DTMF-DECODER

A report on a project to design and implement a DTMF Decoder

Project Page - http://tino1b2be.github.io/DTMF-Decoder

NOVEMBER 30, 2015

MENTOR – ALBERT VISAGIE
VASTech (Pty) Ltd

Contents

Introduction

DTMF stands for Dual Tone Multi Frequency. This is an in-band telecommunication signalling system using voice-frequency band over telephone lines between telephone equipment and other communications devices and switching centres. DTMF was first developed in the Bell System in the USA, and became known under the trademark Touch-Tone for use in the pushbutton telephones supplied to telephone customers, starting in 1963.[1] DTMF replaced the clockwork dial made of moving parts which was getting more and more impractical to use as the number of telephone users increased exponentially.

DTMF is used to represent up to 16 keys (most telephones only use 12 of these). Each key is represented by two different frequencies. The first bin (lower frequencies) consist of frequencies under 1kHz and the second bin (Upper bin) consists of frequencies above 1.2kHz. The combination of the two tones will be distinctive and different from tones of other keys and these tones cannot be mimicked by voice or random signals [2]. The DTMF Frequencies are shown in the figure below:

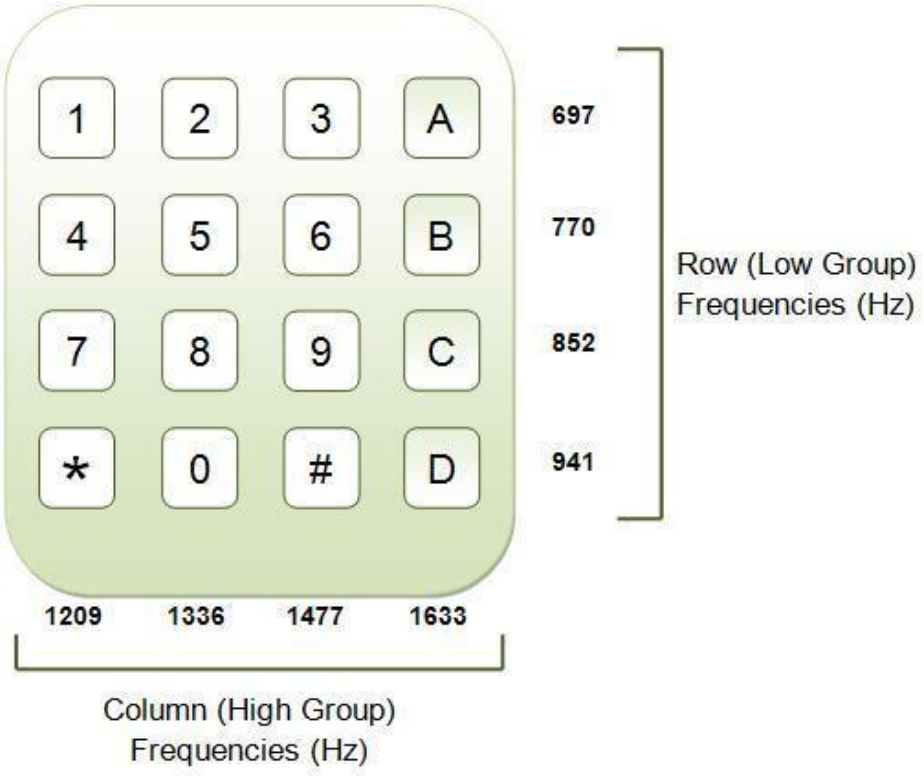

Illustration 1: Picture showing the DTMF Frequencies used to represent each key.

DTMF detection is used to detect DTMF signals in the presence of speech and dialling tones pulses. Other applications of DTMF include, but not limited to, computer applications such as voice and electronic mail, call forwarding [3], controlling robots, automated locking and unlocking systems, displaying dialled numbers on a screen and sharing information consisting of numbers across various network systems. [2]

The intent of this project is to design a DTMF Decoder and implement it in Java. I decided to choose this project because it dives into important concepts in both “sides” of my degree, that is Signal Processing and Software Design.

Research and Background

Although I did an introductory course to Signals and Systems in first semester of my second year, there was still a lot of concepts I had not grasped yet. Most of the knowledge needed to tackle this project was only going to be covered in my third year courses therefore after a long talk with my mentor who shed some much needed light on the project, I spent the whole first week researching on signal processing techniques by reading articles [4] and lectures on MIT OpenCourseWare [5]. All these helped build my understanding of signal processing and detection.

Fast Fourier Transform

This is a DFT algorithm(s) which can compute the DFT of a sequence in less computations for the same number of sample points. Cooley and Turkey and usually credited for inventing the algorithm in 1965 although the development of fast algorithms of FFTs can be traced back to Gauss's unpublished work in 1805. [8]

A possible path to take could be using the FFT to get the whole frequency spectrum. This is definitely a cumbersome computation but it will allow me to detect other frequencies in the signal and decide where certain signals should be rejected or not.

