The Goertzel Algorithm

Kevin Banks - August 28, 2002

The Goertzel Algorithm

The Goertzel algorithm can perform tone detection using much less CPU horsepower than the Fast Fourier Transform, but many engineers have never heard of it. This article attempts to change that.

Most engineers are familiar with the Fast Fourier Transform (FFT) and would have little trouble using a "canned" FFT routine to detect one or more tones in an audio signal. What many don't know, however, is that if you only need to detect a few frequencies, a much faster method is available. It's called the Goertzel algorithm.

Tone detection

Many applications require tone detection, such as:

- DTMF (touch tone) decoding
- Call progress (dial tone, busy, and so on) decoding
- Frequency response measurements (sending a tone while simultaneously reading back the result)-if you do this for a range of frequencies, the resulting frequency response curve can be informative. For example, the frequency response curve of a telephone line tells you if any load coils (inductors) are present on that line.

Although dedicated ICs exist for the applications above, implementing these functions in software costs less. Unfortunately, many embedded systems don't have the horsepower to perform continuous real-time FFTs. That's where the Goertzel algorithm comes in.

In this article, I describe what I call a basic Goertzel and an optimized Goertzel.

The basic Goertzel gives you real and imaginary frequency components as a regular Discrete Fourier Transform (DFT) or FFT would. If you need them, magnitude and phase can then be computed from the real/imaginary pair.

The optimized Goertzel is even faster (and simpler) than the basic Goertzel, but doesn't give you the real and imaginary frequency components. Instead, it gives you the relative magnitude squared. You can take the square root of this result to get the relative magnitude (if needed), but there's no way to obtain the phase.

In this short article, I won't try to explain the theoretical background of the algorithm. I do give some links at the end where you can find more detailed explanations. I can tell you that the algorithm works well, having used it in all of the tone detection applications previously listed (and others).

A basic Goertzel

First a quick overview of the algorithm: some intermediate processing is done in every sample. The actual tone detection occurs every Nth sample. (I'll talk more about N in a minute.)

As with the FFT, you work with blocks of samples. However, that doesn't mean you have to process the data in blocks. The numerical processing is short enough to be done in the very interrupt service routine (ISR) that is gathering the samples (if you're getting an interrupt per sample). Or, if you're getting buffers of samples, you can go ahead and process them a batch at a time.

Before you can do the actual Goertzel, you must do some preliminary calculations:

1. Decide on the sampling rate.
2. Choose the block size, **N**.
3. Precompute one cosine and one sine term.
4. Precompute one coefficient.

These can all be precomputed once and then hardcoded in your program, saving RAM and ROM space; or you can compute them on-the-fly.

Sampling rate

Your sampling rate may already be determined by the application. For example, in telecom applications, it's common to use a sampling rate of 8kHz (8,000 samples per second). Alternatively, your analog-to-digital converter (or CODEC) may be running from an external clock or crystal over which you have no control.

If you can choose the sampling rate, the usual Nyquist rules apply: the sampling rate will have to be at least twice your highest frequency of interest. I say "at least" because if you are detecting multiple frequencies, it's possible that an even higher sampling frequency will give better results. What you really want is for every frequency of interest to be an integer factor of the sampling rate.

Block size

Goertzel block size **N** is like the number of points in an equivalent FFT. It controls the frequency resolution (also called bin width). For example, if your sampling rate is 8kHz and **N** is 100 samples, then your bin width is 80Hz.

This would steer you towards making **N** as high as possible, to get the highest frequency resolution. The catch is that the higher **N** gets, the longer it takes to detect each tone, simply because you have to wait longer for all the samples to come in. For example, at 8kHz sampling, it will take 100ms for 800 samples to be accumulated. If you're trying to detect tones of short duration, you will have to use compatible values of **N**.

The third factor influencing your choice of **N** is the relationship between the sampling rate and the target frequencies. Ideally you want the frequencies to be centered in their respective bins. In other words, you want the target frequencies to be integer multiples of **sample_rate/N**.

The good news is that, unlike the FFT, **N** doesn't have to be a power of two.

