Goertzel algorithm for tone detection on DSP

June 2004.

Contents

1 Introduction

Geortzel algorithm is well known. This description doesn't redo some of these more-or-less generic descriptions. It compiles a lot of information not available in books, articles, etc, to provide a complete description of the use of the Goertzel algorithm for tone recognition. In a sense, this description is a reminder of "what goes where and how" when you want to use the this algorithm.

Alternatives were not considered, there are too many of them. Provided were only the most important definitions and theoretical foundations necessary for understanding the rest of the description, which is something that can not be found in many places.

1.1 Some theoretical foundations

Here we present some theoretical foundation from the field of telecommunications and signal processing needed for understanding the main description.

The signal level in the "milliwatt" decibels (dBm) is calculated like:

$$P_{dBm} = 10\log(\frac{P}{P_{mW}}) \tag{1}$$

Where $P_{mW} = 1mW$, a P is the power in Watts. Logarithm with base 10 is implied. From this equation we get the equation for power, based on the level in decibels:

$$P = P_{mW} \cdot 10^{\frac{P_{dBm}}{10}} \tag{2}$$

Power depending on the voltage is calculated on the resistor of $R_{ref} = 600\Omega$:

$$P = \frac{V_{eff}^2}{R_{ref}} \tag{3}$$

From there we get the effective voltage:

$$V_{eff} = \sqrt{R_{ref}P} \tag{4}$$

and then the amplitude of the voltage:

$$V_m = V_{eff}\sqrt{2} = \sqrt{2R_{ref}P} \tag{5}$$

Numerically, we get a simple formula (SI units):

$$V_m = \sqrt{1200P} \tag{6}$$

From this we can get the level of the signal for A given amplitude:

$$P_{dBm} = 10\log\frac{V_{eff}^2}{R_{ref}P_{mW}} = 10\log\frac{V_m^2}{2R_{ref}P_{mW}} \tag{7}$$

Purely numericaly, this is:

$$P_{dBm} = 10\log\frac{V_m^2}{1,2} \tag{8}$$

1.2 A law

In telephony (European and similar) speech is transmitted packed according to the A law. The A law is designed to enable speech transfer without much distortion (it is supposed to be adjusted to the way the human ear receives sound) however there is some (the famous “you sound different over the phone”). In Japan and America the μ law is used, which is similar.

The A law does the packing/unpacking of the signed 12-bit data into 8 bits. The packed byte can be disassembled like this (one letter represents one bit):

$$\underbrace{S}_{znak}\underbrace{XXXX}_{mantisa}\underbrace{YYY}_{eksponent}$$

Practically, the A law is a sort of floating point number. Obviously, the A law “keeps” only 4 bits worth of useful data about the sample , but it keeps both the sign and the exponent. Decoding according to the A law can implement the following part of the C code:

```
uint16_t DecodeAlaw(uint8_t sample)
{
    uint16_t sign = (sample & 0x80) << 5;
    uint16_t mantisa = sample & 0x78;
    int expo = sample & 0x07;

    return sign | (mantisa << expo);
}
```

If you like to (ab)use the features of C to make this concise:

```
inline uint16_t DecodeAlaw(uint8_t by) {
    return ((by & 0x80) << 5) | ((by & 0x78) << (by & 0x07));
}
```

Since there are only 256 possibilities, this is usually stored in a table. Also, depending on what you need, the character can “expand to the end”, that is, the word we got (two bytes) is also signed. In Volts, the A law range is approximately $\pm 1,5V$. It would be pointless to give the full table of the A law, however the following might be interesting:

- In an unpacked form, one volt is approximately 2619, that is, 0xA3b.
- If the unpacked form is interpreted as a fixed point, of a range from -1 to 1 (that is often done in DSP-s), then 2619, or, one Volt, is approximately: 0.07992554, that is, the volt scaling factor/fixed point is 12.5144596.

Encoding is more difficult (among other things, we need to get 12 bits to encode), however it is practically never done in applications that are of interest to us. For generating tones a table is often made containing samples (8-bit) to be transmitted without processing.