After looking at previous similar projects, most of which were implemented in Assembly and C for micro-controller projects, I noticed most of them used the Goertzel Algorithm to detect the DTMF frequencies. I decided to do more research on reasons why and it turns out it is a much faster method for computing DFTs (Direct Fourier Transform) for specific frequencies.

Goertzel's Algorithm

Created by Gerald Goertzel in 1958, the Goertzel Algorithm is a Digital Signal Processing technique that is used to efficiently compute individual terms of a DFT. It returns the real and imaginary frequency components that a regular DFT or Fast Fourier Transform (FFT) would.

Like the DFT, the Goertzel algorithm analyses one selectable frequency component from a discrete signal.[6][7] Unlike direct DFT calculations, the Goertzel algorithm applies a single real-valued coefficient at each iteration, using real-valued arithmetic for real-valued input sequences. For covering a full spectrum, the Goertzel algorithm has a higher order of complexity than (FFT) algorithms; but for computing a small number of selected frequency components, it is more numerically efficient.

The Goertzel Algorithm may be used to directly obtain DFT data about the 8 DTMF frequencies and compare their magnitudes to determine which tone is being represented.

Software Implementation

After the research I was advised to start by prototyping the solution in MatLab before implementing it in Java. MatLab is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. [9] Because I was new to signal processing, it was important for me to use a tool like MatLab to visualise the data and focus on developing the algorithms instead of spending a lot of time trying to code functions which I had easy access to in MatLab.

Test Data

The first step was to generate test data that I would use to test the decoder. My mentor gave me a guideline of how a good DTMF decoder should perform and told me to look up ITU-T recommendations on DTMF tones. The decoder had to meet the ITU Recommendation as stated in ITU-I Recommendation Q.23 [10] and ITU-I Recommendation Q.24 [11]. Each administration has its own specifications but most of them overlap each other. A summary of the recommendations is shown below (taken from ITU-T Q.24 [11]):

Signal Frequencies:

- Low Band (Hz): 697, 770, 852, 941
- High Band (Hz): 1209, 1336, 1477, 1633

Frequency Tolerances:

- Max. accepted frequency offset: <=1.5% to 1.8% (depending on Administration)
- Min. rejected frequency offset: >= 3.5% to 7.5% (depending on Administration)

Power Levels per Frequency:

- Power range: -27dBm to 0dBm
- Maximum twist (Power difference between the two frequencies): 5 to 10 dBm (depends on Administration)

Signal Timing:

- Min. accepted tone duration: 40ms
- Min. pause (silence between tones): 30ms to 70ms (depending on Administration)
- Max. rejected tone duration: 20ms
- Max. signal interruption: 10ms

I had to generate Test Data to test all the conditions above. My Test Data was generated as follows:

- The sequence of tones will be randomly determined (choosing from the 16 available DTMF characters) and the length of the sequence will also vary randomly from 5 to 50 tones per file.
- The duration of each DTMF tone will be varied randomly between 40ms and 100ms.

- The duration of the pause between DTMF tones will be randomly varied between 30ms and 100ms.
- To vary the power of the signal between 0dBm and -27dBm, the amplitude of the signal will be varied from 0.045 (-27dBm) to 1.0 (0dBm).

Random duration of pauses and tones will ensure that the frames used in the decoding process do not have the same alignment.

Prototyping

This was done in MatLab. I decided to use the Goertzel's Algorithm approach because of its performance advantages over the FFT. After some brain storming with my mentor, I came up with the following outline of the decoder:

- The array of samples will be split into short frames (each frame will be a short period of about 40ms but this duration will be optimized later). Because the Goertzel Algorithm's performance did not depend on a specific bin size (as with the FFT where the bin size has to be a power of 2), I could vary the bin size to make the decoder more accurate. The frames overlap each other by 50% which means more frames are processed, hence decreasing performance, but this will make sure the frames cover the shortest tones properly. The longer these frames were, the better the resolution will be in the frequency spectrum but the poorer it would be in the time domain. An example of how the samples will be divided is shown below.

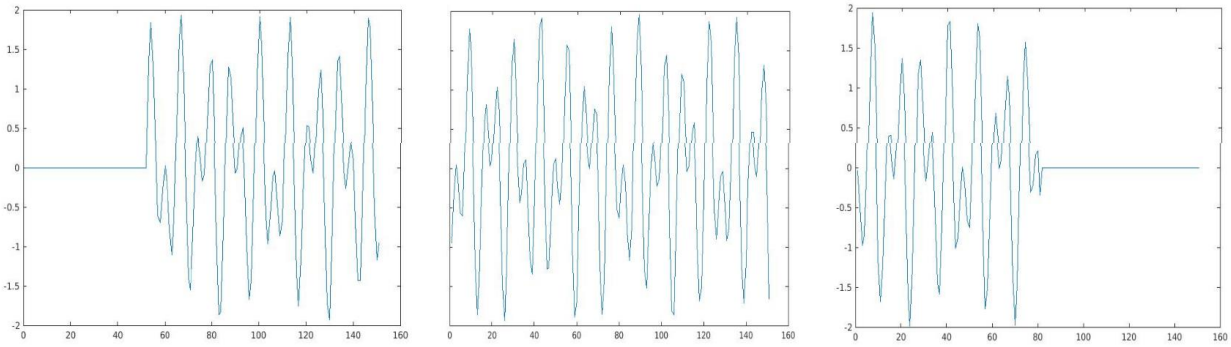

Illustration 2: An example of how the samples may be split to create frames which can be processed separately.

