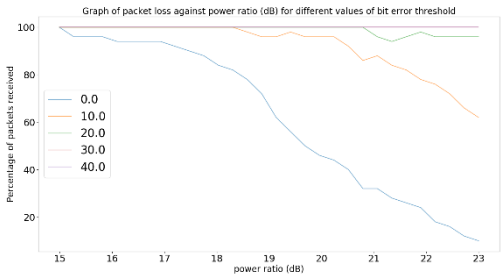



*Figure 19: Graph of packet loss against bit error threshold for start sequence detection (ranging from 0 to 40%)*

In terms of bit error rate, a lower bit error threshold would provide lower bit error rates with greater accuracy. This is due to the fact that for lower bit error threshold rates, the start sequence is detected with greater accuracy as there is less room for error in detecting the start sequence, thus it is more unlikely for the resultant bit stream to be translated left or right, or being detected at a wrong location.

In terms of packet loss rates, a greater bit error threshold would result in greater packet loss rates. This is because greater bit error thresholds allow for more error in the detection of the start sequence, such that even if a few bits are off it can be written off. Especially for the results of bit error threshold of 30 and 40, as 30 and 40 are close to the random bit error rates, it has a great chance of detecting nearly anything as a start sequence.

In choosing a value of bit error threshold, one has to find balance between the bit error rates and packet loss, as lower bit error threshold results in greater accuracy in message decoding, but a lower chance of receiving the packet

## Conclusion

The results of the project have been in-line with current scientific knowledge, and possess many applications, particularly in the creation of start sequences and choosing baud rates, as in various scenarios those can be varied to different effects (E.G. having a low value of baud to ensure accuracy of data when time is not an issue). Possible follow-up actions could be to further examine the results of more extensive MP3 compressions, as well as detection of sound in real world scenarios (E.G. detection of handphones).


## ACKNOWLEDGEMENTS

We would like to thank our mentor, Dr Hofbauer Wulf for his guidance and support throughout the whole project.

We would also like to thank our school and the mentors, Mr Shawn Neo and Mrs Chan Boon Hwee for giving us this opportunity and their support during the research project

## REFERENCES

"FSK - Frequency Shift Keying." André-Marie Ampère - T&M Atlantic. Accessed August 01, 2018. http://www.tmatlantic.com/encyclopedia/index.php?ELEMENT_ID=10422.

"PSK - Phase Shift Keying." André-Marie Ampère - T&M Atlantic. Accessed August 01, 2018. .http://www.tmatlantic.com/encyclopedia/index.php?ELEMENT_ID=10478

"ASK - Amplitude Shift Keying." André-Marie Ampère - T&M Atlantic. Accessed August 01, 2018. .http://www.tmatlantic.com/encyclopedia/index.php?ELEMENT_ID=10420

## ANNEX

*Table 1: List of variables in our code*

| Variable | Symbol | Default value | Sampled range |
|---|---|---|---|
| Standard Variables | | | |
| Sampling rate | sr | 44100 | |
| Baud rate (signals per second) | baud | 45.45 | 20-400 |
| Frequencies of logic 1 and logic 0 | $f_0$ $f_1$ | baud*3 baud*4 | |
| Difference in frequencies bin | df | baud | 3-111 |
| Bit message length | msglen | 30 | - |
| Bit message random seed | original_seed_message | 10 | - |
| Length of start sequence increment | start_len | 20 | 20 – 56 |
| Noise Variables | | | |
| Minimum Noise power | noise_power_start | 10^1.2 | - |
| Maximum Noise power | noise_power_end | 10^2.0 | - |
| Total Noise samples | total_samples | 30 | - |
| Digitisation Variables | | | |

| | | | |
|---|---|---|---|
| Fourier transform window length | ftlength | 1000 | - |
| Squelch minimum noise threshold | threshold | 0 (not in use) | - |
| Hysteresis center | hysteresis | 0 (not in use) | - |
| Bit error threshold for detecting start sequence | biterr_threshold | 20% | 0% to 40% |

