Generation and Detection of Dual Tone Multi Frequency Tones Using the Goertzel Algorithm

1.0 Literature Review

DTMF is acronym for Dual Tone Multi Frequency. When we press a key on the dial pad, a connection is made that generates two tones at the same time. The dual tones identify the key we pressed to any equipment we are controlling **[1].** If we are using a DTMF keypad to remotely control equipment, the tones can identify what unit we want to control, as well as which unique function we want it to perform is predefined by us. **[2]**

Applications of DTMF

1. Voice and electronic mail
2. Call forwarding
3. Displaying dialed numbers on screen
4. Automated locking and unlocking systems,
5. Sharing information consisting of numbers across network of systems.
6. Robot remote control

How DTMF Works

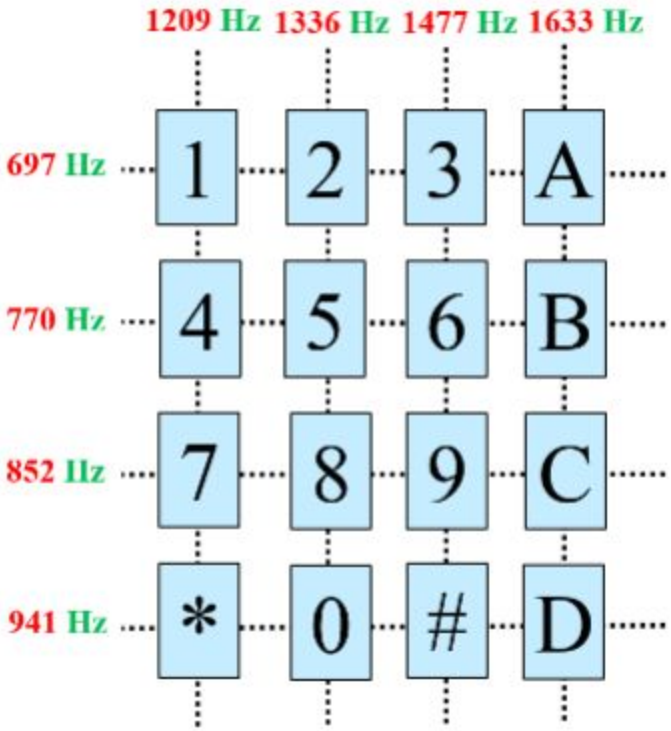


Figure 1.0: Telephone touch keypad

Each key on the keypad is assigned two specific frequencies. For instance if key '5' is pressed a DTMF signal with frequencies 770 Hz and 1336 Hz are generated. The signal is then processed and detection handled using digital filters to detect harmonic content of signal. Logical operations are performed to determine key pressed.

The intent of this project is to design a DTMF tone generator and decoder and implement it in Java Programming Language.

Research and Background

Multiple techniques and algorithms have been developed to detect the presence of known frequencies in a signal. The most common techniques used are **[3]**

1. FFT (Fast Fourier Transform)
2. Goertzel Algorithm

Fast Fourier Transform

The simplest way to detect presence of known harmonics in a given signal is to take the FFT of the signal and use band pass digital filters to check if desired harmonics are present. In the FFT method most of the computed results are discarded hence the method is computationally inefficient and expensive.

Goertzel Algorithm

The Goertzel Algorithm developed by Gerald Goertzel in 1958, is a digital signal processing technique that provides a more computationally efficient way of evaluating the individual terms of the **DFT**(Discrete Fourier Transform).**[4]** Hence it can be applied to applications like DTMF tone recognition.

The Goertzel Algorithm takes a different approach to the direct DFT computations. It applies a single real-valued coefficient at each iteration using real-valued arithmetic for real valued input sequence. Goertzel Algorithm has a higher order of complexity in contrast with the FFT Algorithms. However it is more computationally efficient to problems which involve only computation of a small number of selected frequency components hence makes it suitable in applications where there are limitations in processor memory available.

Comparison of Goertzel Algorithm and Fast Fourier Transform

The Goertzel Algorithm is more efficient than the FFT in computing an N-point DFT if less than $2 \log_2 N$ DFT coefficients are required **[5].** Goertzel Algorithm is therefore more suitable for DTMF detection applications since we only have 8 frequencies of interest.

Consequently i decided to use Goertzel Algorithm approach due to its superior computational efficiency over the FFT algorithms.

Prototyping

Prototyping was done in Matlab. My test dataset of test signals met the standards in the International Telecommunications Union (ITU-T) Recommendation Q.23. DTMF tones are standardized in the International Telecommunications Union's (ITU-T) Recommendation Q.23

Signal Frequencies:
Low Band in Hz : 697, 770, 852, 941
High Band in Hz : 1209, 1336, 1477, 1633

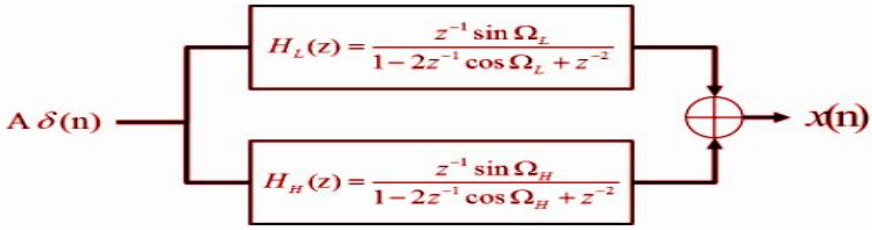
Frequency Tolerances
The tolerance specified in ITU-T Recommendation Q.23 [1] is 1,8 %. However, for Europe the tolerance is considered to be 1,5 % according to the practice as presented in Annex A of ITU-T Recommendation Q.24.