- The frames will each be transformed using the Goertzel Algorithm to obtain the magnitudes of the 8 DTMF frequencies in the frame. One of the ITU-T recommendations is that the frequencies can have an offset of up to 1.5%. This offset could be accounted for by using 3 or 4 frequencies for each DTMF frequency when transforming the frames and then summing up these 2 or 3 magnitudes. So instead of passing 8 frequencies into the Goertzel Algorithm, I used about 25 different frequencies. The two illustrations below show how the plots of the magnitudes from a frame in the “tone” region and in the “pause” region:

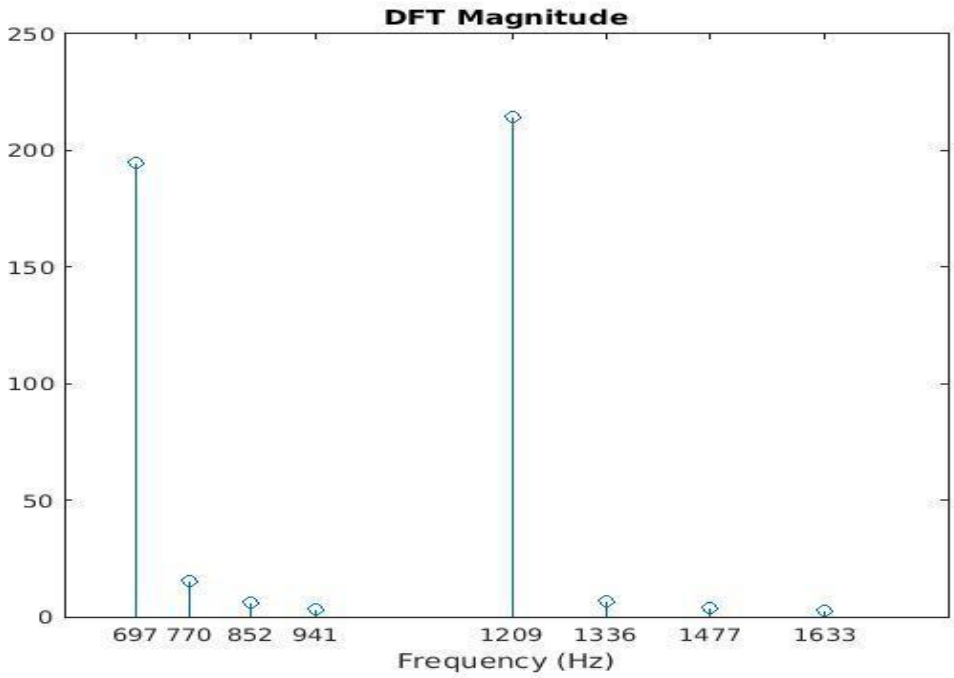

Illustration 3: Plot of the DFT Magnitudes of the 8 DTMF frequencies for a frame in the "tone" region. This particular tone represents a "1".

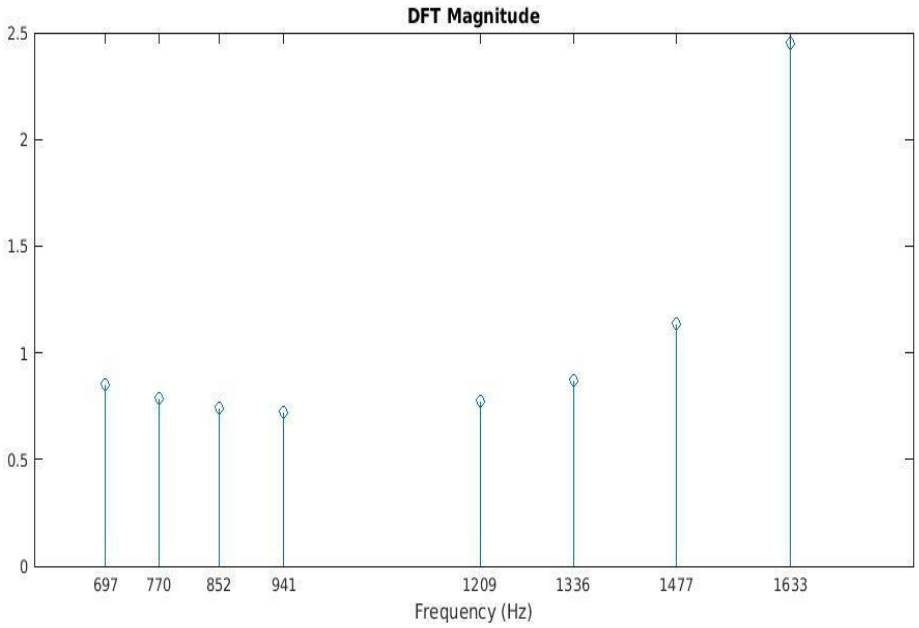

Illustration 4: Plot of the DFT Magnitudes of the 8 DTMF frequencies for a frame in the "pause" region.

