Adaptive Gain Control

Julius Barendt (jur12jba), Isak Börjesson (dat13ibo), Kristoffer Syversen(mat13ksy)

March 28, 2017

Contents

1 Introduction

The goal of this paper is to introduce and explain the concept of an AGC, adaptive gain controller. An Adaptive gain controller is no foreign concept for someone familiar with basic signal processing. AGCs are widely used in devices such as AM Radio receivers, Radar, Video or in your Smart phone. The purpose of an AGC is to regulate a gain on an output signal based on the energy content of the input signal. A normal AGC will increase the gain for low energy samples and decrease it for high thus keeping a fixed volume on the output independently of input variations. We have chosen to implement our AGC to attenuate signals with both high and low energy content meaning we will remove low background noise when no one is speaking and attenuate the signal if someone is screaming or tapping on the microphone. We didn't implement an aggressive AGC since we still wanted some comfort noise.

2 Theory

2.1 The AGC

The theory behind an AGC is fairly simple and can be implemented in two steps. The first step is to derive an appropriate energy content from the sampled input. Since we are working with sample blocks we use 32 samples and generate a power level based on the mean of the squared samples. Below in equation (1) you can see how we calculated the instant energy content called P_i.

$$P_i = \frac{1}{32} \sum_{i=1}^{32} x(i)^2 \tag{1}$$

Then we use a first order IIR-filter where we calculate an average energy level, denoted P_a, based on the previous average energy content and current instant energy content weighted with the variable α as illustrated in equation (2).

$$P_a(n) = \alpha P_a(n-1) + (1-\alpha)P_i \tag{2}$$

The second step is to generate a gain based of this energy level. This is done by converting the energy to full scale dB, *dBFS*. This is done by taking the log of P_a and multiply by 10. When we have our energy content in dBFS we use a predefined function to calculate our gain. The function is illustrated in figure 1. When we convert this value back to a linear we will get a value in the interval $(0, 1]$ this value is our gain. This gain is then multiplied by our original vector containing 32 samples and then sent as output to the headphone jack.

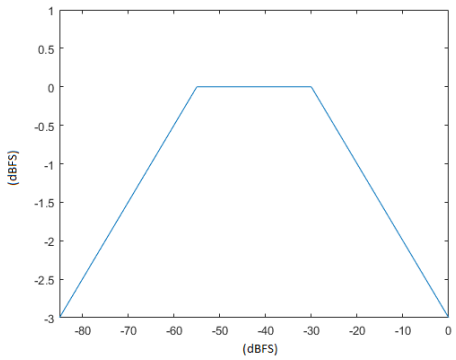

Figure 1: Calculating gain

2.2 The time constant

The time constant is a value that decides the time it takes for the output of the AGC to reflect a 63% percent change in the input amplitude. The time constant, usually written with a τ, can be derived from the constant α in the filter description (2) above by expanding the recursive filter function and rearrange the terms. This is done in a introductory class in signal theory.

This affects how fast the AGC responds to a change in input power, for example if the α is high, then the AGC will take longer changing the output level because (1 - α) will be small. And the other way around for small values of α.

This is useful to know when designing the filter, if someone taps the microphone for example then then we quickly want to decrease the output level and in our case, if no one is talking we want to cut away the noise. We don't want this to happen to quickly though as this will cut away all the comfort noise between words. A perfect combination(that we did not implement would be to have different α, one for the increase and one for the decrease of noise levels, so that the decrease for high volumes works faster than the one for low levels.

The conclusion on this discussion on time constants is that it is important as the choice of one affects how the AGC behaves, a badly tuned time constant makes the AGC unusable.

