ETIN80 Algorithms in Signal Processors - Project Course

Oscar Gunnesson
David Andersson
Filip Pajalic
Instructor: Mikael Swartling

Execution Date: 18:th January - 6:th March
Submission date: March 13, 2017

Contents

1 Description

This projects purpose is to create an Adaptive Line Enhancer (ALE) using a Digital Signal Processor or in short DSP. The goal of the ALE is to remove tonal components from a signal. This technology can for example be useful in industrial environments where you want to suppress tonal components (e.g. sound from horns or industrial equipment) but keep speech. Another area for this is sound recording, to get rid off noise.

In a real life situation it is most likely that the disturbing tonal signal is not known. Therefore the ALE needs an adaptive filter to predict what signal to be removed. This can be solved by using a LMS algorithm, which predicts the filter weights for a Weiner filter. Using only LMS has its flaws, but can be further improved by normalizing the input or adding a leaky component to the filter coefficient update. Further details will be provided in section 4.

Figure 1 shows a block diagram over the ALE using LMS. The input signal is delayed and sent through the LMS algorithm and then compared to the non delayed input signal. This process is repeated until the optimal filter coefficients are found. The delay will strive to make all the uncorrelated parts of the signal pass through unaltered and all the correlated parts to be suppressed.

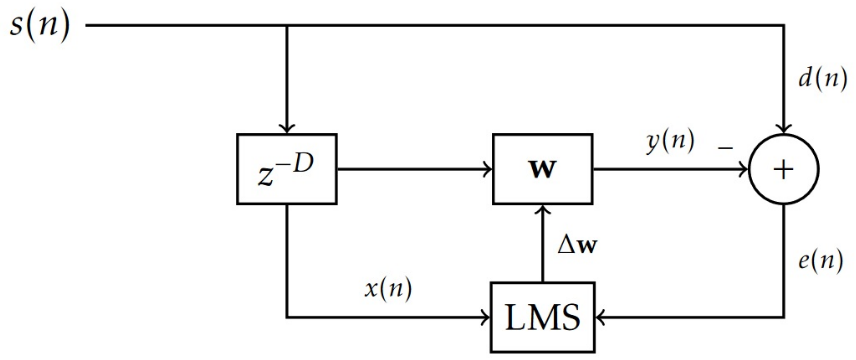

Figure 1: Block diagram over the ALE.

New problems might arise when implementing an algorithm on a DSP versus Matlab as the DSP has real-time requirements that the solution has to account for. This mean that the computation must be completed before the next sample occurs. Choosing the right algorithm and implementing it efficiently is key to meet the computational requirements. The DSP is also limited in memory which means that it is important handle data correctly. One must be selective about what data to store and where, and what is to be remove.

2 Method

2.1 Hardware and experimental setup

The DSP we are using is an ADSP-21262. As mentioned the code is written in C and compiled using Visual DPS 5.1. The setup is a line in to our sound source, a headphone out, four buttons and two LED lights. The LED's signal if the LMS is active. The buttons are used to:

- Adjust step size
- Toggle filtered signal and none filtered signal
- Reset filter coefficients
- Toggle LMS and sound straight from source

2.2 Development process

At first the algorithms was implemented in Matlab to test and verify that the theory was sound, and the code was correct. Using Matlab first was a good way to solely focus on the algorithm and not get any hardware related problems, which might occur when implementing it on the DSP.

When the implementation was approved, it was translated to C code. A testbench was setup in C to test whether or not the translation was successful. This was done using a standard C editor and gcc compiler. We also replaced the algorithms in Matlab with wrapped C code (MEX) to compare with the results of our initial tests.

After everything looked fine in the testbench, the algorithm was integrated with a framework provided in the course. The final tests were done by streaming music with an added sinusoid through the algorithm and then using our ears to hear whether or not the sinusoid is successfully removed.

3 Theory

3.1 LMS

When designing a filter in an optimal scenario where we have a input and a desired signal and the statistics concerning these are know, one could use a wiener filter. In this case we could calculate our coefficients using the correlation matrix of the input (R) and the correlation vector between the input and desired signal (p):

$$w_o = R^{-1} \cdot p \tag{1}$$

The filter coefficients $w_o = [w_1, ..., wn]$ are tuned such that they minimize the error $e(n)$ in the mean squared sense. In figure 2 one can see a block diagram of how the wiener filter works.