- After transforming the frames, I used the DFT data to distinguish between frames in the “tone” region and frames in the “pause” region. From the pictures above, it can be seen that the average

of the magnitudes for the “tone” region are much larger than those in the “pause” region. To distinguish the “pause” frames from the “tone” ones, I decided that the “silent” frames will be those with a mean DFT magnitude less than 70% of the average of the top 3 frames. An illustration of this is shown below:

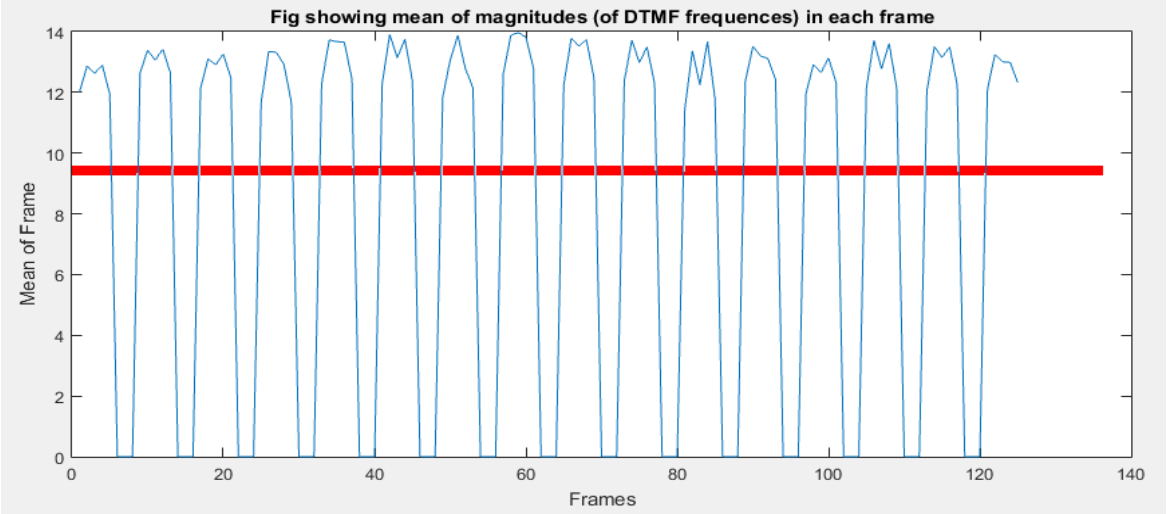

Illustration 5: *Plot of the DFT Magnitudes of the 8 DTMF frequencies for a frame in the "pause" region.*

- After filtering out the frames with the tones, I pass the data into a function which will determine the character represented by the frame by looking at the frequencies present in the frame.

After implementing this algorithm in MatLab I ran the decoder on test data and after tweaks and changed, the decoder had a success rate of 98% on 40,000 test files. A success is recorded when the whole sequence has been successfully and accurately decoded. The changes I made include having the frames overlap by 33 % instead of 50% and increasing the frame size to 46ms. This increased my frequency resolution and because there were more frames now, the time domain resolution was not affected much.

Java Implementation

Implementing the MatLab code in java was a challenge because Java does not have libraries with most of the functions I used in MatLab. After some research I found a webpage that gave a good explanation of the Goertzel Algorithm and used this to implement my own Goertzel Class in Java.

Just like the FFT, the algorithm works with blocks of sample points. For each frequency to be calculated, the following are needed:

- the sampling frequency (**Fs**).
- block size (**N**) i.e number of samples being used.
- Precomputed one sine and one cosine term. (**s** and **c**)

- Precomputed coefficient (**coeff**)

The usual Nyquist Rules apply which means that **Fs** will have to be at least twice that of the highest frequency in question. The frequency of interest has to be an integer factor of the sampling frequency.

N has the same properties as that of the block size in a regular FFT. **N** controls the frequency resolution in the frequency spectrum of the signal which is given as Fs/N. **N** can be chosen such that the frequency resolution covers enough of the DTMF frequencies within their tolerances but it must also be small enough to cover the minimum tone duration. One of the biggest advantages of the Goertzel Algorithm is that unlike the FFT, **N** does not have to be a power of 2 for it to work efficiently.

The following is the pseudocode I used

```
Samples defined here
target = target frequency
Fs = sampling frequency

N = number of samples
k = (int)(0.5*N*target*1/Fs)
ω = 2 * π * k / N;
c = cos(ω);
coeff = 2 * c;

Q0 = Q1 = Q2 = 0;
for each sample s in samples:
  Q0 = coeff * Q1 - Q2 + s
  Q2 = Q1
  Q1 = Q0
end
```

$magnitude^2 = Q_1^2 + Q_2^2 - Q_1 * Q_2 * coeff$

After some consultation with my Mentor Albert, I realised that my algorithm was slow and used up a lot of RAM because I was loading the whole .wav file before decoding. This would be a problem when decoding thousands of files in parallel. The PC would quickly run out of memory. To combat this I had to use an input stream and process the wav files frame-by-frame instead of loading the whole file. This proved to be much faster and also more efficient in terms of resource usage.

