Implementing the Goertzel algorithm in C

1st Nicolas Powell
Faculty of Engineering
University of Bristol
Bristol, United Kingdom
tc21714@bristol.ac.uk

2nd James jones
Faculty of Engineering
University of Bristol
Bristol, United Kingdom
gw21491@bristol.ac.uk

***Abstract*—This report shows the implementation of the Goertzel algorithm on a dual tone multi-frequency (DTMF) signal. Firstly we implement the Goertzel algorithm on a sample with a single frequency in order to gain a greater understanding of the algorithm. We then move onto decoding and detecting multiple frequencies from an encrypted binary file using the Goertzel algorithm. Finally, we take the decoded tone and pass the frequencies into sine waves to be converted into a waveform audio file.**

***Index Terms*—Goertzel, algorithm, DTMF, signal, frequency, sampling rate**

I. Introduction

A. DTMF

DTMF stands for Dual-Tone Multi-Frequency and is a signaling technique used in telecommunication systems to transfer data over phone lines. It is used in applications such as touch-tone dialing where two frequencies combine to create another. Various methods can be used to decode these tones such as the Fourier transform, band pass filter or the Goertzel algorithm.

B. Goertzel algorithm

The Goertzel algorithm is used as a more computationally efficient method of isolating tones as opposed to the Fourier transform or band pass filter. This efficiency is due to it only needing to calculate the Fourier transform at a specific frequency. The algorithm uses its coefficients to calculate a value known as the Goertzel value. The idea being, the higher the Goertzel value, the corresponding frequency is more prominent. The two coefficients and their frequencies corresponding to the highest Goertzel values are then collected to find the tone for the DTMF. When implemented, a feedback loop is required to be completed 205 times, each iteration generating samples to be entered into a feedforward loop. Equations 1 and 2 are also needed for implementation

$$Q_n = x(n) + 2cos(\frac{2\pi K}{N})Q_{n-1} - Q_{n-2} \tag{1}$$

$$|y_K(N)| = Q^2(N) + Q^2(n-1) - 2cos(\frac{2\pi K}{N})Q(N)Q(N-1) \tag{2}$$

```
prod1 = (delay_1*coef_1)>>14;
delay = input + (short)prod1 - delay_2;
delay_2 = delay_1;
delay_1 = delay;
N++;
```

Fig. 1. Feedback loop

```
if (N==206)
{
prod1 = (delay_1 * delay_1);
prod2 = (delay_2 * delay_2);
prod3 = (delay_1 * coef_1)>>14;
prod3 = prod3 * delay_2;
Goertzel_Value = (prod1 + prod2 - prod3) >> 15;
Goertzel_Value <<= 8;
N = 0;
delay_1 = delay_2 = 0;
}
gtz_out[0] = Goertzel_Value;
```

Fig. 2. Feed forward

II. Detecting a single frequency

When detecting a single frequency an implementation of the feedback loop is needed as shown in Figure 1. The feedback loop needs to be repeated 205 times to create enough samples for a reproducible goertzel output. This is ensured by the value N, increasing by 1 each time the function is called, and a if statement stopping at the value of 205. The delay values are all then affected by each new input from the sin wave with its specific frequency. Where delay is N, delay 1 is N-1 and delay 2 is N-2.

When the feedback loop is complete, the feedforward loop begins to generate a Goertzel value. This is implemented as shown in Figure 2. Then similar to Equation 2, delay1 and 2 are both squared. Delay 1 is squared to capture the amplitude and phase of the signal and delay 2 mimics this, however, is only used to increase accuracy. Prod 3 then represents the second part of the equation where both the delay and delay 1 are multiplied by the coefficient. During the first multiplication its also scaled down to prevent anymore overflow. The Goertzel output is also scaled up for a large difference between frequency detected and undetected. An example of the outputs achieved are demonstrated in Figure 3. Where the Goertzel value is 0 when the frequency is undetected, and a high value

('1024') when the frequency is present. This algorithm is also sensitive to nearby frequencies, which results in lower outputs surrounding the corresponding frequency to coefficient rather than an immediate cut off at these close frequencies.

```
The GTZ is 1024        The GTZ is 0
I am leaving Task 1,   I am leaving Task 1,

The GTZ is 1024        The GTZ is 0
I am leaving Task 1,   I am leaving Task 1,

The GTZ is 1024        The GTZ is 0
I am leaving Task 1,   I am leaving Task 1,
```

Fig. 3. Single frequency output with and without correct frequency

III. Detecting multiple tones in an audio file

For the next task a loop is needed to apply all the coefficients in the algorithm at once. Without this only a single frequency could be isolated for each part of the tone.

