Goertzel Implementation Report

Leonardo Coppi
Dept. of Electrical and Electronic Eng.
University of Bristol
Bristol, United Kingdom
leonardo.coppi.2021@bristol.ac.uk

Joe Taylor
Dept. of Electrical and Electronic Eng.
University of Bristol
Bristol, United Kingdom
he21722@bristol.ac.uk

Carl Blastique
Dept. of Electrical and Electronic Eng.
University of Bristol
Bristol, United Kingdom
ov21591@bristol.ac.uk

***Abstract*—This technical report will focus on the implementation of the Goertzel Algorithm in C language using a Dual-Tone multi-frequency (DTMF) signal to detect multiple frequencies and gather results from this implementation**

***Index Terms*—component, formatting, style, styling, insert**

I. Single Goertzel Frequency Detection

A. Goetzel Algorithm Theory

This section of the technical report focuses on the detection of a single frequency using the Goertzel algorithm. Exploring the Goertzel algorithm in more detail, it is used as a digital signal processing technique to calculate the Discrete Fourier transform of a specified frequency , which in our case 697Hz. Below, labelled Equation 1, relates to the detection of the presence of a specific tone where N in our case is set to $N = 206$, and $\backslash Q(N)$" and $\backslash Q(N-1)$" relate to the delays shown in the block diagram of the system labelled Figure 1

$$|Y_k(N)^2| = Q^2(N) + Q^2(N-1) - (coeff)Q(N)Q(N-1) \tag{1}$$

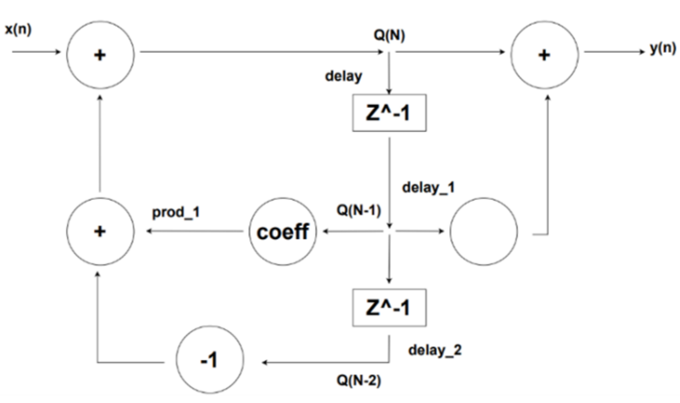

Fig. 1. Block level diagram representing how the Goertzel Algorithm is derived

The way this diagram performs analysis on the signal is taking samples $x(n)N$ times. As $N = 206$, for all samples $N < 206$, the feedback loop is used at fixed intervals given as T the sampling interval, which happens to be the inverse of the sampling rate $\frac{1}{f_s}$. For the 206th sample, $N = 206$, the feedforward loop is used to compute the final value.

The variable labelled "coeff" that appears on both Figure 1 and Equation 1, is used as a constant which determines and sets the value of the frequency response of the digital signal processor (DSP). This frequency response is then used to calculate the Discrete Fourier Transform of the specific frequency we are trying to find.

$$coeff = 2\cos\frac{2\pi k}{N} \tag{2}$$

where:

$$k = \frac{NF_{tone}}{f_s} \tag{3}$$

and where, f_tone is the frequency of the tone and f_s is the sampling frequency. In our case when detecting a frequency of $697Hz$ we have a k value of 18 and coeff equal to exactly 1.703275.

B. Dual-Tone Multi-Frequency systems

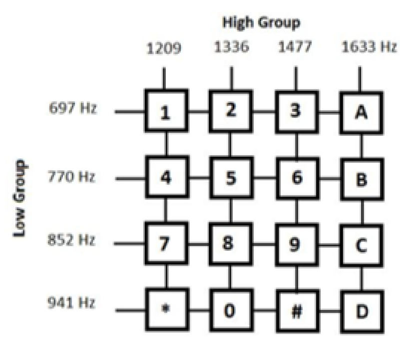

Fig. 2. Common frequencies used for Dual-Tone Multi-Frequency systems [1]

Above, labelled as Figure.1, is the number pad for DTMF signals and if we look specifically at values `1`, `2`, `3` and `A`, these should all be the DTMF signals that should give a high Goertzal output as they are all in the row of $697Hz$. Another example can be, if the user desired to select Digit `9` the two frequencies $1477Hz$ and $852Hz$ would need to have a high Goertzel value.