1.3 Calculations in digital signal processing

The main "catch" in digital signal processing is doing the work with unfit tools, so that available resources would be put to optimal use and to enable processing of multiple channels at once. Namely, if we know the formulas and have a processor that can calculate in a floating point number of 80 bits (in C that is the `long double` type), then we just have to let it do its job, and, for all relevant use cases, there will be no mistakes, overflows, etc. Even 32 bits (in C - `float` type) is often enough and the more expensive DSP-s work with exactly that kind of data.

In practice, 16 bit fixed-points DSP-s are mostly used. In GVS particularly the MC56xxx family. These DSPs have 16-bit architecture, with two accumulators of 36/40 bits. Support for the fixed point number is in the range from -1 to 1 (more precisely from -1 to $1 - 2^{15}$) Practically, we should divide the integer contents of a register by 2^{15} and we will get the value in the range from -1 to 1. Here are some examples:

Hex	Binary	Decimal	Binary (w/point)	fixed point
7FFF	0111 1111 1111 1111.	32767	0.111 1111 1111 1111	0.99997
7000	0111 0000 0000 0000.	28672	0.111 0000 0000 0000	0.875
4000	0100 0000 0000 0000.	16384	0.100 0000 0000 0000	0.5
2000	0010 0000 0000 0000.	8192	0.010 0000 0000 0000	0.25
1000	0001 0000 0000 0000.	4096	0.001 0000 0000 0000	0.125
0000	0000 0000 0000 0000.	0	0.000 0000 0000 0000	0.0
C000	1100 0000 0000 0000.	-16384	1.100 0000 0000 0000	-0.5
E000	1110 0000 0000 0000.	-8192	1.110 0000 0000 0000	-0.25
F000	1111 0000 0000 0000.	-4096	1.111 0000 0000 0000	-0.125
9000	1001 0000 0000 0000.	-28672	1.001 0000 0000 0000	-0.875
8000	1000 0000 0000 0000.	-32768	1.000 0000 0000 0000	-1.0

Most algorithms use signed numbers. The sign occupies the highest bit. The numbers are introduced in a 2s complement. Also, the most often used are instructions in a fixed point. What does that mean? The differences are in multiplication and division. Division is rarely used in these kinds of algorithms, so let's skip it.

Since we are multiplying two 16-bit signed numbers, the result is a signed 31-bit number (in a general case, multiplication of two n-bit signed numbers gives us 2^{n-1}-bit signed number). The question is what to do with that number to extend it to 32-bits? The answer:

- If it's an integer, expand to 32 bits, keeping the sign
- If it's fixed point, shift left by 1 bit (multiply by 2)

That way, the 32-bit number stays in the same range like the initial 16-bit numbers. For further calculations, you would usually take the highest 16 bits, which for fixed point numbers simply means loss of precision.

We should pay attention to range overflow. Namely, multiplying two numbers between -1 and 1 will not overflow, however, addition might. Different algorithms deal with this in different ways, some ignore it, but that is often a sign of a mistake and a need to proclaim the calculations faulty. If it is about recognising the tones, then that usually means that you should give up on recognition - whether the calculations are simply forgotten, or an error is reported to be processed (reported, noted).

Regarding the previous - it's often that in formulas used for signal processing there are some small integer constants. They should not be converted into a fixed point number (by the way, that can not be performed trivially, because you would need to scale many other things, which can be very nonlinear). If it is the number 2, or any power of two, then we should simply apply shifting the required number of bits to the left. If overflow happens, then we have an irregular case we mentioned before (remember that: $2x = x + x$ - we can even use addition with itself instead of shifting).

Another thing regarding overflow. If it happens often, and preciseness is not important to us, or low amplitude signals are simply not important to us, then we can discard some bits from the input signal. For example, if the input is coming from A law expansion, then we can, instead of using 12, use let's say, 10 bits. Of course, the aforementioned Volt and 12-bits integer (or fixed point number) value ratios should be aligned with that (in the case given, multiply or divide by 4).