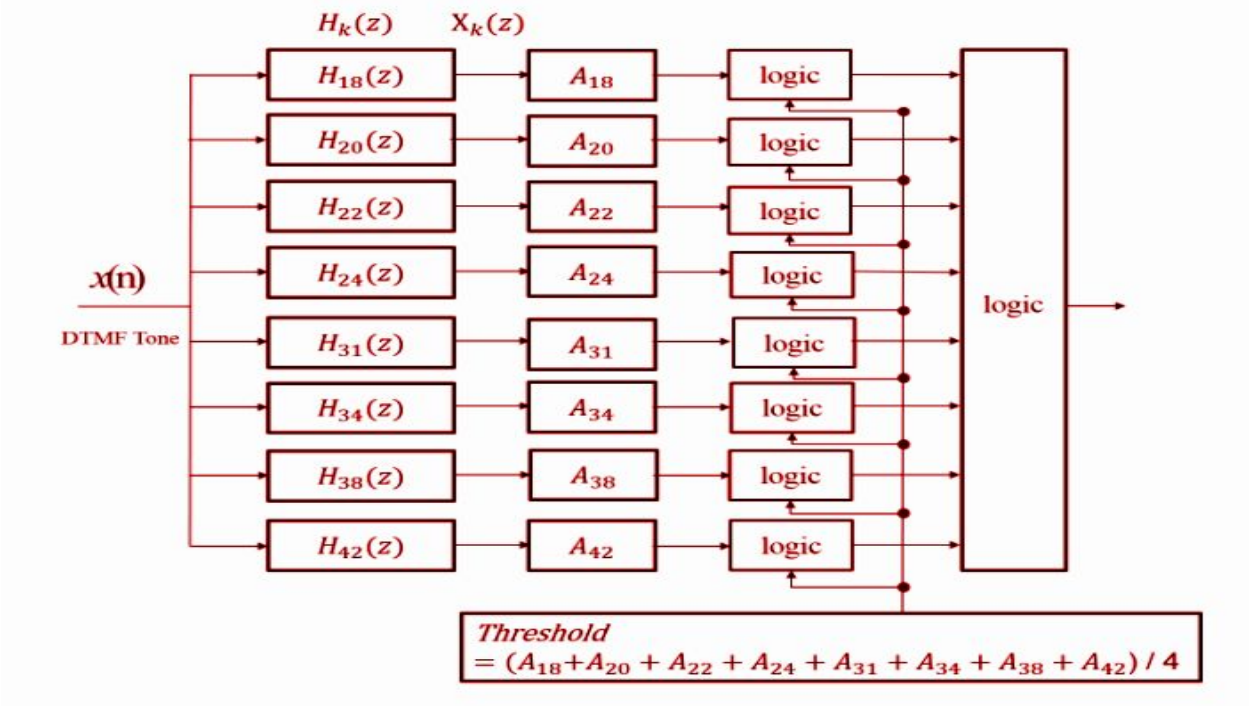
Signal Timing
Min accepted tone duration : 40ms
Min pause (silence between tones) : 30 ms to 70ms
Max rejected tone duration : 20 ms
Max signal interruption : 10 ms

Goertzel Algorithm

$$f_s = 8000 \text{ Hz}$$

$$\Omega = \frac{2\pi f}{f_s}$$





$$H_k(z) = \frac{X_k(z)}{X(z)} = \frac{1}{1 - 2\cos\left(\frac{2\pi k}{N}\right) z^{-1} + z^{-2}}$$

$$k = \frac{f}{f_s} \times N$$

$$N = 205$$

DTMF Frequency (Hz)	Frequency Bin : k
697	18
770	20
852	22
941	24
1209	31
1336	34
1477	38
1633	42

$$A_k = \frac{2}{N}\sqrt{|X(\mathrm{k})|^2}$$

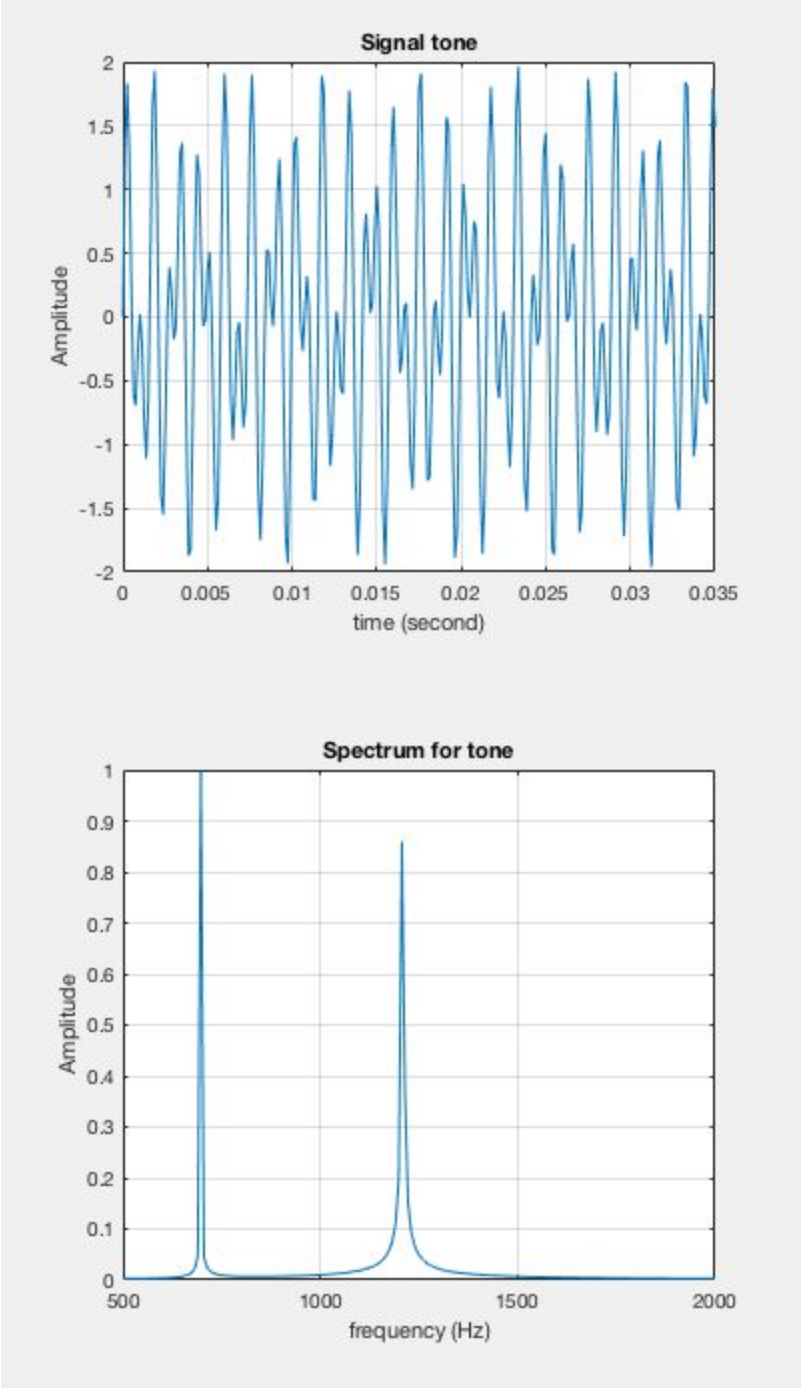
$$\text{Threshold} = (A_{18} + A_{20} + A_{22} + A_{24} + A_{31} + A_{34} + A_{38} + A_{42}) / 4$$

