Adaptive Gain Control in Digital Signal Processors

Daniel Jankovic ada08dja@student.lu.se
Magnus Johansson ada08mjo@student.lu.se
Martin Lichota elt11mli@student.lu.se

July 26, 2015

Contents

1 Introduction

The purpose of this paper is to demonstrate the implementation of AGC, *Adaptive Gain Control*, in a DSP, *Digital Signal Processor*. In analog electronics AGC is essentially a closed-loop regulating circuit, which serves to maintain a stable signal amplitude despite variations at its input.

AGCs are used in big variety of products e.g. radios, TV, radar, hearing protection to name a few. In this project the AGC is constructed through mathematical algorithms and programmed into a DSP. The primary functionality is to reduce static noise and to dampen high power input signals.

2 Theory

The basic concept of an AGC is that it operates within a specific gain interval, which allows it to control the signal depending on its strength. If the signal is weak, the AGC will increase the gain and if the signal is strong, it will do the opposite, that is, decrease the gain. In this project an AGC will be modified to decrease the gain instead of increasing it when the signal is weak. The thresholds are predefined to fit the specific application.

There are many ways to build such a device. In this project, mathematically described, it is constructed by first calculating the power of the signal with the following formula:

$$P(n) = \frac{1}{N} \sum_{k=0}^{N-1} x^2(n-k) \tag{1}$$

where $P(n)$ is a vector.

The vector is then divided into a predetermined amount of fragments, each a sum of the values in the respective range. Instead, the values of the sums of the fragments could be used in a *recursive averaging* which allows averaging without memory. The formula is as follows:

$$P(n) = \alpha P(n-1) + (1-\alpha)x^2(n) \tag{2}$$

where α is a constant between $0, 8 - 1.0$. This procedure is done because of the memory restriction in the DSP. α can be related to a time constant τ, which gives a more intuitive understanding of the forumlas behavior. Repeating formula (2), the recursive process, yields following relation:

$$y = \alpha^n$$

rewriting this equation to

$$y = e^{n \ln \alpha} \tag{3}$$

and differentiation with respect to x,

$$y' = \ln \alpha \, e^{n \ln \alpha}$$

gives $y'(0) = \ln \alpha$, the slope of the tangent line at $n = 0$. Since $y(0) = 1$, the full equation of the tangent line must be

$$h = n \ln \alpha + 1$$

The time constant is defined at the point where the tangent slope crosses the x - axis. This can be found when $h = 0$, which gives

$$\tau = -\frac{1}{\ln \alpha}$$

Inserting $\tau = n$ in equation (3) gives the important relation

$$\alpha = e^{-\frac{1}{n}}, n = tF_s \tag{4}$$

where F_s corrensponds to the sample frequency and t to the time in seconds.

Now the overall power of the signal can be calculated and it remains to determine the gain. The first step is to convert the power values to a logarithmic scale, in this case dB (decibel). It provides a linear relationship thus making it less difficult to create algorithms for the AGC. The conversion formula is

$$y = 10log_{10}x \tag{5}$$

where y is the decibel value and x the power input.

The second step is to determine within which interval the gain should be increased or decreased. The graph in figure 1 aims to give a better understanding of this process.
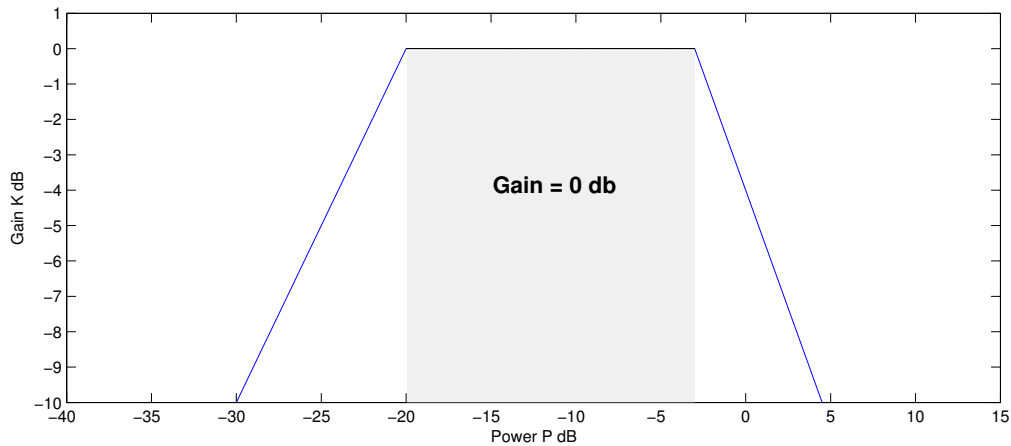

Figure 1: Displaying the different gain intervals.