Precomputed constants

Once you've selected your sampling rate and block size, it's a simple five-step process to compute the constants you'll need during processing:

$$k = (\text{int})\left(0.5 + \frac{N * \text{target\_freq}}{\text{sample\_rate}}\right)$$

$$w = (2*\pi/N)*k$$
$$\text{cosine} = \cos w$$
$$\text{sine} = \sin w$$
$$\text{coeff} = 2 * \text{cosine}$$

For the per-sample processing you're going to need three variables. Let's call them Q_0, Q_1, and Q_2.

Q_1 is just the value of Q_0 *last time.* Q_2 is just the value of Q_0 *two times ago* (or Q_1 last time).

Q_1 and Q_2 must be initialized to zero at the beginning of each block of samples. For every *sample,* you need to run the following three equations:

$$Q_0 = coeff * Q_1 - Q_2 + sample$$
$$Q_2 = Q_1$$
$$Q_1 = Q_0$$

After running the per-sample equations N times, it's time to see if the tone is present or not.

$$real = (Q_1 - Q_2 * cosine)$$
$$imag = (Q_2 * sine)$$
$$magnitude^2 = real^2 + imag^2$$

A simple threshold test of the magnitude will tell you if the tone was present or not. Reset Q_2 and Q_1 to zero and start the next block.

An optimized Goertzel

The optimized Goertzel requires less computation than the basic one, at the expense of phase information.

The per-sample processing is the same, but the end of block processing is different. Instead of computing real and imaginary components, and then converting those into relative magnitude squared, you directly compute the following:

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

This is the form of Goertzel I've used most often, and it was the first one I learned about.

Pulling it all together

Listing 1 shows a short demo program I wrote to enable you to test-drive the algorithm. The code was written and tested using the freely available DJGPP C/C++ compiler. You can modify the **#defines** near the top of the file to try out different values of ***N,*** **sampling_rate**, and **target_frequency**.

The program does two demonstrations. In the first one, both forms of the Goertzel algorithm are used to compute relative magnitude squared and relative magnitude for three different synthesized signals: one below the **target_frequency**, one equal to the **target_frequency**, and one above the **target_frequency**.

You'll notice that the results are nearly identical, regardless of which form of the Goertzel algorithm is used. You'll also notice that the detector values peak near the target frequency.

In the second demonstration, a simulated frequency sweep is run, and the results of just the basic Goertzel are shown. Again, you'll notice a clear peak in the detector output near the target frequency. Figure 1 shows the output of the code in Listing 1.

Figure 1: Demo output

```
For SAMPLING_RATE = 8000.000000 N = 205 and FREQUENCY = 941.000000, k = 24 and coeff = 1.482867


For test frequency 691.000000:
real = -360.392059 imag = -45.871609
Relative magnitude squared = 131986.640625
Relative magnitude = 363.299652
Relative magnitude squared = 131986.640625
Relative magnitude = 363.299652
```

For test frequency 941.000000:
real = -3727.528076 imag = -9286.238281
Relative magnitude squared = 100128688.000000
Relative magnitude = 10006.432617
Relative magnitude squared = 100128680.000000
Relative magnitude = 10006.431641

For test frequency 1191.000000:
real = 424.038116 imag = -346.308716
Relative magnitude squared = 299738.062500
Relative magnitude = 547.483398
Relative magnitude squared = 299738.062500
Relative magnitude = 547.483398