2 Goertzel algorithm

Goertzel algorithm is a way to calculate one result of the Discrete Fourier transform (abbr. DFT).

The main advantage is that, unlike the "full" DFT, it is a filter, in other words it is done for a certain frequency, while the DFT is done on many frequencies, evenly distributed over a range. This is very useful for recognising DTMF tones, since they are at "weird frequencie", which means that it is nearly unmanageable to carry out one DFT for them all, but it is possible to carry out several DFTs that would hit all the needed frequencies.

Also, this algorithm is expressed in a recursive formula (it represents a second order infinite response filter), which can be easier for coding or faster for calculation on some processor, or in a different setting (depending on how the samples arrive).

We will not develop the whole algorithm here, but only the "final solution".

2.1 Definition

We designate $x(n)$ the nth sample, $V(n)$ the nth result of Goertzel calculation, and K_k the Goertzel coefficient for the frequency at which the algorithm is being executed (filtering performed), then the following recurse formula applies:

$$V(n) = K_k V(n-1) - V(n-2) + x(n) \tag{9}$$

Where, by definition:

$$V(-1) = V(-2) = 0$$

Goertzel coefficient for the given frequency f is:

$$K_k = 2\cos 2\pi \frac{f}{f_s} \tag{10}$$

Where f_s is the sampling frequency; in telephony this is often:

$$f_s = 8kHz$$

Based on this we can conclude that for this algorithm it is possible to keep just two variables, it is not necessary to keep the whole row. Also, there is only one coefficient. For linear code, then the following Ruby code does the calculation:

```
v_n = v_n_1 = 0
0.upto(N) { |n|
    v_n, v_n_1 = Kk*v_n − v_n_1 + x[n], v_n
}
```

Ruby was chosen deliberately as it is almost pseudocode, so we od not have to determine the type of variable and concern ourselves with the ranges of values while showing the algorithm.

In the end, that is, after we process all N samples, from the $V(n)$ and $V(n-1)$, we get the following expression for the square of the amplitude of the processed signal at the given frequency:

$$V_{mG}^2 = V(n)^2 + V(n-1)^2 - 2\cos{(2\pi\frac{f}{f_s})}V(n)V(n-1) \tag{11}$$

The full expression was given here deliberately, because it gives a certain hint about its nature (who was the first to say "Cosine theorem"), and we can notice that Goertzel coefficient is used again, that is, this can be written shorter:

$$V_{mG}^2 = V(n)^2 + V(n-1)^2 - K_k V(n)V(n-1) \tag{12}$$

This result is in "Volts squared", however, in the frequency domain, and not in the time domain. That sounds very smart, however, in practice, it is only about the V_{mG}^2 not being a scaled value. To convert it into a time domain, we need to:

$$V_m^2 = (\frac{V_{mG}}{\frac{N}{2}})^2 = \frac{4 \cdot V_{mG}^2}{N^2} \tag{13}$$

For an explanation on why it is done like that, look for resources describing DFT.

Since signal is usually referred to in decibels ("milliwatt"):

$$P_{dBm} = 10\log\frac{4V_{mG}^2}{2R_{ref}P_{mW}N^2} = 10\log 2\frac{V_{mG}^2}{R_{ref}P_{mW}N^2} \tag{14}$$

This is a relatively inconvenient calculation for a DSP (the logarithm is calculated numerically), so levels in decibels are usually predetermined, and in the processing they are converted into squares of Volt, for which the following formula stands:

$$V_{mG}^2 = \frac{R_{ref}P_{mW}N^2}{2}10^{\frac{P_{dBm}}{10}} \tag{15}$$

Purely numerically:

$$V_{mG}^2 = 0,3N^2 \cdot 10^{\frac{P_{dBm}}{10}} \tag{16}$$

If we look at this as a filter, then its bandwith is:

$$B = \frac{f_s}{N} \tag{17}$$

since that is the width of one "frequency bin" in DFT. Since our bandwidth is usually preset, then based on that we can determine the number of samples:

$$N = [\frac{f_s}{B}] \tag{18}$$

If there aren't any other requirements, this is usually a good number. However, primarily detection speed, but also the ranges of recognition and non-recognition, influence this.