**Main code**

```
import scipy
import scipy.signal
import scipy.io.wavfile
import random
import time
import matplotlib.pyplot as plt
import csv
import matplotlib
matplotlib.rcParams.update({'font.size': 22})

func = __import__("Decoding FSK 7 mp3 Functions")                    #import functions as func.


def printProgressBar (iteration, total, prefix = '', suffix = '', decimals = 1, length = 100, fill = '█'):
    """
    Call in a loop to create terminal progress bar
    @params:
        iteration   - Required  : current iteration (Int)
        total       - Required  : total iterations (Int)
        prefix      - Optional  : prefix string (Str)
        suffix      - Optional  : suffix string (Str)
        decimals    - Optional  : positive number of decimals in percent complete (Int)
        length      - Optional  : character length of bar (Int)
        fill        - Optional  : bar fill character (Str)
    """
    percent = ("{0:." + str(decimals) + "f}").format(100 * (iteration / float(total)))
    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
    print('\r%s |%s| %s%% %s' % (prefix, bar, percent, suffix), end = '\r')
    # Print New Line on Complete
    if iteration == total:
        print()
```

```
resuming = True
mp3 = True
#basic variables
sr = 44100
baud = 45.45                                    #symbols per second
OFDM_multiplier = 3
f0 = baud*OFDM_multiplier                       #Frequency of 0 bit
f1 = baud*(OFDM_multiplier+1)                   #Frequency of 1 bit
msglen = 30                                     #number of bits in message to receive\
original_seed_msg = 10
start_len  =4
#Experiment config variables
noise_power_start = 1.2                         #10**noise_power is the power of white
noised used
noise_power_end = 2.0                           #0.5 to 0.9
total_samples = 30                              #total number of samples to do

#Digitalisation Variables
ftlength = 1000                                 #samples in signal per ft
threshold = 0                                   #for squelch --> lower == quieter signals can
go through
hysteresis = 0                                  #(0, 0.5) larger --> greater difference from 0.5
(for bit_ratio) needed for hysteresis

#Stream to bitmsg variables
biterr_threshold = 20                           #percentage similarity to start sequence
needed to count as identified
pop_width = 100                                 #pop_vote over the center pop_width
percent of stream bits for every message bit
pop_step = 1                                    #pop_vote sum --> for index in range (start,
end, step)

#Selecting independant variable for analysis
var_name = "frequency"
lowest_iv =  3                                  #lowest value for iv
highest_iv = 111                                #highest value for iv
total_iv = 5                                    #total number of ivs (between lowest and
highest) to test
setiv = scipy.linspace(lowest_iv, highest_iv, total_iv)          #set of iv values to test
total_packet = 50


#Initial variables
iv_counter = 0
spectrogram_biterr = []
```

```
spectrogram_bitsim = []
spectrogram_packet_received = []
points = scipy.linspace(noise_power_start,noise_power_end,total_samples)
point_axis = scipy.zeros(0)
rms = 0
original_signal = []
total_increment = total_samples*total_iv*total_packet
current_increment = 0

#Main program
time_start = time.time()                                    #Start time variable
print("-------------------------------------------------")
print("Independant Variable being tested:", var_name)
print(lowest_iv, "<", var_name, "<", highest_iv)
print("-------------------------------------------------\n")

def resume():
    with open('IV file saver.csv') as csvfile:
        csv_reader = csv.reader(csvfile, delimiter='|')
        counter = 0
        for row in csv_reader:
            if counter == 0:
                finished_ivs = int(''.join(row))#how many ivs done
            elif counter==1:
                spectrogram_biterr = eval(row[0])