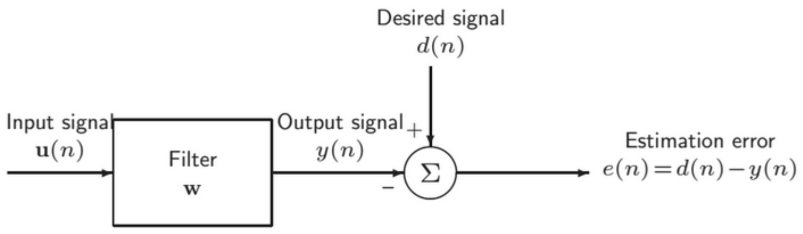

Figure 2: Wiener filter block diagram

But since we have a continuous stream of data, we instead settle for a estimate of the correlations, which results in the Least Mean Square (LMS) algorithm:

$$\begin{aligned} \hat{R}(n) &= u(n) \cdot u(n)^H \\ \hat{p}(n) &= u(n) \cdot d(n)^* \end{aligned} \tag{2}$$

The purpose of the LMS algorithm is to find the near optimal filter coefficients for a given input signal. The LMS method is based on the method of the Steepest descent with the difference that the statistics are continuously estimated. This feature of the LMS algorithm makes it adaptive and suitable for input signals that change over time.

The difference between the Wiener filter and the LMS algorithm can be seen in figure 3 where the Wiener filter using steepest descent converges to the optimal solution, but the LMS has a constant error and never fully reaches the optimum. [1]. The algorithm used is described in table 1.

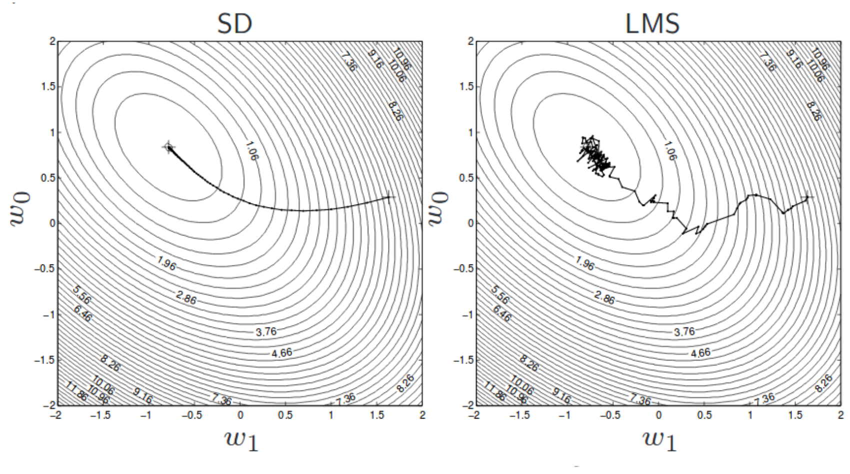

Figure 3: Convergence of steepest descent and LMS algorithm

Algorithm 1 LMS

```
1: procedure UPDATE
2:     $y = \hat{\mathbf{w}}^T \cdot \mathbf{x}$
3:     $e = d - y$
4:     $\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}} + \mu \cdot \mathbf{x} \cdot e^*$
```

where $\mathbf{x}$ is the input vector, d is the desired response, $\hat{\mathbf{w}}$ is an estimate of $\mathbf{w}$, and μ is the step size.

The LMS algorithm only converges if:

$$0 < \mu < \frac{2}{\lambda_{max}}$$

3.2 NLMS

One problem with the standard LMS algorithm is that the step size is fixed. This creates problems when, for example, the amplitude of the signal is altered. If this is the case the algorithm might not converge in sufficient time, or diverge. To safeguard against this NLMS normalizes the amplitude using the $\|\mathbf{x}\|^2$ and a guard term a making the step size change [1]. The algorithm used is described in table 2

Algorithm 2 NLMS

```
1: procedure UPDATE
2:     $y = \hat{\mathbf{w}}^T \cdot \mathbf{x}$
3:     $e = d - y$
4:     $\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}} + \frac{\tilde{\mu}}{a+\|\mathbf{x}\|^2} \cdot \mathbf{x} \cdot e^*$
```

where $\mathbf{x}$ is the input vector, d is the desired response, $\hat{\mathbf{w}}$ is an estimate of $\mathbf{w}$, and $\tilde{\mu}$ is the step size. The NLMS algorithm has a standard stability if:

$$0 < \tilde{\mu} < 2$$

3.3 Leaky NLMS

Another problem that can occur when using the LMS is that the tonal component the the algorithm are supposed to filter out, might not be fixed. Imagine being in a industrial environment where multiple sources might be present at different time points. The filter might converge to remove a high pitch sinusoid, which at another time stops sounding. This will leave the filter removing frequencies which are not there and instead filtering out useful sound. To combat this, a ”leaky” factor is introduced to the previous filter coefficients $w = \alpha \cdot w + \beta$, $\alpha \leq 1$ [1]. The algorithm used is described in table 3

Algorithm 3 Leaky NLMS

```
1: procedure UPDATE
2:     $y = \hat{w}^T \cdot \mathbf{x}$
3:     $e = d - y$
4:     $\hat{w} = \hat{w}(1 - \mu \cdot \alpha) + \frac{\tilde{\mu}}{a + \|\mathbf{x}\|^2} \cdot \mathbf{x} \cdot e^*$
```

Where $0 \leq \alpha \leq \frac{1}{N}$

4 Result

The LMS, NLMS and leaky were successfully implemented in C and run on the targeted DSP. Due to NLMS being an addition to the LMS, the LMS was replaced by NLMS as the project progressed. We discovered early in the process that the step size was difficult to tune using LMS, and it quickly diverged when changing the volume of our source signal. This also goes for the leaky factor, even though we did not preform any extensive tests on the functionality. We did some test starting and stopping sinusoids, but could not hear any significant changes. One possibility would be to use another input signal that was more susceptible to this, were it was easier to differentiate.