Nonetheless, it is good to notice that the number of samples cannot be arbitrary, at least not without consequences. Namely, as we have said, this algorithm is simply a way to calculate one result of a DFT. Thus, N should be chosen so that one result of DFT (the one we calculate with thios algorithm) is such that it catches our frequency in the middle of its frequency bin, so it reduces the amount of waste of our frequency to neighboring bins while doing DFT.

2.2 Observations

If we observe Goertzel algorithm “at work”, we can see that $V(n)$ gets various values, as if they were oscillating. On the other hand, $V_{mG}^2(n)$ increases monotonously. Generally, $V_{mG}^2(n)$ should be mathematically interpreted as a sequence that is converging towards the square of amplitude of the given signal at the given frequency.

If we observe on samples of the A law, we will note that the values $V(n)$ gets can be even over 300 (in SI units, Volts). Compared to the range of the A law, that’s over 265 times larger. Therefore, in some worst cases, we would need 8-9 more bits than there are in the A law, so, for a fully mathematically clean calculation, we need at least a 21-bit processor, practically 24-bit. Especially since the power we get goes over 20 000, for which even 24 bits is not enough.

This is why signal attenuation is often applied, as we mentioned before, with shifting the result of the A law for a few bits (two is somehow the most common - shifting for two bits is dividing by 4). We call this attenuation the “stupid weakening”, because it is carried out always, without analysis of the received signal.

We should note that the values for $V(n)$ increase with the decrease of Goertzel frequency, which is not really intuitive, since $V(n)$ is dependent on Goertzelg coefficient K_k. For most implementations, we look for a frequency up to 3000 Hertz, which, according to **??** for K_k, keeps us in the range $[0, 3\pi/4)$ for the given cosine, while it is constantly decreasing. That is, obviously, $V(n)$ has some sort of inverse proportionality (non-linear) to K_k.

Of course, this problem does not apply to power, otherwise this algorithm wouldn’t make any sense! However, we should note that, in that sense, Goertzel algorithm is easier to implement (we need less bits) for higher frequencies.

If we look at Goertzel algoritm as a filter, then we can get the corresponding frequency characteristic of this filter. The picture of this characteristic for this filter at 2000 Hz, in range from 1800 to 2200 Hertz is provided on the picture **??**.

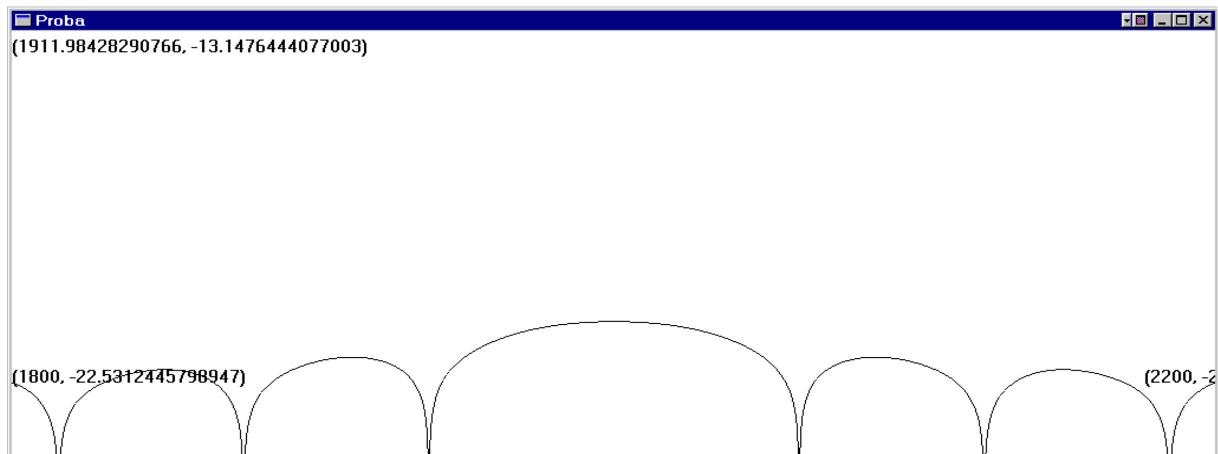

Figure 1: Frequency response of Goertzel fiter at 2000Hz in frequency range of 1800 to 2200 Hz