            elif counter==2:
                spectrogram_bitsim = eval(row[0])
            elif counter==3:
                spectrogram_packet_received = eval(row[0])
            counter+=1
        print ("RESUMINGGG")
        if finished_ivs != total_iv:
            print("resuming from", finished_ivs+1)
        else:
            print ("ready to display results")

        return finished_ivs,spectrogram_biterr,spectrogram_bitsim,spectrogram_packet_received

if (resuming):
    finished_ivs,spectrogram_biterr,spectrogram_bitsim,spectrogram_packet_received = resume()
for iv in setiv:
    '"PLEASE SET THE INDEPENDANT VARIABLE HERE"'

    f0 = baud*iv                                    #Frequency of 0 bit
    f1 = baud*(iv+1)
```

```
    iv_counter+= 1
    if resuming:
        if finished_ivs == total_iv:
            break
        elif (iv_counter<= finished_ivs):
            current_increment += total_packet*total_samples
            continue

    #resetting variables
    biterr_arr = []
    bitsim_arr = []
    packet_received_arr = []
    counter = 0
    for x in points:
        seed_msg = original_seed_msg
        counter += 1
        packet_received = 0 #number of packets received for this point
        biterr_sum = 0
        bitsim_sum = 0
        print("Power:", str(counter)+'/'+str(total_samples))
        print ("Independant Variable Value:", iv)
        print("Independant Variable Val", iv_counter, "/", total_iv)
        for packet_no in range(total_packet):
            print ("packet:", packet_no+1)
            seed_msg += 1
            #Creation of sound file
            original_signal = func.encode(sr, baud, f1, f0, msglen, start_len, seed_msg)
            #Creation of start codon
            start  = []
            for q in range(1,start_len):
                start = start + q*[0]
                start = start + q*[1]

            #MP3 and Importing file

            if mp3:
                func.mp3_encode()
                sr, original_signal = scipy.io.wavfile.read("sample.wav")
                original_signal = original_signal.astype(scipy.float64)
                original_signal = scipy.asarray([x/32767 for x in original_signal])


            dt = 1/sr

            original_signal_rms = (func.rms(original_signal))**2                #getting the average
power of the original signal
```

```
        rms = original_signal_rms
        signal = scipy.zeros(len(original_signal))

        #white noise generator
        scipy.random.seed(seed_msg)

        noise_signal = scipy.random.normal(0,1,size = (sr*20))
        noise_power_amplifier = 10**x
        for i in range (len(original_signal)):
           signal[i] = (noise_power_amplifier**0.5) *noise_signal[i] + original_signal[i]

        #Creating original_bitmsg
        original_bitmsg = scipy.zeros(0)                    #array for original bit message
        random.seed(seed_msg)                               #seed for generating random
sequence
        for i in range(msglen):
           original_bitmsg = scipy.append(original_bitmsg,random.randint(0,1))

        #Calculations
        power_ratio = original_signal_rms/(func.rms(signal)**2)        #power ratio
decreases with noisier signal
        amplitude_ratio = power_ratio**0.5                  #power proportional to
amplitude squared
        signal = signal*amplitude_ratio

        #Analysis of signal----------
        #print('fourier start')
        arrf0 = abs(func.fourier(signal,f0,sr,ftlength,25))
        arrf1 = abs(func.fourier(signal,f1,sr,ftlength,25))
        print('fourier')

        #Cleaning Stream-------------------------
        stream, bitratioarray = func.create_stream(arrf0, arrf1, f0, f1, threshold, hysteresis)
        #tream = func.streamclean(stream,cleanwindow)
        t = scipy.linspace((ftlength/ (2*sr)), (signal.size/sr ), len(stream))

        #Getting Bitmsg From Stream--------------
        stream_dt = dt*len(signal)/len(stream)
        bitmsg = func.get_bitmsg(stream, baud, start, stream_dt, msglen, biterr_threshold,