3 Implementation

3.1 MatLab

We started out by implementing the AGC in Matlab. First we wrote the function to calculate the energy content of a block of samples. We knew that the DSP would give us chunks of data in 32 elements long vectors. All that was left to determine was the weighting variable α. We knew that we wanted to rely more on the current sample block than the previous, which means that our α should be in the range $[0.90, 1)$. The closer towards 1 the more we will rely on our current block. We experimented some with different values on α and found that choosing it to $\alpha = 0.97$ gave us a smooth energy development. The middle graph in figure 2 is the energy content. The red graph is t randomly generated noise and the green graph is the output after going through our algorithms. We see that the rise in energy content is relatively slow but this also means that we will have a less

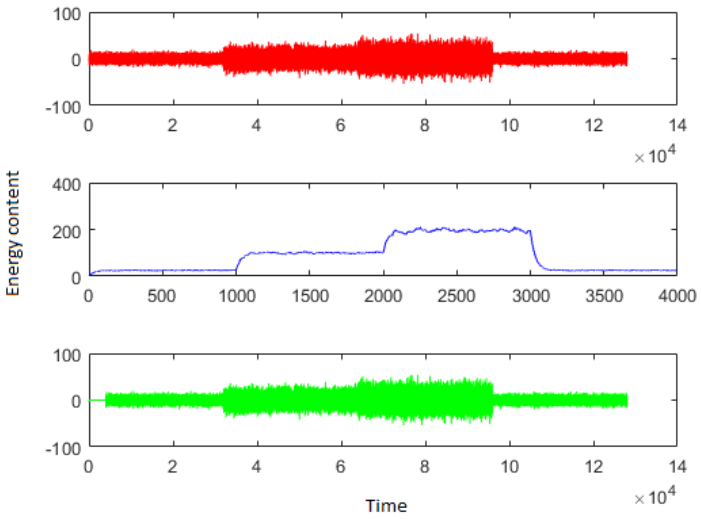

Figure 2: Generated signal, energy content and attenuated signal

jumpy signal. Choosing α is a trade off between the energy rise time and consistency. The code is fairly simple and is written below.

```
function y = process(x, prevPL)
    a=0.97;
    y=a*prevPL+(1-a)*sum(x.^2)/32;
```

```
end
```

The next objective was to generate a gain based on this energy content. We converted our energy to full scale dB by the simple formula

$$dbFS = 10 * \log_{10}(prevPL) \tag{3}$$

and matched our result with an desired output as in figure 1.

The shape of this function is determined by 4 factors. An increasing slope, decreasing slope, a minimum threshold and a maximum threshold. Initially we guessed these values but as we tested our implementation on the hardware we figured out which limits were appropriate based on the behavior of the DSP. After converting the matched decibel value back to linear scale we get our gain which we multiply by the initial input and then put as output.

3.2 C

The code loading the audio from the microphones and later pushing the data to the headphone jack was done in C. The DSP processor is event driven and when a button is pressed or audio in recorded a event is generated, and different function calls are performed based on what kind of event. To know function a event should call the following line of code is used

```
interrupt(SIG_SP1, process);
interrupt(SIG_USR0, keyboard);
interrupt(SIG_TMZ, timer);
```

`SIG_SP1` is the incoming audio event, and it should call a function process. `SIG_USR0` is the button pressed event that should call the keyboard function and `SIG_TMZ` is a timer interrupt calling the timer function.

The function process is where the AGC is applied.

```
void process(int sig){
        int n;
    sample_t *u32 = dsp_get_audio_u32();    /* mic 1 and 2 in, headset out */

    for(n=0; n<DSP_BLOCK_SIZE; ++n) {
        mic1[n] = u32[n].right;
    }

    if(on > 0) {
        prvSmpl = PWR(mic1,prvSmpl);

        float g  = GAIN(prvSmpl);

                gains = powf(10, g);
    } else {
        gains = 1;
    }



    for(n=0; n<DSP_BLOCK_SIZE; ++n) {
        u32[n].right = (mic1[n]*gains+ mic1[n]*gains);
        u32[n].left  = (mic1[n] + mic1[n])*gains;
    }

}
```

The framework of the DSP works in the way that if we read from the array `u32` we get the audio in data but if we write to the array we write to the audio output. We begin by reading all the audio data into our own array called `mic1`. Lastly we multiply our array with the calculated gain and write then to the `u32` array and thus we get the modified audio playing through the headphones. Between reading and writing we send the data through our assembly functions, `PWR` takes the audio array and the power from the last segment and calculates the current segments power by squaring all the samples and dividing by the sample size. `GAIN` takes the calculated power and performs

the AGC calculations as explained previously. `prevSmpl` is a global floating point variable holding the power from the previous processed segment. `On` is a variable that we can change by pressing a button on the DSP, it turn on or off our AGC.