Test Results

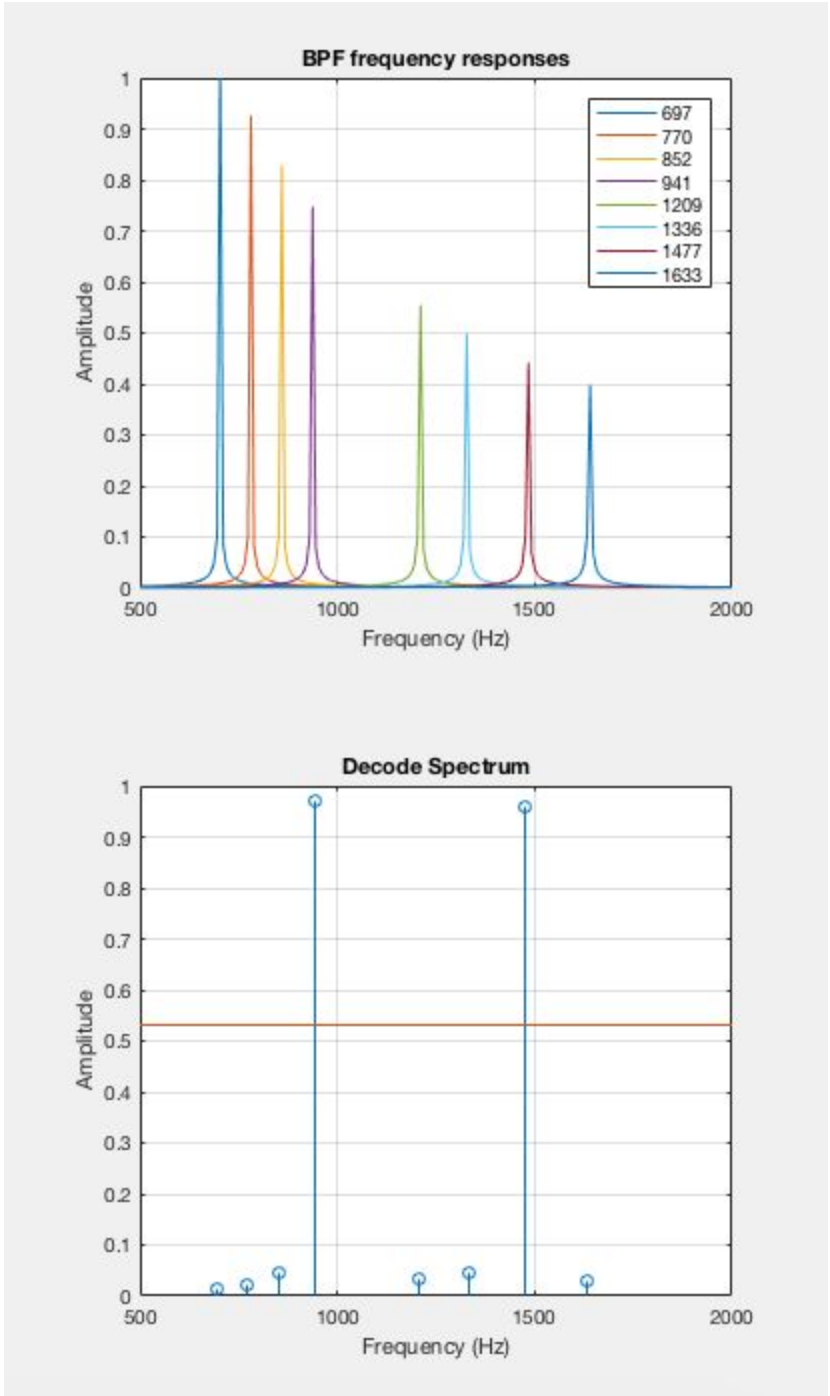
Key Pressed = 1

Frequency content = 697 Hz and 1209 Hz

Generated tone Spectrum



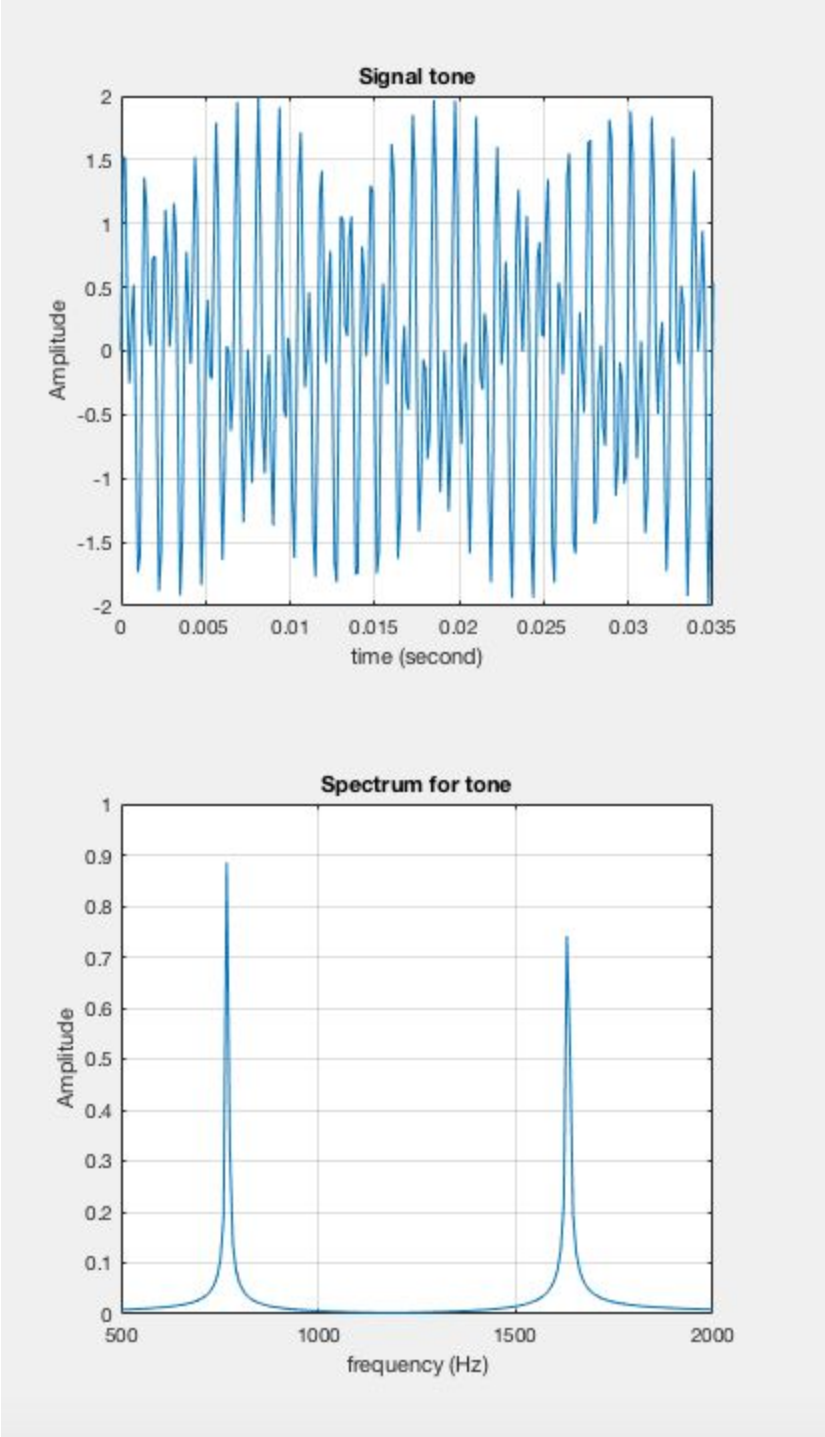
Band Pass Frequency Responses and Decoded Spectrum


BPF frequency responses
Amplitude
Frequency (Hz)
697
770
852
941
1209
1336
1477
1633
Decode Spectrum
Amplitude
Frequency (Hz)