pop_width, pop_step)
        if bitmsg != 2:
           #Accuracy Analysis----------
           biterr_rate = func.bit_error_rate(original_bitmsg,bitmsg)
           bit_similarity = func.string_similarity(bitmsg,original_bitmsg)
           #Recording of results-------------------
           biterr_sum += biterr_rate
```

```
            bitsim_sum += bit_similarity
            packet_received += 1

            #print (str(bit_similarity)+ '% bit similarity')
            print (str(scipy.round_(biterr_rate, 1))+"% bit error rate")
         else:
            print ("Packet lost")
         #printProgressBar(current_increment, total_increment, prefix = 'Progress:', suffix =
'Complete', length = 50)
         current_increment+= 1

         #Showing results-----------------------

         print('\n')
      if (packet_received > 0):
         biterr_arr.append(biterr_sum/packet_received)
         bitsim_arr.append(bitsim_sum/packet_received)
      else:
         biterr_arr.append(100)
         bitsim_arr.append(0)
      packet_received_arr.append(100*packet_received/total_packet)
      print("POINT DONE\n")
      #TIME ESTIMATE
      time_end = time.time()
      #print("-----------------TIME TAKEN:", int((time_end-time_start)/60), "minutes and ",
int((time_end-time_start))%60, "Seconds")
      time_taken = time_end - time_start
      time_per_point = time_taken / ((iv_counter-1)*total_iv + counter)
      time_remaining = time_per_point*(total_iv - iv_counter)*(total_samples) + (total_samples -
counter)*time_per_point
      time_remaining = "TIME REMAINING: " + str(int((time_remaining)/60))+ " minutes and
"+ str(int(time_remaining)%60)+ " Seconds"
      printProgressBar(current_increment, total_increment, prefix = '', suffix = time_remaining,
length = 50)
      print ("\n")



   spectrogram_biterr.append(biterr_arr)
   spectrogram_bitsim.append(bitsim_arr)
   spectrogram_packet_received.append(packet_received_arr)
   with open('IV file saver.csv', 'w', newline='') as csvfile:
      spamwriter = csv.writer(csvfile, delimiter='|', quotechar=' ',
quoting=csv.QUOTE_MINIMAL)
      spamwriter.writerow([iv_counter])
      spamwriter.writerow([spectrogram_biterr])
```

```
        spamwriter.writerow([spectrogram_bitsim])
        spamwriter.writerow([spectrogram_packet_received])
    print ("Independant variable value finished")
    print ("--------------------------------------------------------------------------------")

time_end = time.time()
print("TIME TAKEN:", int((time_end-time_start)/60), "minutes and ", int((time_end-
time_start))%60, "Seconds")

#Colour Mesh (WIP)
def show_mesh_biterr():
    plt.pcolormesh(noise_power_amplifier_axis, setiv, spectrogram_biterr)
    plt.title("Spectrogram of bit error against noise amplifier for different IVs")
def show_mesh_bitsim():
    plt.pcolormesh(noise_power_amplifier_axis, setiv, spectrogram_bitsim)
    plt.title("Spectrogram of bit similarity against noise amplifier for different IVs")

#Multiple Line Graph (for displaying results)
original_signal = func.encode(sr, baud, f1, f0, msglen, start_len, original_seed_msg)
rms = func.rms(original_signal)
scipy.random.seed(original_seed_msg)
noise_signal = scipy.random.normal(0,1,size = (sr*20))
noise_signal = noise_signal[:len(original_signal)]
power_axis = [10*scipy.log10((10**x)*func.rms(noise_signal)**2/(rms**2)) for x in points]

def show_biterr():
    plt.cla()
    for ivcounter in range(total_iv):
        plt.plot(power_axis,spectrogram_biterr[ivcounter], label = str(setiv[ivcounter]), alpha = 0.6)
    legend = plt.legend(loc='center left', fontsize='large')
    plt.title("Graph of bit error against power ratio (dB) for different values of "+ var_name + "
with MP3 compression",  fontsize=20)
    plt.xlabel("power ratio (dB)", fontsize=20)
    plt.ylabel("Bit Error", fontsize=20)
    plt.show()
def show_bitsim():
    plt.cla()
    for iv_counter in range(total_iv):
        plt.plot(power_axis,spectrogram_bitsim[iv_counter], label = str(setiv[iv_counter]), alpha =
0.6)