The image is a bit spartan, because it was taken from a screen of a Ruby program which draws

the frequency characteristics of various filters. There are specialized programs for that, however this one is part of a development system made at GVS. For example, the program is interactive, so it displays attenuation on chosen frequency (the one upon which we click). Nicer pictures can be found in books that describe DFT.

Anyway, it should be clear enough that we are talking about a set of “bells”, the highest (and widest) around Goertzel frequency, and the other ones are slightly decreasing and are narrower than and symmetrical to that frequency.

However, here we can notice a problem. Attenuation of the first bells (left and right) is -12dB, which is too little for the majority of implementations in tone recognition. Namely, a bigger level range is usually required. Without any extra work, Goertzel filter for 2000 Hz, on image **??**, would catch frequencies even at 1912 Hz, because a range above 20dB is needed, and in the image (upper left corner) we can see that the attenuation at that frequency is less than -20dB (it amounts to -13,14dB)

There are two possible approaches. One is that the samples are passed through a window function. Window functions are filters, but made with the intention of solving exactly these kinds of problems. There are a few popular ones, primarily Hamming and Blackman. All of these functions attenuate these secondary bells, and the majority also attenuate the “main” bell a bit, and all of them also widen the main bell (thereby spreading the bandwidth), while narrowing the secondary bell. Hamming’s window weakens the bells neighboring the main one significantly, and then the others a bit less, however, the lowest weakening of the neighboring bells is -43dB. Hamming’s window approximately doubles the bandwidth. Blackman’s window weakens all the secondary bells almost evenly, the lowest weakening is around -60dB, and the main belly widens approximately three times.

Even though window functions that do not attenuate the main bell exist, they are less popular because they widen it *very much*, and with it the bandwidth. In that sense, the mathematics for tone recognition is more complex, which is why the window functions are rarely used in the implementations for tone recognition. That does not mean you shouldn’t use it, however, it does mean that in this description they were not further considered and a different idea was implemented. We call it “smart weakening”.

Smart weakening is a kind of automatic signal amplification regulation. Namely, before going through Goertzel filter, let’s calculate the power of the input signal (in practice, we add up all the squares of the samples, which is the unscaled measure of the square of the effective voltage of the input signal). Then, depending on the level and the needed range, we attenuate the input signal for a certain level. That is usually done by shifting all the samples to the right by a certain number of bits (which is dividing by 2, 4, 8, 16…).

We should notice that smart weakening decreases the need for “stupid” weakening. Namely, if we attenuate the signal by, let’s say, 18dB at most, that means approximately 3 bits. Therefore, if we need a weakening of only three bits so we would stay in range while calculating, then we don’t need any stupid weakening. Even if we need a total weakening of five bits, this smart weakening means that we need a stupid weakening of just two bits. Usage of window functions doesn’t usually decrease the need for weakening because of range overflow, though further analysis, not yet performed, could prove window functions useful in that sense.

2.3 Implementation

For implementation, the interesting things are the ones which come as a consequence of a request for receivers and which are a consequence of the environment (processor and others) in which the algorithm is implemented.

2.3.1 Calculations

We noticed that, in the general case, 16 bit is not enough for calculating Goertzel algorithm. What do we do then? Having in mind that we are mainly using 16-bit DSPs?