C. Code discussion

in the code we use the Q15/hexadecimal format, $0x6D02$. When this frequency is detected a high Goertzal value will be printed into the console, however for the other case of this frequency not being detected by our system, the output to the terminal should display a value of 0.

Looking back to Figure 1, we have the final variable `prod_1` which is simply just the coefficient multiplied by the first delay given as `delay_1` where the coefficient is again, the filters frequency response and the delay is the time delay between the input signal and the filters output.

```c
81   *  ====================
82   */
83  void clk_SWI_GTZ_0697Hz(UArg arg0)
84  {
85      /* */
86      static int iteration_num = 0;
87      static int Goertzel_Output = 0;
88  
89      static short Q;
90      static short Q1 = 0;
91      static short Q2 = 0;
92  
93      int maths1, maths2, maths3;
94  
95      short input;
96      short coef_1 = 0x6D02; //hex for 697
97  
98      input = (short) sample >> 12;
99  
100     // first part
101     maths1 = (Q1*coef_1)>>14;
102     Q = input + (short)maths1 - Q2;
103     //
104 
105     if (iteration_num==206)
106     {
107         maths1 = (Q * Q);
108         maths2 = (Q1 * Q1);
109         maths3 = (Q * Q1 * coef_1) >> 14;
110 
111         Goertzel_Output = (maths1 + maths2 - maths3);// >> 15;
112         Goertzel_Output <<= 1;  // scale up for sensitivity
113         iteration_num = 0;
114         Q = Q1 = Q2 = 0;
115     }
116 
117     //update delayed variables
```

Fig. 3. Snippet of the code composed, consisting of the Goertzel Algorithm implemented to detect a singular frequency signal

Looking at Figure 3, specifically line 96, we define the frequency of interest we are trying to detect by asserting, in hexadecimal format $697kHz$ to the variable `coef_1`. Line 101-102 of code describes the feedback loop on Figure 2 where we are just using the block diagram to derive the variable for "`Q` and `maths1`.

Lines 107-117, implement the use of Equation 1, where $Q^2(N)$ in the equation is `Q` $\times$ `Q` on line 109 which creates `maths1`. This is the same for the other variables apart from `maths3` where we scale it down by 14, to make it fit into the `short` data type. Finally, to complete Equation 1 and find the Goertzel value, we add `maths1` and `maths2` together and take the difference of `maths3` away from this value. Once scaled up by 1 for sensitivity, the variable `gtz_out[0]` stores the final Goertzel value in an array.

Below, labelled as Figure 3 is the results of our implementation, showing the Goertzel values to be high meaning the specified frequency we are trying to detect, $697Hz$, has been detected. It is also important to note, when observing the results in the console, that the higher Goertzel value the closer the measured frequency is to the frequency associated to the coefficient.

II. Multi-Goertzel Frequency Detection

The code from figure generates Goertzal values for the 8 predetermined frequencies, utilizing 8 parrallel ISRs. The output, which represents the strength of a specific frequency component in the sample being analyze is then stored in the array. `gtz_out`

```c
/* TODO 1. Complete the feedback loop of the GTZ algorithm*/
/* ========================= */
int n;
// Create array of previous values. Set to static so they are constant throughout iterations. Initialized to zero.
static int maths1[8], maths2[8], maths3[8] = {0};           // need to be int to hold large products
static short Q[8], Q1[8], Q2[8] = {0};  // can be short to conserve memory due to small variable size

for (n = 0; n < 8; n++) {
    maths1[n] = (Q1[n]*coef[n])>>14;
    Q[n] = input + (short)maths1[n] - Q2[n];

    //update delayed variables
    Q2[n] = Q1[n];
    Q1[n] = Q[n];
}
/* ========================= */
N++;

//Record stop time
stop = Timestamp_get32();
//Record elapsed time
tdiff = stop-start;

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

    /* TODO 2. Complete the feedforward loop of the GTZ algorithm*/
    /* ========================= */
    for (n = 0; n < 8; n++) {

        //Same as GTZ_one_freq, just all variables are now arrays for all 8 iterations
        maths1[n] = (Q[n] * Q[n]);
        maths2[n] = (Q1[n] * Q1[n]);
        maths3[n] = (Q[n] * Q1[n] * coef[n]) >> 14;

        Goertzel_Value[n] = (maths1[n] + maths2[n] - maths3[n]) >> 8;

        gtz_out[n] = Goertzel_Value[n];
    }
    //reset all!
    int k;
    for(k=0; k<8; k++) {
        maths1[k] = 0;
        maths2[k] = 0;
        maths3[k] = 0;
        Q[k] = 0;
        Q1[k] = 0;
        Q2[k] = 0;
    }

    /* ========================= */
    flag = 1;
    N = 0;
```