Basically, the interval containing lower energy has its gain increased by a linear formula

$$y = kx + m \tag{6}$$

and in the interval with plausible amount of energy we keep the gain at maximum. When the energy reaches undesirably high levels, the gain decays by (6) as well but with a negative slope instead of a positive.

The gain itself is modeled to vary between 0 and 1, where 0 represents no gain at all and 1 is maximum gain. These values are also converted to dB hence the y-axis varies between 0 dB and -10 dB. Lastly, the gain factor is multiplied with the output signal.

3   Implementation

The implementation stage is divided into two parts - one that is conducted in Matlab and a second part where the DSP is programmed using C programming language.

The code of each respective part will be shown in the subsections below followed by a description of how it works.

3.1   Matlab

```
audio = getaudiodata(recObj);
buff = buffer(audio, 256);

buffSquared = buff.^2;
```

In the code segment above the values from the audio signal are obtained and inserted into a vector. Then, the vector is divided into fragments of 256 values each which are inserted into a matrice. Lastly the matrice is squared in accordance with formula (1).

```
for i = 1:n
  vec(i, 1) = sum(buffSquared(1:end,i));
```

```
end

row = size(vec, 1);
```

The code is summing the fragments and inserting the results into another vector (vec). The size of the rows is saved in a variable.

```
a1 = 0.99;
a2 = 0.8;
alpha = a1;

P = zeros(row,1);
P(1,1) = 0;
 for i = 2:size(vec,1)-1
     if vec(i,1) > vec(i-1,1)
         alpha = a2;
     else
         alpha = a1;
     end
     P(i,1) = alpha*P(i-1,1) + (1-alpha)*vec(i,1);
 end
```

Now comes the tricky part; recursive averaging (2) is used to calculate the average power of the fragments (vec). Different values of alpha are used depending on if the power is rising or declining. A lower value gives a larger slope whereas a higher value gives a smaller slope. When the power is rising it is of great importance to quickly lower the gain. When the power is declining a soft transition is desirable, which can be obtained with slower gain change (higher value of alpha).

Once the correct alpha is chosen, the power values of the fragments are calculated using the formula (2). The exact values used in this project were determined by experimentation and simulation which will be further discussed in section 4.2.

3.2 C-language

Programming a DSP from scratch is a time consuming task and to complete the project within the given deadline the provided framework was used as starting point. It consisted of one *.h* and two *.c* classes where one of them had the *Main* function. The standard configuration in *framework.h* was used since the DSP did not need further tweaks.

In *main.c* the functionality was expanded in regard to the *process()* function. New variables were introduced to support the implementation of the AGC. Majority of them where brought out in the global scope since calculations where needed in forthcoming interrupt.

```
...
static float pow_sum;       // power sum of quadratic X[n]-value for entire block
static float pow_sum_old;   // placeholder for old value, important for deciding alpha
static float pow_rec;       // memory cheap recursive power algorithm
static float alpha;         // alpha value
static float y;             // inverse log value
float constant LOW_THRES = -20; // Threshold for low power
float constant HIGH_THRES = -3; // Threshold for high power
bool filter;                // bool value for enabling the AGC filter
...
```

void process(int sig) is the interrupt function which performs the AGC algorithm in the DSP. It executes in the following order:

1. Get block audio from the DSP, put the sample in a array with fixed size.

    ...

```
// Get a pointer to the current audio block.
sample_t *audioin  = dsp_get_audio();
sample_t *audioout = dsp_get_audio();

// Copy audio from left channel to input buffer. Audio samples are 32 bit
// fixed-point values in the range [-1, 1] so no additional scaling is required.
for(n=0; n<DSP_BLOCK_SIZE; ++n) {
X[n] = audioin[n].left;
}
...
```