Goertzel coefficient and V(n) For brevity, it's convenient to write only the coefficient, but since Goertzel coefficient can be greater than 1, for processing it is better to use the form:

$$V(n) = 2cos_k V(n-1) - V(n-2) + x(n) \tag{19}$$

where: $cos_k = \cos(2\pi\frac{f}{f_s})$. Thus, cos_k has to be in the -1 to +1 range, so we only have the factor 2 to think about. Here's how to implement this on DSP 56166:

```
    ; y0 ::= kos, x0 ::= v_n_1, b ::= v_n_2, a ::= t
    ; r3 ::= p (samples), y1 ::= sample (current)
    ;; loop (x1 ::= nSamples)
    do x1,L1                       ; for (; nSamples > 0; --nSamples) {
    mpy y0,x0,a                    ;     t = kos * v_n_1;
    asl a                          ;     t <<= 1
    add y1,a  x:(r3)+,y1           ;     t += sample; sample = *p++
    sub b,a                        ;     t -= v_n_2
    tfr x0,b  a,x0                 ;     v_n_2 = v_n_1, v_n_1 = t
L1                                 ; } // end for
```

Of course, this is just a part of the whole algorithm, but it is its core. We should note that this code does not deal with overflow, but relies on the fact that DSP56166 records overflow, so the check comes after the loop. Of course, we are left with the task of trying to avoid the overflow.

Firstly, depending on the frequency we have to catch, we can check which values $V(n)$ can get. We have the Ruby program from a while ago, with a little update:

```
v_n = v_n_1 = 0
v_n_max = 0
0.upto{N} { |n|
    v_n, v_n_1 = Kk*v_n - v_n_1 + x[n], v_n

    v_n_max = abs(v_n) if abs(v_n) > v_n_max
}
```

We get (in `v_n_max`) the maximam $V(n)$ gets. This simple calculation can be done in almost any programming language or a spreadsheet.

However we calculate `v_n_max`, if we divide it with the maximum of A-law, we get the ratio; rounding it gives us how many more bits we need "above" A-law:

```
MAX_A_LAW = 1.52
bits_more = Math.log10(v_n_max/MAX_A_LAW) / Math.log10(2.0) + 1
```

We used the well known identity: $\frac{\log_a x}{\log_a b} = \log_b x$.

If `bits_more > 3`, we have to discard the lowest bitse of the input signal (samples). But $V(n)$ depends on signal level, not just frequency. A-law provides only 4 bits of sample data, the rest is frequency and sign. Thus, we shall lose bits only for the weakest signals, which often are of no concern for tone recognition. Especially since "smart weaking" can help.

If we turn the previous calculation of `bits_more` into a function, then we can:

```
MAX_A_LAW = 1.52 # defined 'somewhere'

def determineMoreBits(x)
    v_n = v_n_1 = 0
    v_n_max = 0
    x.each { |x_n|
        v_n, v_n_1 = Kk*v_n - v_n_1 + x_n, v_n
        v_n_1 = t;

        v_n_max = abs(v_n) if abs(v_n) > v_n_max
    }
    Math.log10(v_n_max/MAX_A_LAW) / Math.log10(2.0)
end

def weaken(x, k)
    x.map! { |xn| xn / k }
end

x = []
## !! Fill x 'somehow'

weakening = 1
while DetermineMoreBits(x) > 0
    weakening *= 2
    weaken(x, weakening)
end
```

When this execults `slablyenye` will hold the needed weakening of the input signal. This is a power of two because this is most convenient, we can simply shift samples by a number of bits.

This should be determined for the highest signal level we need to recognize (taking into account the smart weakening). We don't care about anything higher. Namely, we have to keep track of the overflow during calculation, because it is difficult to be completely sure that we got rid of all possibilities of overflow with the smart and stupid weakening. If the overflow still happens during calculations, we should stop the execution of Goertzel algorithm and report error. In that sense the overflow will happen for signals higher than the highest one, which is not a problem.

We should note that one bit of weakening amounts to 6dB attenuation. That is illustrated in the following chart:

YYY	dBm
111	+3
110	-3
101	-9
100	-15
011	-21
010	-28
001	-34
000	-40

YYY represent the three bits of the exponent of A-law encoding.

Calculations of the power of the amplitude Once we calculate $V(n)$, all that's left is the final result. Using similar techniques as before, we can conclude that the power of the amplitude is up to a couple of hundred times greater than $V(n)$ (SI units are different, however, numerically it doesn't matter). Where do we find more bits?