The basic outline of the algorithm is as follows:

- Open an input stream for the audio file
- determine the frame size for the Goertzel transform
- Read a frame and check if the signal's power is not too low (not lower than the -27dBm as per ITU-T recommendations.) If the power is too low, the frame will be rejected and treated as a pause.

- The frame is transformed using Goertzel function to return the powers of the 8 DTMF Frequencies.
- To check for noise, the ratio of the sum of the two highest peaks from the lower and upper frequency bin to the sum of all the 8 frequencies is compared to a predetermined value. If the ratio is too low then the frame is noisy thus rejected and treated as a pause.
- The two DTMF peaks from each bin are then used to identify the character.
- The character for this frame is stored for later usage. If the same character appears 2 or more times consecutively then this will be recorded as a hit.
  The number of consecutive characters to flag a hit depends on the minimum tone duration being used in the decoding process. If we use the ITU-T recommendation of 40ms, there must be at least 2 consecutive characters, is 60ms is used, there must be 3, if 80ms is used, there must be 4, and so forth.

Testing

The decoder performed very well using the Goertzel function. It reported the same results as the MatLab code. That is, a success rate of 98% and a hit rate of 99.7% for 40,000 test files.

Noise and Speech

MatLab has a function called "Add White Gaussian Noise" **y = awgn(x,snr)** which allows the user to add white Gaussian noise to a signal, **x**, to produce another signal, **y,** with a **SNR** given specified by the user in decibels. I used this function to noise to the existing test data to produce test data with SNR ranging from 0db to 30db.

The noise performance of the Goertzel DTMF decoder was impressive. The test results show that the decoder can handle signals with a SNR of XXDB perfectly well. A plot of the success rate vs SNR is shown below:

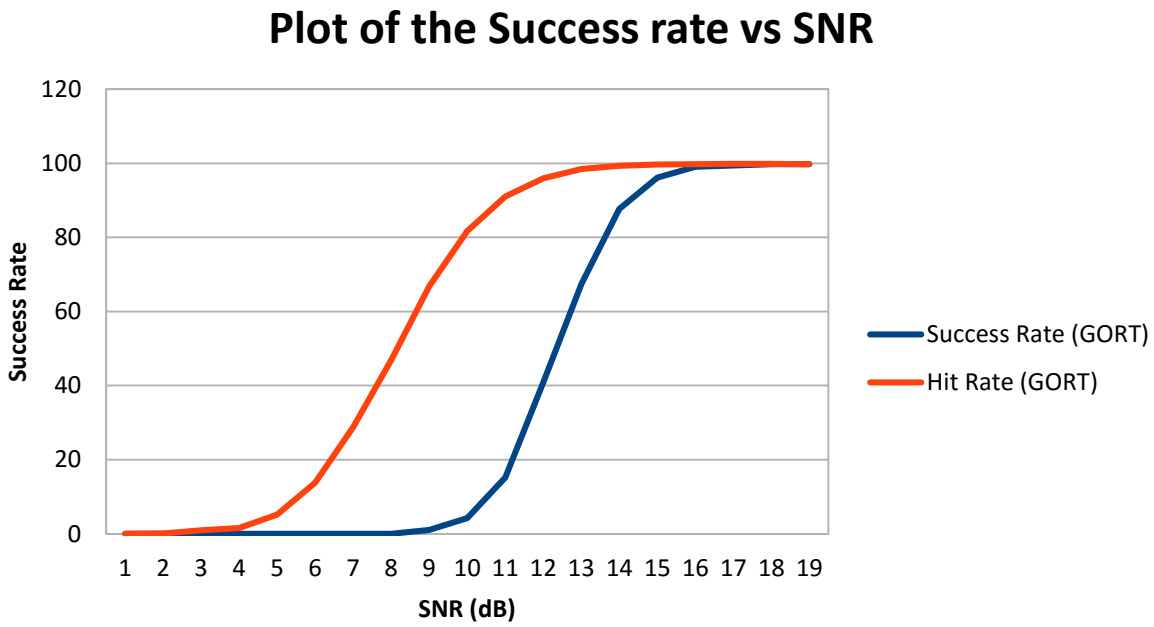

Illustration 6: Plot of the success rates of the Goertzel DTMF Decoder vs SNR.

The decoder was run on random audio recordings with human speech and the performance was very poor, with the duration of these recordings varying from 10s to 60s. Out of the 15,000 files that were tested, it reported that DTMF tones were found in 68% of the files but in fact very few of the files had any DTMF tones in them (less than 3%). This poor performance could easily be attributed to background music and the speech itself within the recordings. I tweaked the parameters as much as possible but there was no improvement at all and this is when my mentor advised me to try out the FFT approach instead of Goertzel one. By using the FFT approach, I will have access to the complete frequency spectrum of the signal allow me to detect other frequency peaks caused by speech and music that would not appear within the DTMF frequency bins. I also realized that the longer my "minimum tone duration" is, the better the filtering out

of the audio files it. The disadvantage of this is that the decoder does not follow ITU-T recommendations anymore and thus might actually miss some DTMF tones shorter than the extended durations.