2. Calculate power summation for the block and save the value.

```
...
// Square X[n] and add to pow_sum
// Linear buffering
for(n=0; n<DSP_BLOCK_SIZE; ++n) {
pow_sum += X[n] * X[n];
}
// Save the previous power sum calculations
pow_sum_old = pow_sum;
...
```

3. Decide alpha, depending whether the power is increasing / decreasing compared to previous result. *explained in section 3.1.* An alpha value of 0.8187 and 0.99 gives approximate integration time of 10 ms respectively 2 s.

```
...
// Decide alpha in regard to the power calculations
// If power is increasing a lower alpha is applied
// and vice versa. alpha values are predefinied.
if (pow_sum > pow_sum_old) {
alpha = 0.8187;
} else {
alpha = 0.99;
}
...
```

4. Perform low memory cost recursive summation in order to find the possible gain.

```
...
// Recursive averaging algorithm
pow_rec = (alpha *pow_rec) + (1-alpha)*pow_sum;
...
```

5. If AGC is enabled, convert the value from the recursive power algorithm to *dB* and determine if the power is below, above or within the predetermine range. If the logarithmic value is not within the interval, calculate the gain with predefined linear function. Convert the *K* variable back to power. This will be the factor which the input signal will be multiplied with. Perform the multiplication and save the result in a new array.

```
...
    if (filter) {
        float value = 10*log10(pow_rec);
        if (value < LOW_TRES) {
            K = (value + 20);
            factor = pow(10,(K*0.1));
        } else if (value > HIGH_TRES) {
            K = (-1.333333 * value) + (3 * -1.333333);
```

```
            factor = pow(10,(K*0.1));
        } else {
            factor = 1;
        }
        for(n=0; n<DSP_BLOCK_SIZE; n++) {
            Y[n] = X[n] * factor;
        }
    } else {
        memcpy(Y, X, sizeof(X));
    ...
```

6. Put the data back into the registers.

```
    ...
    for(n=0; n<DSP_BLOCK_SIZE; ++n) {
        audioout[n].left = Y[n];
        audioout[n].right = Y[n];
        }
    }
    ...
```

4 Problems

4.1 Finding good alpha

One problem with implementing the AGC is finding good alpha values to get good and smooth processing of the sound. Having poorly chosen alpha values can severely affect the sound quality. There are no universal good alphas, rather we have to calibrate the alpha values for each independent input source. This is a trial and error process that takes time to complete. By comparing the recursive averaging algorithm in Matlab to the linear averaging algorithm we got an idea of what range our alpha values should be within. The method we used in Matlab is very time consuming and can not be used with the DSP's limited resources. The revealing of the relation between alpha and time constant τ could have saved us additional time.

4.2 Time-related debugging issues

A real-time problem with the DSP is printing data to the terminal when running in emulation mode. The standard c-function *printf* is used to print data. The problem with doing this however is that each print takes $\sim$ 500ms and since we get a new block of audiodata every 2ms there will be unprocessed data flowing through the DSP. This means that if the processing of a block is delayed by something, for instance *printf*, the block will simply pass through without being filtered or altered in anyway.