Here we can resort to the following trick. If we really did stay within the range of 16 bits, since that practically represents the description in a fixed point number, then we have stayed within range $[-1, 1)$. Within that range, any multiplication stays within that range. In the expression of power we have two squarings that stay within range, as well as one multiplication that stays within range and a multiplication with two. If we divide $V(n)$ and $V(n-1)$ by two, then do the math, it will surely stay within the range of $[-1, 1)$.

It will not be the amplitude square we are looking for, but an attenuated one. On the other hand, we need to convert this result to a meaningful value somehow. Is it really important by what factor? We should note that we can't get more than 16 bits from a 16 bit processor within real-time constraints. In any case, 16 bits of accuracy is enough for the majority of our needs (they are mapped to decibels according to the attached chart).

Let's introduce it mathematically. Let's start with the equation for power (**??**) and divide it by four:

$$\frac{V_{mG}^2}{4} = \frac{V(n)^2}{4} + \frac{V(n-1)^2}{4} - \frac{K_k V(n) V(n-1)}{4}$$

Rearrange:

$$\frac{V_{mG}^2}{4} = (\frac{V(n)}{2})^2 + (\frac{V(n-1)}{2})^2 - K_k \frac{V(n)}{2}\frac{V(n-1)}{2}$$

If we divide $V(n)$ and $V(n-1)$ by two, then calculate the power using **??**, we'll get power divided by 4 Thus we only have to add 4 to all other factors we acquired along the way of doing the math on 12-bit samples on fixed point DSP. In short, we have to divide the equation **??** by four.

We need to take into account "stupid weakening". Since: $V_{eff} \sim V_m$, then: $V_m \sim x^2$, meaning that Goertzel result is proportional to the square of input signal. Which gives us:

$$V_{mGs}^2 = \frac{0,3N^2 \cdot 10^{\frac{P_{dBm}}{10}}}{K_s} \tag{20}$$

V_{mGs}^2 is what you get as the Goertzel output if input signal (x_s) attenuated by K_s.

Practically, equation **??** is to be calculated "in advance" and the value we get should be compared to the output of the Goertzel algorithm (filter).

We should note that this has nothing to do with the "smart weakening". The smart weakening is primarily used for solving small scale problems of bell weakening of Goertzel filter. It will reduce the need for "stupid weakening", however it does not affect equation **??**, because it does not weaken all signals, only some.

Finally, we should note that in practice no case was observed in which $V(n)$ and $V(n-1)$ are in range -1 to 1 while V_{mG}^2 was out of that range. Nevertheless, it hasn't been proven that it is impossible, so until it is eventually proven, it is safer to execute the action of dividing by four.

2.3.2 Smart weakening

We already described it, however, we have a question of how we determine which signals get weakened and which ones do not? We mentioned that we calculate the sum of the squares of the samples, which is a measure of the square of the effective voltage. To be precise:

$$V_{eff}^2 = \frac{1}{N}\sum_{i=0}^{N} x_i^2$$

Replacing V_{eff}^2 and rearranging:

$$NR_{ref}P_{mW} \cdot 10^{\frac{P_{dBm}}{10}} = \sum_{i=0}^{N} x_i^2$$

taking into account the "stupid weaking", purely numerically:

$$\frac{0,6N \cdot 10^{\frac{P_{dBm}}{10}}}{K_s^2} = \sum_{i=0}^{N} x_{si}^2 \tag{21}$$

Therefore, the thresholds for smart weakening, in ("milliwatt") decibels, should be recalculated into a sum of the squares of the (potentially attenuated) samples according to equation **??**, and then use that result in comparisons.

2.3.3 Detection requirements

Tones of one or two frequencies are the most commonly recognised. When it is two frequencies, then there are multiple possibilities - a set of frequencies that can appear, and we have to determine which two of them have actually appeared. The requirements differ, however, some things are common.

It is possible, for each of the frequencies, to deduce when it has to be, and when it must not be recognised. This is usually a frequency range (e.g., $\pm 15Hz$), but can be a level range (let's say, -4 to -32 decibels).