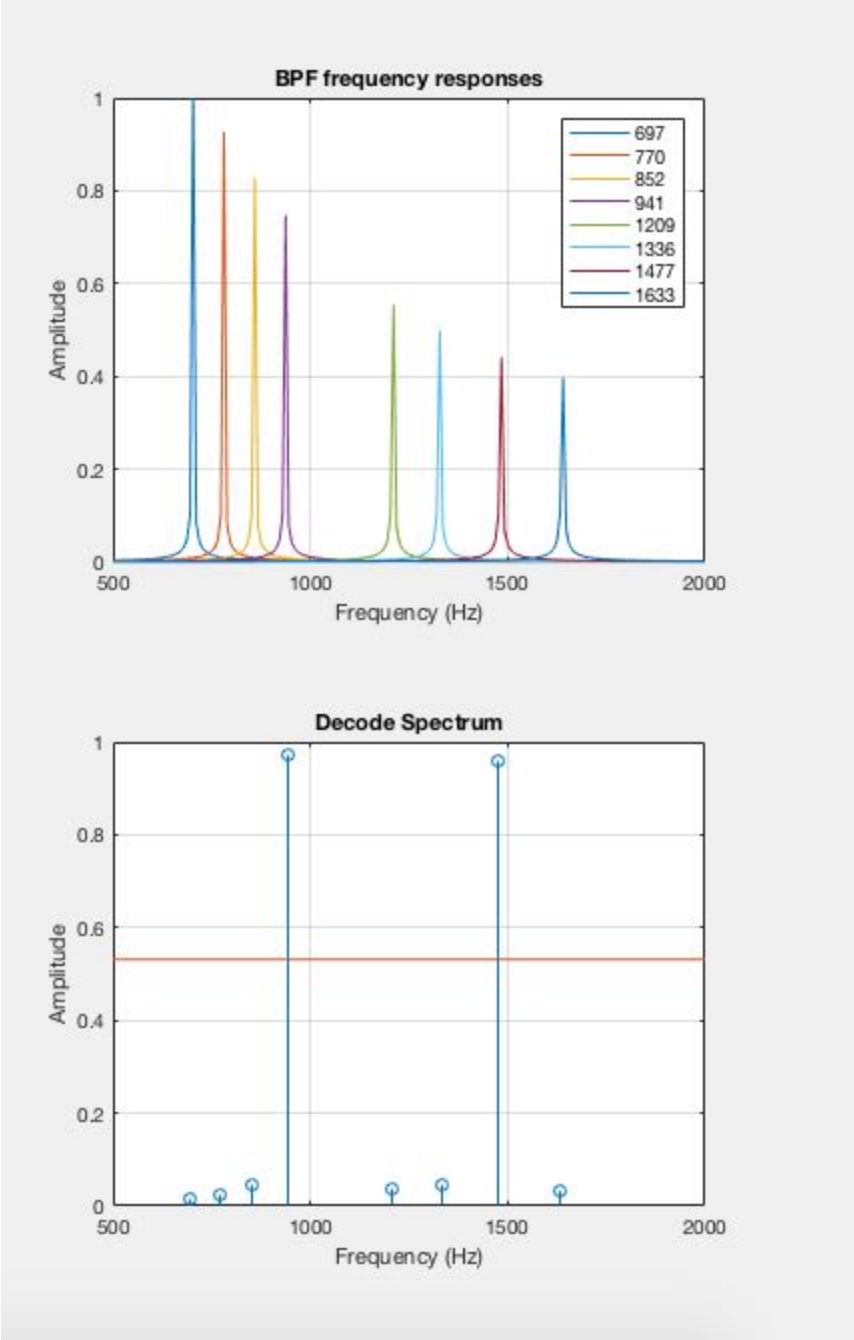
Keys pressed: B

Frequency content = 770 Hz and 1633 Hz

Generated tone Spectrum



Band Pass Frequency Responses and Decoded Spectrum



Tests carried out in Matlab were successful. The goertzel algorithm was further implemented In a Java DTMF dialer which runs Goertzel Algorithm under the hood and detects key dialed in real time.

Java Implementation

1. Audio Sampling

In Digital Signal Processing, sampling is the conversion of continuous time signal to discrete time signal. In Audio sampling, the **sampling rate** (in Hz) measures the number of audio samples generated per second. Low sampling rates result in antialiasing and information loss. The Nyquist-Shannon sampling theory states that signals should be sampled at least twice the maximum frequency in signal. The range of frequencies detected by human ear is 20Hz - 20 kHz, therefore sampling audio at 44100 Hz for Compact Disc Digital Audio and 48000 Hz for DVD is logical. **[6]**

Bit depth is the number of bits of information per sample. Bit depth corresponds to the resolution of each sample. In audio processing Compact Disc Digital Audio uses 16 bits/sample and **DVD-Audio** and **Blu Ray Disc** 24 bits/sample. Higher bit depths greater than 24 bits/sample and sampling rates of 96kHz or 192kHz only become beneficial when it comes to audio mixing in music production to prevent audio quality loss.