3.3 Assembly

Our assembly implementation consist out of two files and functions. The first `PWR` calculates the power by looping through all the samples in an array and squaring it. The second `Gain` performs our AGC calculation.

```
i0 = f4; //Pointer to in, prevIn in f8
f0 = 0.97;
f8 = f0 * f8; // \alpha*prevIn
```

On this DSP the variables sent to a function are stored in register R4 and R8 for the first and second parameters. At the end of the function, if it should return a value it is always the value in `R0` that is returned. Using this knowledge we can optimize the memory usage by storing most of our calculation affecting our return value in R0 as this is the value return anyways. Register `Rx` and `Fx` is the same physical register the only difference is thar `Rx` is a integer value and `Fx` is a floating point.

The most interesting part of `PWR` is the loop that goes through all our data points.

```
lcntr=32, do sum_loop-1 until lce;

        f4 = dm(i0,m6);
        f6 = f4;
        f2 = f4 * f6; //square sample

        f0 = f0 + f2; // accumulate

sum_loop:
```

`lcntr` is a loop counter where we specify we want to loop the `sum_loop` 32 times. `f4 = dm(i0,m6);` tells our compiler that we want to take the value stored in `i0` (prevIn pointer) and then increase it by `m6` which is a register that always contains the number 1. Works the same way as *f4 = prevPL[i++]* would in C. If we instead had used the call `f4 = dm(m6, i0)` we would only have got the first value of the array and not increasing it, the C equivalent would be `f4 = prevPL[i + 1]`. Then we copy the register `f4` into `f6` to be able to square the value without a power function. Lastly we accumulate the squared sum in `f0` because thats the value we want to return, after dividing by 32 which we do by multiplying the `f0` register by 0.03125.

In `GAIN` we have two interesting parts, the first is external function calls to C code.

```
ccall(_log10f);
f1 = 10.0;
f0 = f1 * f0;
```

Here we want to perform the C code `f0 = 10*log10(prevSmpl)`. Here we used the fact that the input variables always are stored in register `f4`. At the start of our function we have `prevSmpl` in `f4` from our C code, if we do nothing and just call `ccall(_logf)` the register `f4` will be used as input to the log function so we have performed `f4 = log10(prevSmpl)` right at the beginning. We also use the fact that return values are always stored in `f0`, so after the `ccall` we have the result in `f0`, all we have left to do is to multiply `f0` with 10.

The second part is conditional jumps.

```
f10 = f0 - f1;
if LT jump _LWR;
```

To do conditional jumps the DSP uses the last computation done by the ALU. Here we want to check if `f0` is grater than `f1`. If `f1` is grater the result will be negative, and we use the keyword `LT` which means that if the result is below zero it should jump to the function `LWR`. In the same way if the result is greater than zero and we want to jump, we use the keyword `GT` instead.

The DSP also have a lot of different features that we did not use. Mainly the ability to perform two instructions at the same time. Because unlike ordinary systems this DSP have two types of memory, one *Program Memory (PM)* and one *Data Memory (DM)*. The program memory is set

in code and do not change during run time, while data memory is dynamic. Usually you can not read and write to memory at the same time, but having two different memories leads to the ability to read from PM and at the same time write to DM or vice versa.

4 Result

We managed to implement our AGC in MATLAB, C and in assembly. However it was very aggressive and acted more as a noise gate, but this is something we can easily fix by tweaking our α value and the cut off limits. Further improvements of the AGC could include some more tweaking of our 4 variables that determine slope and limits. We could also implement two different α for when the energy contenis rising or declining. This would allow us to get an AGC that responds quickly when the energy content rises quickly but also keep a steady volume when the energy content is stable.