    legend = plt.legend(loc='center left', fontsize='large')
    plt.title("Graph of bit similarity against power ratio (dB) for different values of " + var_name ,
fontsize=20)
    plt.xlabel("power ratio (dB)", fontsize=20)
    plt.ylabel("Bit Similarity", fontsize=20)
    plt.show()
```

```
def show_packet():
    plt.cla()
    for iv_counter in range(total_iv):
        plt.plot(power_axis,spectrogram_packet_received[iv_counter], label =
str(setiv[iv_counter]), alpha = 0.6)
    legend = plt.legend(loc='center left', fontsize='large')
    plt.title("Graph of packet loss against power ratio (dB) for different values of " + var_name + "
with MP3 compression", fontsize=20)
    plt.xlabel("power ratio (dB)", fontsize=20)
    plt.ylabel("Percentage of packets received" , fontsize=20)
    plt.show()


#debugging**********************************************************************
**********************************************************************************
**************************************************
def debug(): #list of debugging tools
    print ('debugging tools:' + '\n' + 'signal' + '\n' + 'f01' + '\n' + 'bitratio'+ '\n' + 'stream'+ '\n' +
'bitmsg'+'\n' + 'biterr'+'\n' + 'bitsim'+'\n' + 'multiplot (beta)')
    command = input()
    command = str(command)
    if command == 'signal': signal_debug()
    elif command == 'f01': f01_debug()
    elif command == 'bitratio': bitratio_debug()
    elif command == 'bitmsg': bitmsg_debug()
    elif command == 'stream': stream_debug()
    elif command == 'biterr': biterr_debug()
    elif command == 'bitsim': bitsim_debug()
    elif command == 'multiplot':
        print ('available tools for multiplot:' +'\n'+ 'bitratio'+ '\n' + 'stream'+ '\n' + 'bitmsg'+'\n' +
'biterr'+'\n' + 'bitsim')
        plt.figure(1)
        plt.subplot(211)
        command = input()
        command = str(command)
        if command == 'bitratio': bitratio_debug()
        elif command == 'bitmsg': bitmsg_debug()
        elif command == 'stream': stream_debug()
        elif command == 'biterr': biterr_debug()
        elif command == 'bitsim': bitsim_debug()
        plt.subplot(212)
        command = input()
        command = str(command)
        if command == 'f01':f01_debug()
        elif command == 'bitratio': bitratio_debug()
        elif command == 'bitmsg': bitmsg_debug()
```

```
        elif command == 'stream': stream_debug()
        elif command == 'biterr': biterr_debug()
        elif command == 'bitsim': bitsim_debug()

    plt.show()

def signal_debug():                                     #graph of soundwave
    plt.figure(1)
    plt.subplot(211)
    plt.plot(original_signal)
    plt.title('original signal')
    plt.subplot(212)
    plt.plot(signal)
    plt.title('read from wav signal')

def f01_debug():                                        #graph of frequency 1 and frequency 0
    plt.figure(1)
    plt.subplot(211)
    plt.plot(t, arrf1)
    plt.title('array of f1')
    plt.subplot(212)
    plt.plot(t, arrf0)
    plt.title('array of f0')

def bitratio_debug():                                   #graph of bit ratio
    plt.plot(t, bitratioarray)
    plt.title('bitratioarray')

def bitmsg_debug():                                     #graph of the message in bit form
    plt.figure(1)
    plt.subplot(211)
    plt.plot(bitmsg,'o-')
    plt.title('bit_message')
    plt.subplot(212)
    plt.plot(original_bitmsg, 'o-')
    plt.title('original bit message')

def stream_debug():                                     #graph of the bitstream
    plt.plot(t, stream,'o-')
    plt.title('stream')

def biterr_debug():
    plt.plot(point_axis, biterrorarr,'o-')
    plt.gca().invert_xaxis()
    plt.title("bit error rate for different rms ratios of white noise")
```

```
def bitsim_debug():
    plt.plot(point_axis,bitsimarr,'o-')
    plt.gca().invert_xaxis()
    plt.title("bit similarity rate for different rms ratios of white noise")

def stream_debug():                                    #graph of the bitstream
    plt.plot(stream,'o-')
    plt.title('stream')
    plt.show()
```