A. Change to the algorithm implementation

```
for (i = 0; i < 8; i++){
    prod1 = (delay_1[i]*coef[i])>>14;
    delay[i] = input + (short)prod1 - delay_2[i];
    delay_2[i] = delay_1[i];
    delay 1[i] = delay[i];
```

Fig. 4. Feedback loop for all frequencies

As shown in Figure 4 the coefficients are looped through. To avoid change in delay values whilst the function is being called, each delay value is also stored in an array. This allows the same delay values to be called in the feedforward loop, therefore improving accuracy. Similar to the feedback loop,

```
if (N == 206) {
    //Record start time
    start = Timestamp_get32();


    for (i = 0; i < 8; i++){
        prod1 = (delay_1[i] * delay_1[i]);
        prod2 = (delay_2[i] * delay_2[i]);
        prod3 = (delay_1[i] * coef[i])>>14;
        prod3 = prod3 * delay_2[i];
        Goertzel_Value = (prod1 + prod2 - prod3);
        Goertzel_Value <<=8;
        delay_1[i] = delay_2[i] = 0;
        gtz_out[i] = Goertzel_Value;
    }
```

Fig. 5. Feed forward for all frequencies

all the delay values are called when N reaches 206. Two separate loops are used here because when combined, N would increase by the size of the loop, and the delay values would be inconsistent. Each Goertzel value is then stored in a 'gtzout' array for 8 Goertzel values of a single digit. This is all then looped in the 'util' file to get each Goertzel value for all 8 digits in the tone.

B. Creating a Goertzel cut off

With the scaling in this code, the output value is above 300000 if the frequency is present and below 100000 if not. With a cut off set, anything below this value is then ignored whilst the two remaining frequencies make a single key on the keypad.

```
for (i = 0; i < 8; i++) {
    int gtz_cutoff = 200000;
for (h = 0; h < 4; h++) {
    if (gtz_out[h] > gtz_cutoff){
        for (k = 0; k < 4; k++) {
            if (gtz_out[k+4] > gtz_cutoff){
                result[n] = pad[h][k];}
        }}}
```

Fig. 6. Picking Keys based off Goertzel value

As shown in Figure 6, 3 loops are used. The first loop iterates through each Goertzel value found. Then the second loop goes through each row of the character pad and the fourth through each character in the row. This works by searching rows then columns of the character pad. Searching for the first present frequency then the second and making the result equal to the key found. As mentioned previously, Figure 6 is

```
The corresponding key is #
The corresponding key is B
The corresponding key is 4
The corresponding key is 6
The corresponding key is D
The corresponding key is 1
The corresponding key is #
The corresponding key is 1
```

Fig. 7. Final result from data file

then looped eight times for each of the results from the 8 digit tone given in the data file. The result of these are demonstrated in Figure 7. All of these values are then stored in the result array from the first loop.

C. Writing the detected tones to an audio file

With the corresponding keypad digits stored in the results array, a switch statement inside of a for loop can be used to assign two frequency values to each of the 8 digits, as seen in Figure 8. If a character is unrecognised, both of the frequencies will revert to default at 0Hz. The switch statement is looped through eight times, each time receiving the next digit from the results array. For example, the first digit in the results array is '#' which assigns 'freq1' to 941Hz and 'freq2' to 1477Hz which is later used in the generation of the sine waves.

```
for(z=0;z<n_tones;z++) {
    digit = keys[z];
     int freq1, freq2;
     switch (digit) {
         case '1': freq1 = 697; freq2 = 1209; break;
         case '2': freq1 = 697; freq2 = 1336; break;
         case '3': freq1 = 697; freq2 = 1477; break;
         case '4': freq1 = 770; freq2 = 1209; break;
         case '5': freq1 = 770; freq2 = 1336; break;
         case '6': freq1 = 770; freq2 = 1477; break;
         case '7': freq1 = 852; freq2 = 1209; break;
         case '8': freq1 = 852; freq2 = 1336; break;
         case '9': freq1 = 852; freq2 = 1477; break;
         case '*': freq1 = 941; freq2 = 1209; break;
         case '0': freq1 = 941; freq2 = 1336; break;
         case '#': freq1 = 941; freq2 = 1477; break;
         case 'A': freq1 = 697; freq2 = 1633; break;
         case 'B': freq1 = 770; freq2 = 1633; break;
         case 'C': freq1 = 852; freq2 = 1633; break;
         case 'D': freq1 = 941; freq2 = 1633; break;
         default: freq1 = 0; freq2 = 0; break;
     }
```

Fig. 8. Keypad to frequency conversions

With each digit assigned its two frequencies, the next step is to transform the frequencies into sine waves and combine them in order to create each tone. As seen in Figure 9, a for

```
for (i = 0; i < samples_per_tone; i++) {
    t = (float)i / fs;
    sample = (short) 16384*sin(2.0*PI*freq1*t) + 16384*sin(2.0*PI*freq2*t);
    buffer[i + z * samples_per_tone] = sample;
}
```

Fig. 9. Sine wave generation

loop is used to generate the series of samples for the sine wave, which are stored in the buffer array. The variable 't' is equivalent to the 'TICKPERIOD', where tick period is 1/8000. 't' is then defined using the preset value of fs ,10000, and 'i' from the for loop, resulting in 0,1/10000 to 7/10000. This is equivalent to 'TICKPERIOD' multiplied by 'tick' just using a different method. The variable 't' then replaces 'Tick' and 'TICKPERIOD' given previously in the variable sample. The sample is also converted to a short with 16384 as the value multiplying the two sin waves for a clearer signal. 8 samples are then generated for each tone and stored in the buffer array.

With all eight digits transformed into full tone waveform, and stored in the buffer array, they are ready to be written to the 'answer.wav' file where the sequence of keypad tones can be listened to.