Bit depth = 16 bits/sample
Sampling rate = 41 000 Hz

Grid horizontal granularity is defined by the sampling rate whilst vertical granularity is defined by bit depth. Real life sounds however have richer harmonic spectrum. Adding and selecting right harmonics we can synthesize sounds from various instruments.

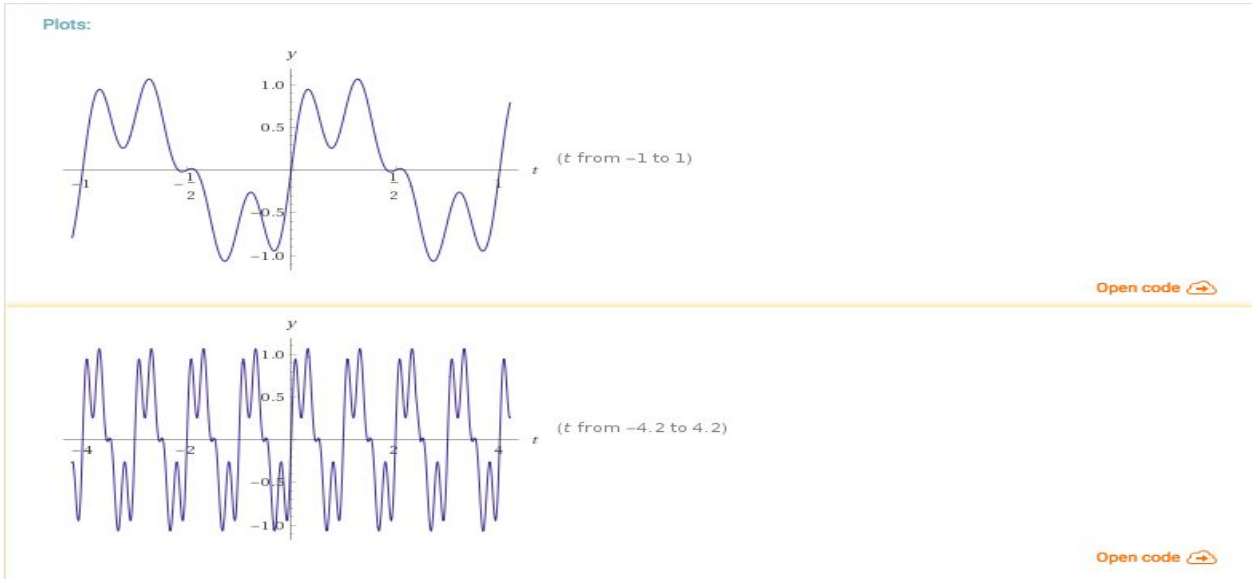


Figure 3: Continuous complex wave

Figure 4: Discrete Complex wave

```
3    double sampleRate = 44100.0;
4    double seconds = 2.0;
5    double f0 = 440.0;
6    double amplitude0 = 0.8;
7    double twoPiF0 = 2 * Math.PI * f0;
8    double f1 = 6 * f0;
9    double amplitude1 = 0.2;
10   double twoPiF1 = 2 * Math.PI * f1;
11   float[] buffer = new float[(int) (seconds * sampleRate)];
12   for (int sample = 0; sample < buffer.length; sample++) {
13       double time = sample / sampleRate;
14       double f0Component = amplitude0 * Math.sin(twoPiF0 * time);
15       double f1Component = amplitude1 * Math.sin(twoPiF1 * time);
16       buffer[sample] = (float) (f0Component + f1Component);
17   }
```

Figure 2.0: Code computes samples and store the values in a buffer

Note: A frequency of 440 Hz was chosen since it is used as a general standard for music pitch.