I modified the code to use the FFT instead of the Goertzel's Algorithm. The only difference in the algorithm is when checking for noise. Instead of calculating the detection ratio over just 8 frequencies, the detection ratio will be now calculated using all points in the power spectrum of the signal. This will make sure that other peaks that aren't DTMF will be filtered out. I ran the decoder again on the same audio recordings and plotted the percentage of files found to have DTMF tones vs the minimum tone duration used to decode the files:

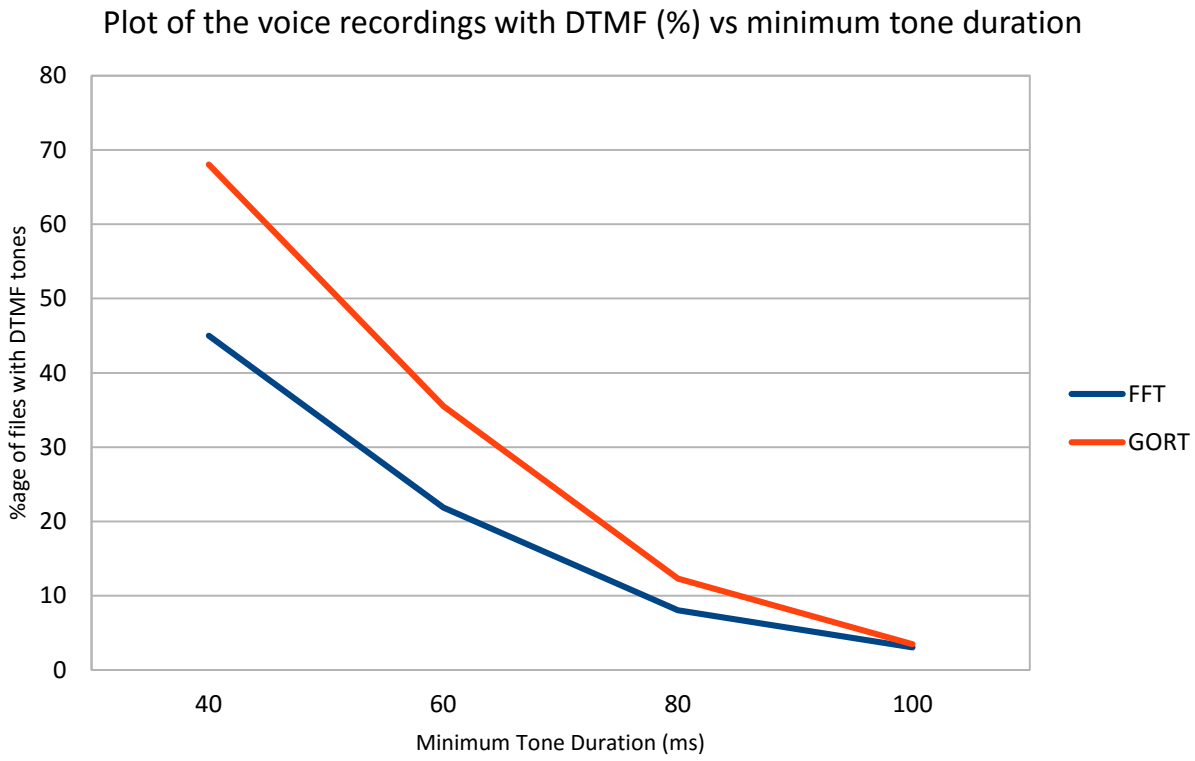

Illustration 7: Plot showing the number of files found to have DTMF vs the minimum tone duration used

Goertzel vs FFT

I ran noise tests on the decoder with the new FFT implementation and the performance was almost the same with the Goertzel implementation although the FFT method seems to have a better hit-rate at lower SNR. The plots to compare the two are shown below:

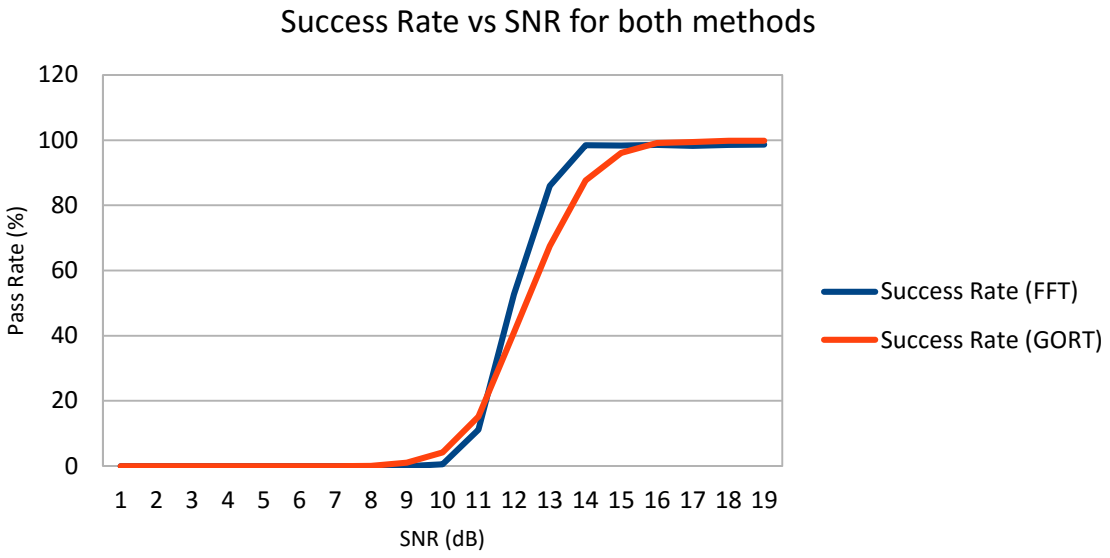

Illustration 8: Plot of the Success rate vs SNR

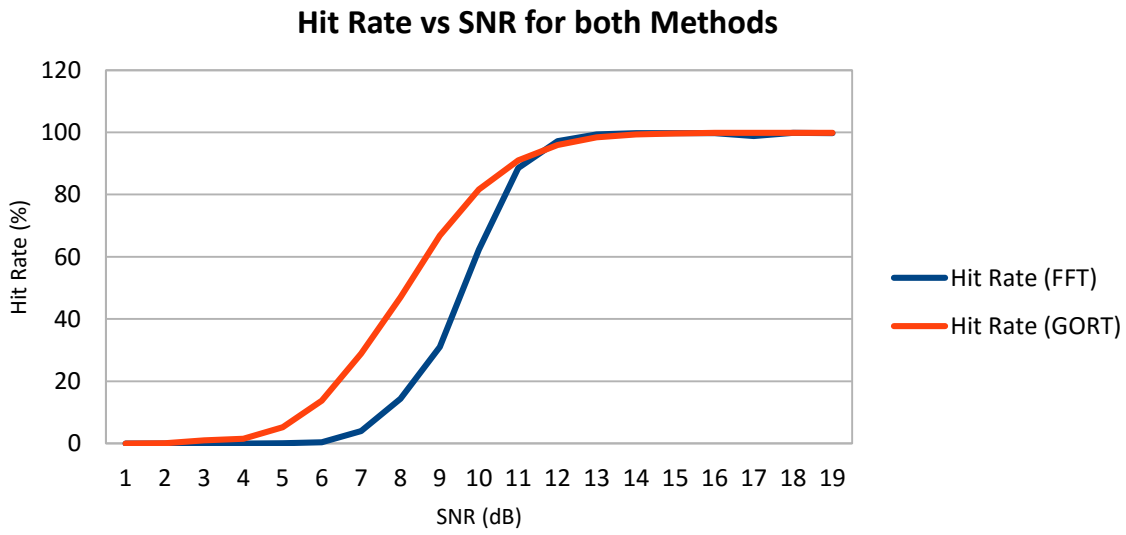

Illustration 9: Plot of the Success rate vs SNR

During the project I had expected the Goertzel's Algorithm approach to be much faster than the FFT approach based on the fact that despite Goertzel's Algorithm having a higher complexity, it should still have been faster for a handle number of frequencies compared to the FFT approach. I ran some tests to test this hypothesis and got interesting, unexpected results. I ran the decoder on sets of data ranging from 10,000 files to 100,000 files and timed the time it took for both types of decoders to decode the files. I plotted the time taken vs the number of files:

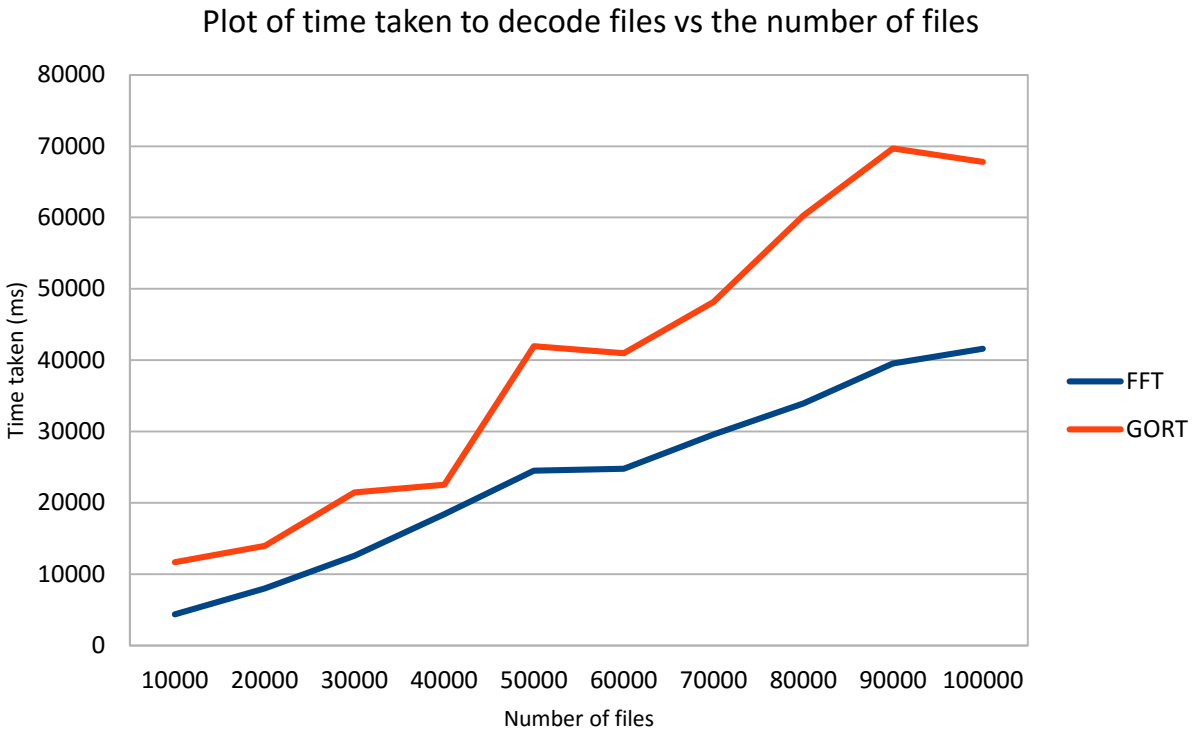

Illustration 9: Plot of the time taken to decode the files vs number of files decoded

From the plot it can be deduced that my previous assumption was incorrect. In fact, the FFT approach proved to be much faster than the Goertzel's Algorithm. A simple explanation of this is that although, theoretically, the Goertzel's algorithm should have been faster than the FFT, the function I used (from Apache Commons Math library) to perform the FFTs was fully optimized but the Goertzel's Class I created had no optimization at all hence it performed badly vs the FFT.

Conclusion

I had only 6 weeks to design, implement and test this DTMF decoder. The main goal was reached but there is still more work to do if there was more time. The project, together with all the source code is on github under the MIT License. I created a Java API for the DTMF-Decoder and it has the following specifications:

DTMF-Decoder API Specifications

- DTMF Decoder for .wav and .mp3 files or when given an array of sample points.
- Has an audio file interface which can be implemented for more audio file types (ogg, wma, etc...)
- DTMF Tone/Sequence Generator that can export to .wav files.
- Goertzel Class which can be used independently with arrays of sample points representing a signal.
- The API includes a GUI App which is a DTMF Decoder/Generator.

Possible Improvements

- Optimising the Goertzel Class to improve on speed and performance.
- Coming up with a more efficient way to detect noise and human speech to improve rejection and minimise false hits when decoding random noise files.
- Decoder could give location of detected tones within the audio file.

References

1) Dodd, A. (2002). *The essential guide to telecommunications*. Upper Saddle River, NJ: Prentice Hall PTR.*2) (picture) http://www.engineersgarage.com/tutorials/dtmf-dual-tone-multiple-frequency*

3) G. L. Smith, Dual-Tone Multi-frequency Receiver Using the WE DSP16 Digital Signal Processor, AT&T Application Note.

4) 2010 4th International Symposium on Communications, Control and Signal Processing (ISCCSP 2010) p1-5

5) Alan Oppenheim. 6.341 Discrete-Time Signal Processing, Fall 2005. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu (Accessed 26 Jan, 2016). License: Creative Commons BY-NC-SA

6) 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.

7) Chen, Chiouguey J. (June 1996), Modified Goertzel Algorithm in DTMF Detection Using the TMS320C80 DSP (PDF), Application Report, SPRA066, Texas Instruments

8) Heideman, Michael T.; Johnson, Don H.; Burrus, C. Sidney (1985-09-01). "Gauss and the history of the fast Fourier transform". Archive for History of Exact Sciences 34 (3): 265–277. doi:10.1007/BF00348431. ISSN 0003-9519.

9) Cimss.ssec.wisc.edu, (2016). *What is Matlab*. [online] Available at: http://cimss.ssec.wisc.edu/wxwise/class/aos340/spr00/whatismatlab.htm [Accessed 4 Dec. 2015].

10) ITU-T Recommendation Q.23 - Technical Features of Push-Button Telephone Sets. (1988). 1st ed. [ebook] INTERNATIONAL TELECOMMUNICATION UNION. Available at: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-Q.23-198811-I!!PDF-E&type=items [Accessed 9 Dec. 2015].

11) ITU-T Recommendations Q.24 - Multifrequency Push-Button Reception. (1988). 1st ed. [ebook] INTERNATIONAL TELECOMMUNICATION UNION. Available at: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-Q.24-198811-I!!PDF-E&type=items [Accessed 7 Dec. 2015].