**Code for Modular Functions**

```
import scipy
import scipy.signal
from difflib import SequenceMatcher
import matplotlib.pyplot as plt
import random
import subprocess

#Functions
def rms(wave):
    return scipy.sqrt(scipy.mean(wave**2))

def fourier(signal, f, sr, windowlength, pos_increment):
    omega = 2 * scipy.pi * f
    expfactor = scipy.exp(-1j*omega*scipy.arange(windowlength)/sr)
    window = 0.5*(1+scipy.cos(scipy.linspace(-scipy.pi, scipy.pi, windowlength,
endpoint=False)))
    fftfactor = expfactor * window

    position = 0
    ft = []
    while len(signal) >= position+windowlength:
        fourieramplitude = sum(signal[position:position+windowlength] * fftfactor)
        ft.append(fourieramplitude)
        position += pos_increment
    ft = scipy.array(ft)
    return ft

def create_stream(arrf0, arrf1, f0, f1, threshold, hysteresis):
    bitratioarray = scipy.zeros(0)
    state = 1
    stream = scipy.zeros(0)
    for x in range (arrf0.size):                                    #for each time t
        bit_ratio = arrf1[x]/(arrf1[x]+ arrf0[x])
        bitratioarray = scipy.append(bitratioarray, bit_ratio)
        if arrf1[x]< threshold and arrf0[x] < threshold:                    #ignore if noise
```

```
            stream = scipy.append(stream, 1)
            #print('1')
            continue
        elif state ==1:
            state = (bit_ratio > (0.5 - hysteresis))                    #hysteresis
            stream = (scipy.append(stream, state))
            #print(state)
        elif state ==0:
            state = (bit_ratio > (0.5 + hysteresis))                    #hysteresis
            stream = (scipy.append(stream, state))
            #print(state)
    return stream, bitratioarray

def get_bitmsg(stream, baud, start_sequence, dt, msglen,biterr_threshold, pop_width, pop_step):
    bit_time = 1/baud
    search_container = []
    message_container = []
    message_started = False
    i = 0
    while(True):
        start = int(i/dt +0.5)
        end = int ((i+bit_time)/dt +0.5)
        bit_length = end - start                                    #elements per message bits
        if (end >= len(stream)):
            print ("FAILED to identify start sequence")
            return 2

        #Pop_Vote
        start+= int((100-pop_width)*bit_length/200 +0.5)                    #adjusting for
pop_width
        end -= int((100-pop_width)*bit_length/200 +0.5)
        average =0
        samples = 0
        for index in range(start, end, pop_step):
            average += stream[index]
            samples += 1
        average /= samples

        if (not message_started):
            if average >0.5:
                search_container.append(1)
            else:
                search_container.append(0)
        else:
            if average >0.5:
                message_container.append(1)
```

```
            else:
                message_container.append(0)

        if ((not message_started) and len(search_container) >= len(start_sequence)):
            if bit_error_rate(search_container[len(search_container) - len(start_sequence):],
start_sequence)<=biterr_threshold:
                print("Start Sequence Bit Error Rate: ",
scipy.round_(bit_error_rate(search_container[len(search_container) - len(start_sequence):],
start_sequence), 1))
                message_started = True
                i+= bit_time    #BS fix to move everything by 1 bit
        if len(message_container) == msglen:
            return message_container
        i += bit_time

def bit_error_rate(original_bitmsg, bitmsg):
    error = 0
    original_bitmsg_length = len(original_bitmsg)
    bitmsg_length = len(bitmsg)

    if original_bitmsg_length > bitmsg_length:
        original_bitmsg = original_bitmsg[:bitmsg_length]

    elif original_bitmsg_length < bitmsg_length:
        bitmsg = bitmsg[:original_bitmsg_length]

    error = scipy.mean(abs(scipy.subtract(original_bitmsg,bitmsg)))              #take absolute
value of the difference between original message and decoded message, the average will give bit
error rate

    return (error*100)



def string_similarity(original_message, message):   #input of 2 list
    original_message = ''.join(str(int(e)) for e in original_message)
    message = ''.join(str(int(e)) for e in message)
    similarity = SequenceMatcher(None, original_message, message)
    stringsim = (similarity.ratio()*100)
    return stringsim

def mp3_encode():
    subprocess.call(["ffmpeg",  "-y", "-i",  "sample.wav","compressed.mp3"], shell=True)
    #ffmpeg -y -i compressed.mp3 sample.wav
    subprocess.call(["ffmpeg",  "-y", "-i",  "compressed.mp3","sample.wav"], shell=True)
```

```
#ENCODING FUNCTION----------------------------------------------------------
def encode(sr, baud, f1, f0, msglen, startlen, seed_msg):
    #Initialise Variables
    T1 = 1/f1
    T0 = 1/f0
    A = 1
    dt = 1/sr

    bit_time = 1/baud
    t = scipy.arange(0,bit_time,dt)

    #returns a float array for 1 or 0
    def one():
        return 2*scipy.pi*f1*scipy.ones(int(sr/baud))
    def zero():
        return 2*scipy.pi*f0*scipy.ones(int(sr/baud))

    #Initialising original bit message-------------------------------------------------------------------------
-----------------------------------------
    original_bitmsg = []                            #array for original bit message
    random.seed(seed_msg)                                   #seed for generating random sequence
    for x in range(msglen):
        original_bitmsg += [random.randint(0,1)]

    start = []
    for x in range(1,startlen):                             #creation of start sequence
        start = start + x*[0]
        start = start + x*[1]

    message = 50*[1] + start +[0]+original_bitmsg +50*[1]

    omega = scipy.zeros(0)                          #array for phases
    for bit in message:
        if bit:
            omega = scipy.append(omega, one())
        else:
            omega = scipy.append(omega, zero())

    phases = scipy.cumsum(omega)*dt
    wave = A * scipy.sin(phases)

    scipy.io.wavfile.write("sample.wav", sr, wave)
    return wave
#FUNCTIONS FOR
FUN**************************************************************************
****************************
```

```
def cls():
    print("\n"*70)

def printf():
    scipy.set_printoptions(threshold=scipy.nan)
```