Fig. 4. Snippet of the code composed, consisting of the Goertzel Algorithm implemented to detect multiple frequencies.

The tones that are present within the sample are found by checking the output for the highest Goertzel value. This helps to avoid errors that could occur when multiple frequencies are present corresponding to different digits. To represent a digit on the keypad this must be done for all frequencies in the low row and high column. If a frequency is not present from either the high column or low row then no digit can be represented. The code in Figure 4 loops through the values produced by the Goertzal algorithm for all frequencies and updates the value of `max_val` if the presence of a frequency is higher than the one already stored. in line 65 the index of the row and collum is then assigned to `results[]` which can be used to find the corresponding digit that the frequency represents in Figure 2.

The frequencies contained in the data file change every 0.210 seconds meaning the Goertzel Algorithm must be reevaluated with the same period. This happens 8 times meaning a total of 8 digits are represented. After running a debug of the code in Figure 5 The digit represented by the tone and the Goertzal value for each tone is shown in the terminal of CCS

```c
    for(n=0;n<8;n++) {
        while (!flag) Task_sleep(210);
        /* TODO 3. Complete code to detect the 8 digits based on the GTZ output */
        /* ========================== */
        //printf("%s %d \n", "Digit: ", n);
        int row, col;
        row = col = 0;
        int max_val = 0;
        max_val = 0;

        // For each row, check the gtz value and find the one with the highest value
        for (i = 0; i < 4; i++) {
            //printf("%d | %d \n", i, gtz_out[i]);
            if (gtz_out[i] > max_val) {
                max_val = gtz_out[i];
                row = i;
            }
        }

        max_val = 0; // Clear last_max for column

        // For each column, check the gtz value and find the one with the highest value
        for (i = 4; i < 8; i++) {
            //printf("%d | %d \n", i, gtz_out[i]);
            if (gtz_out[i] > max_val) {
                max_val = gtz_out[i];
                col = i-4;
            }
        }

        //printf("%s %d %d \n", "Final result (row) (col) : ", row, col);
        result[n] = pad[row][col];
        /* ========================== */
        printf("%c\n", result[n]);
        flag = 0;
    }
    printf("\nDetection finished\n");
    printf("Generating audio\n");
    task2_dtmfGenerate(result);
    printf("Finished\n");
}
```

Fig. 5. Code snippet calculating the maximum Goertzel value for high and low frequencies which corresponds to a digit on a keypad.

```
Digit:  0
0 | 83043
1 | 2632171
2 | 30161
3 | 6822
4 | 132373
5 | 15
6 | 0
7 | 10
Final result (row) (col) :  1 0
4
Digit:  1
0 | 2823172
1 | 8800
2 | 1381
3 | 124
4 | 12
5 | 19
6 | 130
7 | 330419
Final result (row) (col) :  0 3
A
Digit:  2
0 | 3
1 | 24
2 | 91
3 | 189287
4 | 137375
5 | 9
6 | 11
7 | 10
Final result (row) (col) :  3 0
*

Digit:  3
0 | 1140
1 | 1256
2 | 2563
3 | 219604
4 | 6773
5 | 1344175
6 | 5296
7 | 913
Final result (row) (col) :  3 1
0
Digit:  4
0 | 3
1 | 24
2 | 91
3 | 189287
4 | 137375
5 | 9
6 | 11
7 | 10
Final result (row) (col) :  3 0
*
Digit:  5
0 | 97404
1 | 2550573
2 | 20988
3 | 2617
4 | 5340
5 | 1343464
6 | 4821
7 | 558
Final result (row) (col) :  1 1
5

Digit:  6
0 | 17
1 | 47
2 | 139
3 | 193419
4 | 39
5 | 85
6 | 108766
7 | 138
Final result (row) (col) :  3 2
#
Digit:  7
0 | 85396
1 | 2625901
2 | 28857
3 | 6734
4 | 134
5 | 84
6 | 194
7 | 329176
Final result (row) (col) :  1 3
B
```