5 Results

With our filter algorithm implemented on the DSP we have a functioning AGC. We can hear how background noise is dampened when we activate our filter using the buttons on the DSP. We can also hear how loud noises are dampened quite well. To get a better perception of the implementation of the AGC in the DSP, an eight seconds long audio recording, consisting of eight hard knocks, was played through the DSP. Following is the result of the audio being played with the AGC disabled, respectively, enabled.
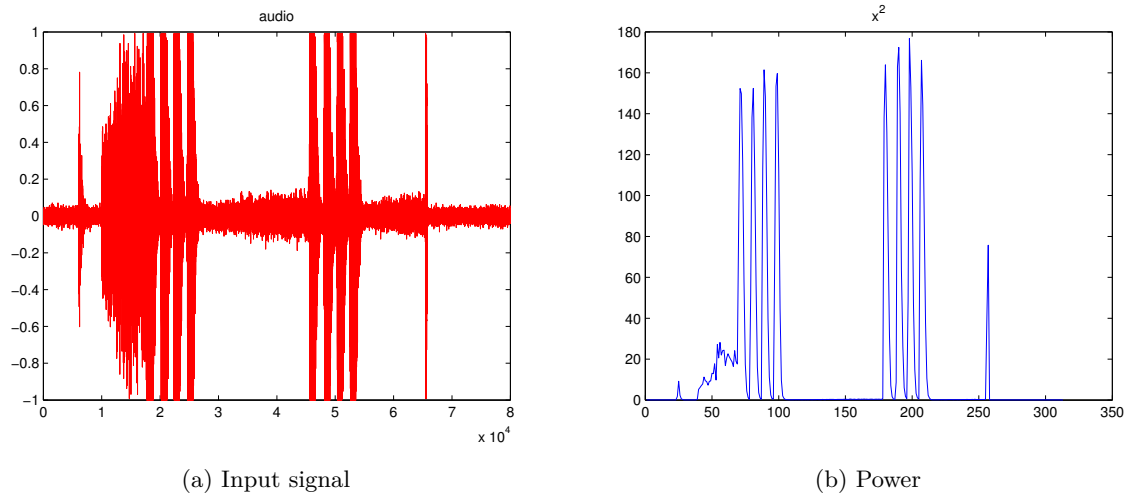
(a) Input signal

(b) Power

Figure 2: Graphs representing the eight knocks with the AGC disabled
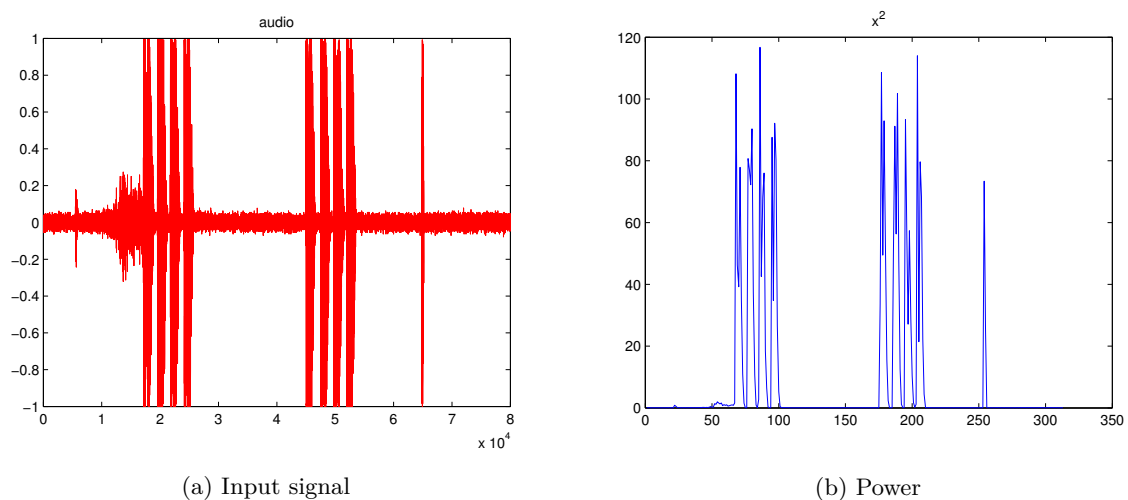

(a) Input signal

(b) Power

Figure 3: Graphs representing the eight knocks with the AGC enabled

6   Conclusions

In doing this project we have learned the basic algorithms behind commercial AGCs. We learned how an DSP works and seen how these devices can be used to manipulate sound in real-time. We have learned how to use their limited resources together with the mathematical models used in measuring sound and controlling gain. We have also seen what problems can arise when programming a DSP. Like we discussed in the problem section debugging a DSP can be tricky since printouts can seriously affect the DSP performance. We have also seen how the selection of alphas can affect sound quality and the performance of the gain filter we have made.

We have also learned the programming enviroment VisualDSP++ and used our knowledge in the C language to implement our program.