2. Java Audio Generation in Java

The float buffer is converted into Pulse Code Modulated (PCM) bytes which are supported by sound card.

```
19       /* To make sound audible we can write the output to a WAVE file.
20          WAVE file consists of a header followed by sound data[which is the PCM format
21          we computed and stored in byteBuffer array]
22          We can write header using Java library javax.sound.sampled
23       */
24       File outfile = new File ("outfile.wav");
25       boolean bigEndian = false;
26       boolean signed = true;
27       int bits = 16;
28       int channels = 1;
29       AudioFormat format;
30       format = new AudioFormat(sampleRate, bits, channels, signed, bigEndian);
31       ByteArrayInputStream bais = new ByteArrayInputStream(byteBuffer);
32       AudioInputStream audioInputStream;
33       audioInputStream  = new AudioInputStream(bais, format, buffer.length);
34       AudioSystem.write(audioInputStream, AudioFileFormat.Type.WAVE, outfile);
35       audioInputStream.close();
```

Figure 2.1: Code to generate WAV file in Java

```
39      /*
40      * Play Buffer Directly
41      */
42      SourceDataLine line;
43      Dataline.Info info;
44      infor = new Dataline.Info(SourceDataLine.class,formart);
45      line =(SourceDataLine) AudioSystem.getLine(infor);
46      line.open(fomart);
47      line.start();
48      line.write(byteBuffer, 0, byteBuffer.length);
49      line.close();
```

Figure 2.2: Code to produce audio to speaker

3. DTMF Detection Implementation Goertzel Algorithm

In audio processing operations are done in blocks because operations on individual samples are not time efficient and practical. To estimate the frequency of an audio at least one complete period of signal is required. In audio processing a block size of 1024 samples is recommended in audio processing **[7].**

Sample rate = 44.1 kHz
Block size = 1024 samples
Therefore one block of 1024 samples is equivalent to 23ms of audio

4. Goertzel Algorithm Implementation

Goertzel Class

The Goertzel class contains an implementation of the Goertzel algorithm. It can be used to detect if one or more predefined frequencies are present in a signal. The Custom Goertzel Class implements the AudioProcessor Interface. The AudioProcessor is responsible for the actual data processing Interface: A buffer with some floats and the same information in raw bytes

```java
    /**
     * If the power in dB is higher than this threshold, the frequency is
     * present in the signal.
     */
    private static final double POWER_THRESHOLD = 37;// in dB

    /**
     * A list of frequencies to detect.
     */
    private final double[] frequenciesToDetect;
    /**
     * Cached cosine calculations for each frequency to detect.
     */
    private final double[] precalculatedCosines;
    /**
     * Cached wnk calculations for each frequency to detect.
     */
    private final double[] precalculatedWnk;

       /**
     * A calculated power for each frequency to detect. This array is reused for
     * performance reasons.
     */
    private final double[] calculatedPowers;

    private final FrequenciesDetectedHandler handler;
```

Figure 2.3:

```java
    public Goertzel(final float audioSampleRate, final int bufferSize,
            double[] frequencies, FrequenciesDetectedHandler handler) {

        frequenciesToDetect = frequencies;
        precalculatedCosines = new double[frequencies.length];
        precalculatedWnk = new double[frequencies.length];
        this.handler = handler;

        calculatedPowers = new double[frequencies.length];

        for (int i = 0; i < frequenciesToDetect.length; i++) {
            precalculatedCosines[i] = 2 * Math.cos(2 * Math.PI
                    * frequenciesToDetect[i] / audioSampleRate);
            precalculatedWnk[i] = Math.exp(-2 * Math.PI
                    * frequenciesToDetect[i] / audioSampleRate);
        }
    }
```

Figure 2.4: Goertzel function to populate the precalculated Cosines array and precalculated Wnk(s) array.

The Goertzel function takes in the Sampling Rate, size of Buffer with samples, the frequencies [set of all possible DTMF frequencies] as Arguments.

Goetzel class implements the AudioProcessor Interface

```java
public interface AudioProcessor {
    /**
    * Process the first (complete) buffer. Once the first complete buffer is
    * processed the remaining buffers are overlapping buffers and processed
    * using the processOverlapping method (Even if overlap is zero).
    * @param audioFloatBuffer The buffer to process using the float data type.
    * @param audioByteBuffer  The buffer to process using raw bytes.
    * @return                 False if the chain needs to stop here, true otherwise. This can be used to imple
    * e.g. a silence detector.
    */
    boolean processFull(final float[] audioFloatBuffer, final byte[] audioByteBuffer);

    /**
    * Do the actual signal processing on an overlapping buffer. Once the
    * first complete buffer is processed the remaining buffers are
    * overlapping buffers and are processed using the processOverlapping
    * method. Even if overlap is zero.
    * @param audioFloatBuffer The buffer to process using the float data type.
    * @param audioByteBuffer  The buffer to process using raw bytes.
    * @return False if the chain needs to stop here, true otherwise. This can be used to implement e.g. a sile
    */

    boolean processOverlapping(final float[] audioFloatBuffer, final byte[] audioByteBuffer);
```