Tests with different setups with delay, step size and filter coefficients was done in Matlab to get a feel of how the algorithms were effected. The parameters that worked well in Matlab coincided well with what the theory said. To get a feel of the step size needed we plotted the convergence of the filter as in figure 4 and 5. Figure 4 depicting when a convergence was not fast enough and 5 a typical good convergence.

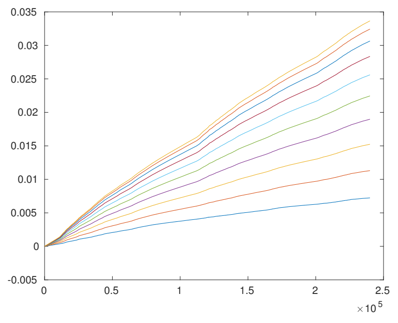

Figure 4: Slow convergence of 10 filter coefficient

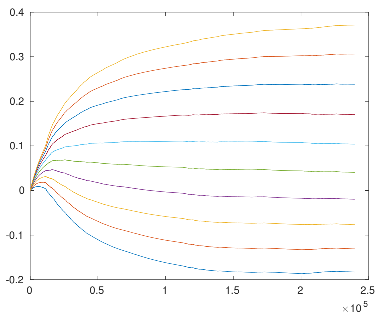

Figure 5: Fast convergence if 10 filter coefficient

As for the DSP, the delay had to be large enough for the music to be uncorrelated, but small enough to not use up too much memory. In contrast to Matlab we found that it was hard to hear the difference after a while when increasing delay. In the end we used a delay as large as our filter size when testing for convenience.

Having a small step size caused the signal to converge too slow and if it was to big caused it to diverge. We found that the size of the step size could be much larger when running the algorithm in the DSP as supposed to Matlab.

As for the filter size, we ended up using a filter of size 150. This too because we could no longer hear any significant improvement. The filter size was fairly consistent with our findings in Matlab.

Another thing we tested was using multiple sinusoids. The idea behind this is that a single sinusoid creates a notch filter canceling out that specific frequency, but if to many sinusoids are present, the notch filter starts to remove part of the desired signal (In our case the music). Here we encountered some issued, while we could easily cancel multiple sinusoids when using a computer as our input, playing both music and various sinusoids, we could not reproduce this in other cases. For example when we generated sinusoids using Matlab, added music and used a phone as source, there were distinct noise that could not be filtered out. One explanation could be that the sinusoids became correlated in some sense during some step of the process.

5 Discussion

Using the NLMS algorithm with a leaky factor suppressed the tonal components and gave us the best result. It took a lot of trial and error with the stepsize, delaysize and blocksize to get it to sound as good as possible. Theory and Matlab testing did not fully agree with what the DSP wanted as values. Matlab gave us a better understanding of the problem and a great overview as we could plot interesting signals. But when it was supposed to be implemented on the DSP problems occurred that were not present in Matlab.

An alternate solution that we worked on was the Block-LMS which worked perfectly in Matlab. The algorithm was also implemented in C and tested on a computer before transferring the code to the DSP. When the algorithm was brought over to the DSP it refused to work properly and we could not figure out why. But it taught us something about the limitations and what not to use on the system. A possibility for why the block-LMS failed is the fact that we could not get a high filter length before the code would crash on execution. This could be due to the fact that the code had flaws that we were not able to identify. The question is also if the block-LMS is suitable for this task, considering that a improvement, the fast-LMS, exist to combat the high computational complexity. Another approach to this project would have been to go for the fast-LMS instead.

The method of using a wrapper for our C-code was by far the easier ways of debugging, since we had a hard time getting printouts from what was going on inside the DSP. This if of course a general problem when working with the DSP, but one that we were not accustomed to. It also gave us the opportunity to see step by step how the Matlab and C-code differed using the same input signal. Our initial test using gcc when for example implementing the circular buffer were toy examples which did not always give enough information. The only problem we encountered with the wrapping was that there we some numerical difference when comparing, probably cause by different round-offs.

In conclusion concerning the block-LMS, it was very different to implement a reference version on a computer and to implement it straight on the DSP. Problems occurred when the c-code was integrated in the DSP program even though every step of the progress was thoroughly tested. When instead implementing the normal LMS and NLMS the translation was straight forward, but instead there were other DSP related problems we encountered. There were a lot of crashes that we could not debug, and essentially had to work around instead of fixing. Concluding, there are things you simply cant account for if you are not familiar with the DSP. But the problem was solved in the end and it works surprisingly well.

6 References

References

[1] Haykin S.,2014 *Adaptive Filter Theory*, Essex: Pearson Educational Limited