For Goertzel algorithm, the most important condition for recognition, from which we get the bandwidth, which gives the first approximation of the number of samples (equation **??**). As we have noticed, this number of samples can be changed (reduced, usually for faster recognition), however not significantly. There, the "overlap" trick is often applied - i.e. that the number of samples that are passed through the algorithm is as designed, but that the filtering is done more often than the number of samples; after one processing, not all samples are discarded, some samples are saved and reused.

The levels for detection are calculated with the equation **??** and are then used in processing, however, first we need to determine the "stupid weakening". Stupid weakening is determined by processing the signal at Goertzel frequency and the maximum level (+3dBm) and seeing by how much the signal should be attenuated.

It is there we run into trouble, since there is usually a request for what must not be recognised, which can pose a problem with Gercel's algorithm as a filter. It's possible that a Goertzel filter that fulfills all requirements simply does not exist! There we apply smart weakening and choose the sample length. As a rule, we should be able to come to a solution, if the requirements are not too strict.

Also, our trick with weakening can pose a problem, if there is a requirement that a signal of a greater level than specified *must not* be recognised. Luckily, this is almost never the requirement in tone recognition.

Special "benefits" arise for some sorts of dual tones, at which the requirements for all frequencies are not the same, and various combinations of requirements also exist, whereby, for the sake of simplicity, we take the same number of samples for all frequencies as per rule (of course, we have to let ther algorithm through all frequencies that potentially occur). For example, a DTMF

receiver is inconvenient because the frequencies are all mutually prime, so no number of samples matches all frequencies, thus we should find the one that is least bad or implement complex logic that synchronizes more receivers of certain tones with a different number of samples. We should note that in the 21st century it has become widely known that Goertzel algorithm is not appropriate for a complete fulfillment of the requirements for DTMF receivers, precisely because of their strict nature. However, Goertzel algorithm is still the most widespread, so it is good to understand when requirements simply cannot be fulfilled.

All in all, this is a classic engineering problem, in which we have to, with a bit of calculation and trial, conclude which is the least bad solution (hopefully good enough). Sometimes another filter can be added, and then we can conclude something based on that. For this to be done within a reasonable time frame, we need to make tests (providing data with samples) and then pass them through an algorithm set up in a way that will show whether the requirements are fulfilled or not.

2.3.4 The level difference between two tones in a dual tone

When dual tones are detected, one of the parameters of recognition is that the level difference between tones in a dual tone must be less than a certain value, given in decibels. Of course, since in Goertzel algorithm we get the square of the amplitude, we can not simply subtract the new values, we need some mathematics again. If we designate that difference with P_{dBR}, then we have the following derivation (under the condition that $VmG1 > VmG2$):

$$
\begin{aligned}
P_{dBR} &> 10 \log \frac{P_{G1}}{P_{G2}} \\
P_{dBR} &> 10 \log \frac{V_{mG1}^2}{V_{mG2}^2} \\
10^{\frac{P_{dBR}}{10}} &> \frac{V_{mG1}^2}{V_{mG2}^2}
\end{aligned}
$$

Of course, DSP division is unavailable or too slow. However, the problem is not only division, the problem is that P_{dBR} values are always greater than 1dB, so the left side of the equation is always greater than one, and we are working with values within the range of -1 to +1. So, we have to deal with that as well. Here is the further derivation:

$$
\begin{aligned}
\frac{1}{10^{\frac{P_{dBR}}{10}}} &< \frac{V_{mG2}^2}{V_{mG1}^2} \\
10^{-\frac{P_{dBR}}{10}} &< \frac{V_{mG2}^2}{V_{mG1}^2} \\
10^{-\frac{P_{dBR}}{10}} \cdot V_{mG1}^2 &< V_{mG2}^2
\end{aligned}
$$

In other words, we have to determine which tone's squared amplitude is higher, then multiply it with $10^{-\frac{P_{dBR}}{10}}$ and that has to be less than (or possibly equal to) the output of Goertzel filter.