Figure 2.5:

```java
    @Override
    public boolean processFull(float[] audioFloatBuffer, byte[] audioByteBuffer) {
        double skn0, skn1, skn2;
        int numberOfDetectedFrequencies = 0;
        for (int j = 0; j < frequenciesToDetect.length; j++) {
            skn0 = skn1 = skn2 = 0;
            for (int i = 0; i < audioFloatBuffer.length; i++) {
                skn2 = skn1;
                skn1 = skn0;
                skn0 = precalculatedCosines[j] * skn1 - skn2
                        + audioFloatBuffer[i];
            }
            double wnk = precalculatedWnk[j];
            calculatedPowers[j] = 20 * Math.log10(Math.abs(skn0 - wnk * skn1));
            if (calculatedPowers[j] > POWER_THRESHOLD) {
                numberOfDetectedFrequencies++;
            }
        }

        if (numberOfDetectedFrequencies > 0) {
            double[] frequencies = new double[numberOfDetectedFrequencies];
            double[] powers = new double[numberOfDetectedFrequencies];
            int index = 0;
            for (int j = 0; j < frequenciesToDetect.length; j++) {
                if (calculatedPowers[j] > POWER_THRESHOLD) {
                    frequencies[index] = frequenciesToDetect[j];
                    powers[index] = calculatedPowers[j];
                    index++;
                }
            }
            handler.handleDetectedFrequencies(frequencies, powers,
                    DTMF.DTMF_FREQUENCIES, calculatedPowers.clone());
        }

        return true;
    }
```

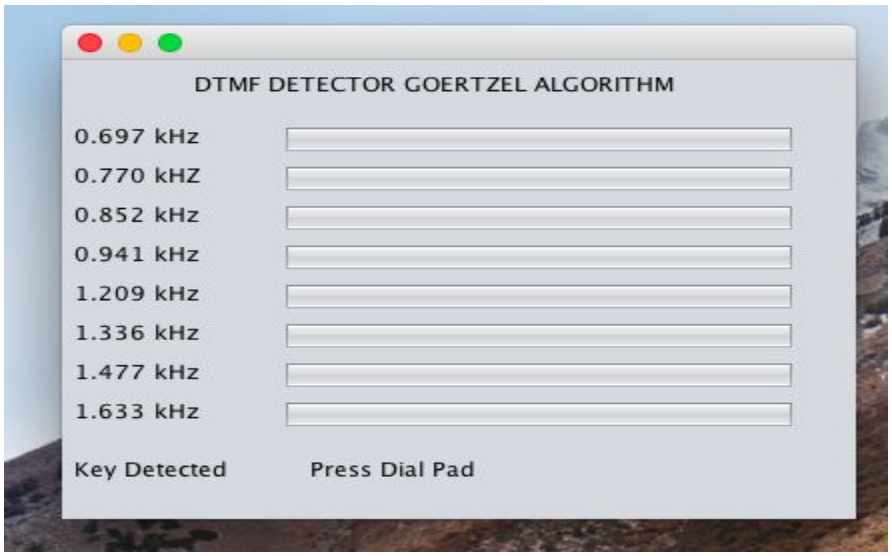
```
118     @Override
119     public boolean processOverlapping(float[] audioFloatBuffer,
120             byte[] audioByteBuffer) {
121         processFull(audioFloatBuffer, audioByteBuffer);
122         return true;
123     }
124
125     @Override
126     public void processingFinished() {
127     }
128
129 }
```

Finally we have the Dtmf_GUI class which runs the Goertzel class under the hood to detect the frequencies present hence the character pressed.

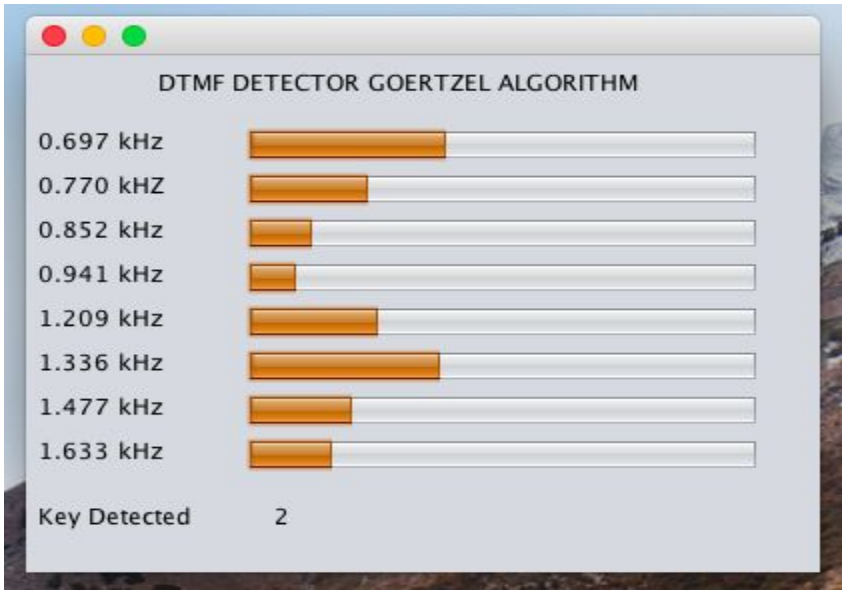
```
327     /**
328     * Process a DTMF character: generate sound and decode the sound.
329     * @param character The character.
330     * @throws UnsupportedAudioFileException
331     * @throws LineUnavailableException
332     */
333     public void process(char character) throws UnsupportedAudioFileException, LineUnavailableException{
334     final float[] floatBuffer = DTMF.generateDTMFTone(character);
335     final AudioFormat format = new AudioFormat(44100, 16, 1, true, false);
336     final AudioFloatConverter converter = AudioFloatConverter.getConverter(format);
337     final byte[] byteBuffer = new byte[floatBuffer.length * format.getFrameSize()];
338         converter.toByteArray(floatBuffer, byteBuffer);
339     final ByteArrayInputStream bais = new ByteArrayInputStream(byteBuffer);
340     final AudioInputStream inputStream = new AudioInputStream(bais, format,floatBuffer.length);
341     final AudioDispatcher dispatcher = new AudioDispatcher(inputStream, stepSize, 0);
342     dispatcher.addAudioProcessor(goertzelAudioProcessor);
343     dispatcher.addAudioProcessor(new BlockingAudioPlayer(format, stepSize, 0));
344     new Thread(dispatcher).start();
345     }
```