Freq=	641.0	rel mag^2=	146697.87500	rel mag=	383.01160
Freq=	656.0	rel mag^2=	63684.62109	rel mag=	252.35812
Freq=	671.0	rel mag^2=	96753.92188	rel mag=	311.05292
Freq=	686.0	rel mag^2=	166669.90625	rel mag=	408.25226
Freq=	701.0	rel mag^2=	5414.02002	rel mag=	73.58002
Freq=	716.0	rel mag^2=	258318.37500	rel mag=	508.25031
Freq=	731.0	rel mag^2=	178329.68750	rel mag=	422.29099
Freq=	746.0	rel mag^2=	71271.88281	rel mag=	266.96796
Freq=	761.0	rel mag^2=	437814.90625	rel mag=	661.67584
Freq=	776.0	rel mag^2=	81901.81250	rel mag=	286.18494
Freq=	791.0	rel mag^2=	468060.31250	rel mag=	684.14935
Freq=	806.0	rel mag^2=	623345.56250	rel mag=	789.52234
Freq=	821.0	rel mag^2=	18701.58984	rel mag=	136.75375
Freq=	836.0	rel mag^2=	1434181.62500	rel mag=	1197.57324
Freq=	851.0	rel mag^2=	694211.75000	rel mag=	833.19373
Freq=	866.0	rel mag^2=	1120359.50000	rel mag=	1058.47034
Freq=	881.0	rel mag^2=	4626623.00000	rel mag=	2150.95874
Freq=	896.0	rel mag^2=	160420.43750	rel mag=	400.52521
Freq=	911.0	rel mag^2=	19374364.00000	rel mag=	4401.63184
Freq=	926.0	rel mag^2=	81229848.00000	rel mag=	9012.76074
Freq=	941.0	rel mag^2=	100128688.00000	rel mag=	10006.43262
Freq=	956.0	rel mag^2=	43694608.00000	rel mag=	6610.18994
Freq=	971.0	rel mag^2=	1793435.75000	rel mag=	1339.19226
Freq=	986.0	rel mag^2=	3519388.50000	rel mag=	1876.00330
Freq=	1001.0	rel mag^2=	3318844.50000	rel mag=	1821.76965
Freq=	1016.0	rel mag^2=	27707.98828	rel mag=	166.45717
Freq=	1031.0	rel mag^2=	1749922.62500	rel mag=	1322.84644
Freq=	1046.0	rel mag^2=	478859.28125	rel mag=	691.99658
Freq=	1061.0	rel mag^2=	284255.81250	rel mag=	533.15643
Freq=	1076.0	rel mag^2=	898392.93750	rel mag=	947.83594
Freq=	1091.0	rel mag^2=	11303.36035	rel mag=	106.31726
Freq=	1106.0	rel mag^2=	420975.65625	rel mag=	648.82635
Freq=	1121.0	rel mag^2=	325753.78125	rel mag=	570.74841
Freq=	1136.0	rel mag^2=	36595.78906	rel mag=	191.30026
Freq=	1151.0	rel mag^2=	410926.06250	rel mag=	641.03516
Freq=	1166.0	rel mag^2=	45246.58594	rel mag=	212.71245
Freq=	1181.0	rel mag^2=	119967.59375	rel mag=	346.36337
Freq=	1196.0	rel mag^2=	250361.39062	rel mag=	500.36127

```
Freq=  1211.0 rel mag^2=      1758.44263 rel mag=     41.93379
Freq=  1226.0 rel mag^2=    190195.57812 rel mag=    436.11417
Freq=  1241.0 rel mag^2=     74192.23438 rel mag=    272.38251
```

To avoid false detections in production code, you will probably want to qualify the raw detector readings by using a debouncing technique, such as requiring multiple detections in a row before reporting a tone's presence to the user.

As you can see, the Goertzel algorithm deserves to be added to your signal processing toolbox.

Kevin Banks has been developing embedded systems for 19 years, as a consultant and as an employee at companies including SCI, TxPORT, DiscoveryCom, and Nokia. Currently he is back in consulting mode. He can be reached at kbanks@hiwaay.net.

References

Here are some links to other resources you may find useful:

www.numerix-dsp.com/goertzel.html

ptolemy.eecs.berkeley.edu/papers/96/dtmf_ict/www/node3.html

www.analogdevices.com/library/dspManuals/Using_ADSP-2100_Vol1_books.html

www.analogdevices.com/library/dspManuals/pdf/Volume1/Chapter_14.pdf

www.analogdevices.com/library/dspManuals/Using_ADSP-2100_Vol2_books.html

www.analogdevices.com/library/dspManuals/pdf/2100Chapter_8.pdf

www.delorie.com/djgpp/

Return to the September 2002 Table of Contents