Fig. 6. Debug Output in terminal after running code in Figure 5

III. Generating audio file from decoded tone

This section will discuss Task 2c, where we complete the last TODO section to generate an audio file based on the decoded tone

With the detected keypresses, they can be re-encoded back into pairs of DTMF frequencies. In the given template a `.wav` file is created with specific headers. These define a header size of 16 bits, and a bitrate of 10kHz.

The audio format in the WAV file is uncompressed in the linear pulse-code modulation (LPCM) format, and stores the waveform's amplitude at the sample rate (10kHz). Due to it's 16-bit format, the input to the buffer must be no larger than a short signed interger type. Consequently, since the amplitude is the sum of two sine waves, each sine wave magnitude cannot be larger than half of a short int type's max value - $32767/2$ - to ensure that even if both sine waves result in ± 1 the max absolute value doesn't exceed 32767.

On Figure 7, the digits are assigned to their corresponding frequencies. For example, lines 93-94, if digits `A`, `B`, `C` or `D` are selected the corresponding frequency of $1633Hz$ is assigned; the same case goes for the rest of the digits assigned below.

```c
87          /* TODO 4. Complete the DTMF algorithm to generate audio signal based on the digits */
88          /* ========================== */
89
90          /* get frequencies from digit */
91          int freq1, freq2;
92
93          if (digit == 'A' || digit == 'B' || digit == 'C' || digit == 'D') {
94              freq2 = 1633;
95          } else if (digit == '3' || digit == '6' || digit == '9' || digit == '#') {
96              freq2 = 1477;
97          } else if (digit == '2' || digit == '5' || digit == '8' || digit == '0') {
98              freq2 = 1336;
99          } else if (digit == '1' || digit == '4' || digit == '7' || digit == '*') {
100             freq2 = 1209;
101         }
102
103         if (digit == '1' || digit == '2' || digit == '3' || digit == 'A') {
104             freq1 = 697;
105         } else if (digit == '4' || digit == '5' || digit == '6' || digit == 'B') {
106             freq1 = 770;
107         } else if (digit == '7' || digit == '8' || digit == '9' || digit == 'C') {
108             freq1 = 852;
109         } else if (digit == '*' || digit == '0' || digit == '#' || digit == 'D') {
110             freq1 = 941;
111         }
112
113         /*printf("Freq1: %d Freq 2: %d \n", freq1, freq2);*/
114
115         /* produce sin waves with frequencies */
116         float output;
117
118         for (i = 0; i < samples_per_tone; i++) {
119             /* (float) has to be added because i and fs are both ints */
120             float low_freq = sin(2*PI*freq1*((float) i/fs));
121             float high_freq = sin(2*PI *freq2*((float) i/fs));
122
123             /* without the sets the magnitude to remain within a (short), since that is the headersize we're using*/
124             output = (short) (32767/2)*(low_freq + high_freq);
125
126         /*Put it in buffer*/
127             buffer[(samples_per_tone*n) + i] = output;
128         }
129
130         /* ========================= */
```

Fig. 7. Code snippet used in generating WAV file

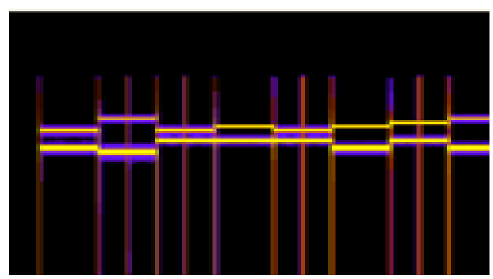

Fig. 8. Spectral waveform of generated audio file on a logarithmic scale

References

[1] DTMF Controlled Home Automation System. (n.d.). Engineers Garage. https://www.engineersgarage.com/dtmf-controlled-home-automation-system-using-8951/