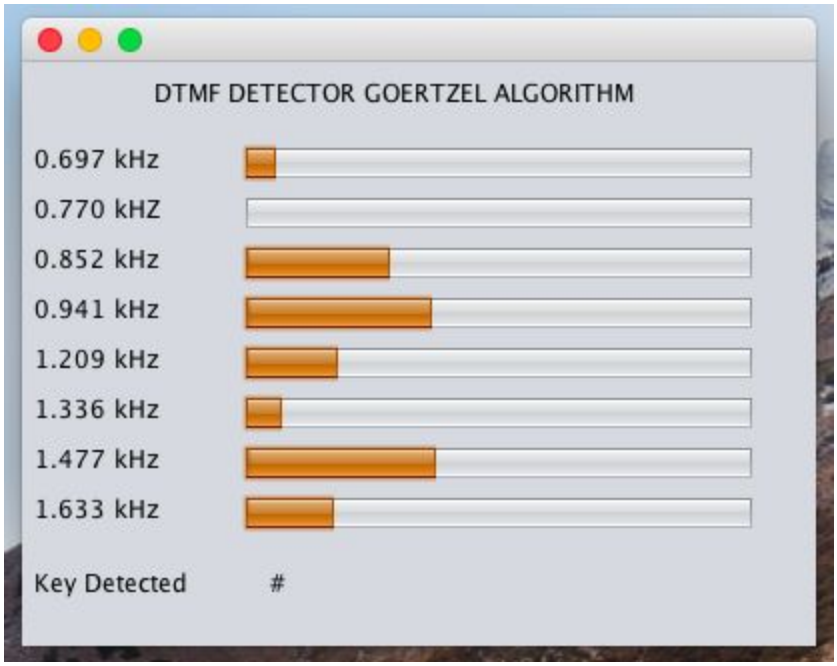
Graphical User Interface Output

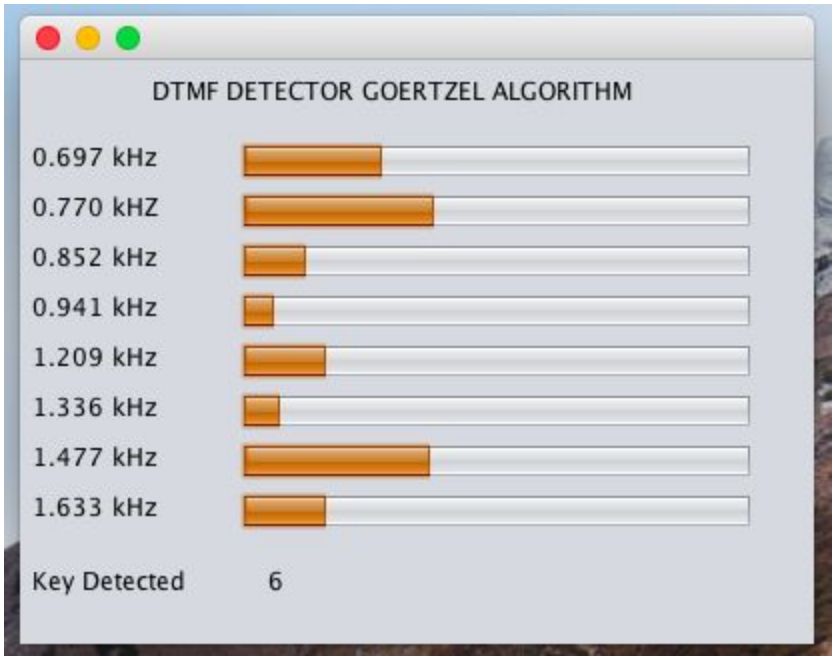
The java program runs the goertzel algorithm under the hood to detect the key pressed in real time. The power bars determine the harmonic content of each the 8 possible frequency content of DTMF signal generated by pressing the dial key on keyboard.



Below are results from running the java program







Results from the Java program showed a 100% success rate.
Source code for the project can be found on https://github.com/Chitova263/Dsp-DTMF

Conclusions

Recommendations

References

1.Shekhawat, Rajesh Singh & Saini, Amit & Kumar Bhataia, Ravinder. (2016). DTMF Controlled Robotic Car. International Journal of Engineering Research and V5.10.17577/IJERTV5IS010548.

2.Abah O. Sunday, Visa M. Ibrahim, Abah Joshua," Remote Control of Electrical Appliances Using GSM Networks", International Journal of Engineering Research and Development, Volume 1, Issue 9 (June 2012), PP.38-45, ISSN: 2278-067X

3. http://www.mstarlabs.com/dsp/goertzel/goertzel.html

4. Mock, P. (March 21, 1985), "Add DTMF Generation and Decoding to DSP-µP Designs" (PDF), *EDN*, ISSN 0012-7515; also found in DSP Applications with the TMS320 Family, Vol. 1, Texas Instruments, 1989.

5. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Prentice-Hall, 1990.

6. https://wiki.audacityteam.org/wiki/Talk:Sample_Rates**.**

7. Steven W. Smith, Ph.D The Scientist and Engineer's Guide to Digital Signal Processing