Algorithms in Signal Processors(EITN80) - application Adaptive line enhancer

Group:1
Peter Moodie - bts14pmo,
Sven Andersen - jap13san
Robin Sauer - dat14rsa

March 14, 2018

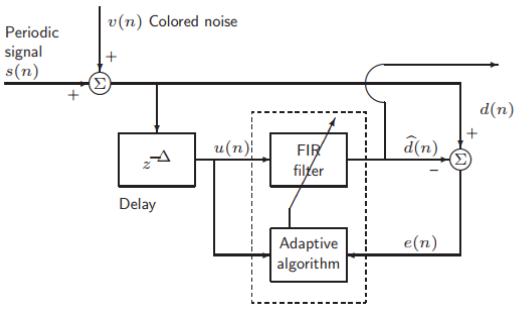

Figure 1: Adaptive Line Enhancer block diagram

Adaptive Line Enhancer (ALE)

In this report, signal separation will be illustrated through adaptive filter principles. An adaptive line enhancer strengthens the tonal component, which is periodic in nature, from a signal containing both the narrow-banded tonal component and a wide-banded signal. The wide-banded signal is usually called the noise. The line enhancer uses adaptive filters which automatically update the coefficients of the FIR filter. The signal $s(n)$ in this report consists of the sum of three sinusoidal waves with equal amplitudes of the frequencies $1000, 2000$ & $3000Hz$ which will be corrupted firstly with colored noise. The noise signal $v(n)$ in this case is a colored noise signal - music.

An ALE is based on the linear prediction which predicts the future value of a discrete time signal based on its past values. Our periodic signal can be perfectly predicted using its past samples. As seen in *figure* 1, our signal $s(n)$ is corrupted with noise $v(n)$ producing a new signal $d(n)$. Our new signal $d(n)$ is then fed into unit-delay-elements D. This delayed signal $u(n) = d(n - D)$ is used as an input signal to the adaptive filter which produces an output $\hat{d}(n)$. This output is then subtracted from the desired response signal, $d(n)$, giving an error signal. The periodicity of our signal means that $s(n)$ remains approximately (a shift in phase) the same while our noise component becomes uncorrelated.

The predictor can then only make a prediction about the sinusoidal component of our signal. Through the adaptive filter, we then obtain the signal $\hat{d}(n)$ which only contain the sinusoidal component. After subtracting $\hat{d}(n)$ from the desired signal $d(n)$ we will get an error $e(n)$ that contains the colored noise(music). Thus we have separated the two signals.

Methods

The algorithms chosen for this project are the LMS and NLMS algorithms.

The Least Mean Square (LMS)

With our cost function being the mean-squared error $J(w)$, the way to minimize our errors is by taking the derivative of our cost function which gives us our gradient vector. For the optimal coefficients, our gradient vector must be equal to zero. The LMS algorithm estimates the gradient vector, $\nabla(n) = -2\mathbf{p} + 2\mathbf{R}\mathbf{w}(n)$ by estimating the correlation matrix $\mathbf{R}$ and the cross-correlation vector $\mathbf{p}$ through:

$$\mathbf{R}(n) = \mathbf{u}(n)\mathbf{u}^H(n) \tag{1}$$

$$\mathbf{p}(n) = \mathbf{u}(n)d(n) \tag{2}$$

The LMS algorithm utilizes the gradient vector of the cost function to converge on the optimal wiener solution. Then with each iteration, the filter tap weights of the adaptive filter are updated according to the following formula:

$$e(n) = d(n) - \mathbf{u}^H(n)\mathbf{w}(n) \tag{3}$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathbf{u}(n)e(n). \tag{4}$$

The estimation-error seen in equation (3) and the tap-weight filter coefficient-update in (4).

Here $\mathbf{u}(n)$ is the input vector of the time delayed input values

$$\mathbf{u}(n) = [u(n) \quad u(n-1) \quad u(n-2) \quad \cdots \quad u(n-N+1)]^T. \tag{5}$$

The vector

$$\mathbf{w}(n) = [w_0(n) \quad w_1(n) \quad w_2(n) \quad \cdots \quad w_{N-1}(n)]^T \tag{6}$$

represents the coefficients of the adaptive FIR filter tap weight vector at time n.[1] The parameter μ is the step size parameter. The rate of convergence of the LMS algorithm depends heavily on the step-size parameter. A reduction in the step-size lowers the rate of convergence but also lowers the estimation error i.e. increases the accuracy. This step size parameter controls the influence of the updating factor. If this parameter is too large, then the filter becomes unstable and the output diverges. The mean-squared error per adaptation is given by $J(n) = E[|e(n)|^2]$. [3]

The Normalized Least Mean Square (NLMS)

The LMS algorithm is sensitive to the scaling of its input i.e. the input signal power makes it difficult to choose an appropriate μ value which guarantees stability. The normalized least mean square algorithm (NLMS) is an extension of the LMS algorithm which bypasses this issue by calculating maximum step size value which is calculated using the following formula.

$$\hat{\mu} = \frac{\mu}{\mathbf{u}(n)^T\mathbf{u}(n) + a} \tag{7}$$

This means a new step size is calculated for each iteration hence we need not know of the statistical properties of the signal prior to the execution of the algorithm. To lower the influence of the input signal amplitude on the gradient noise, the step size is scaled: is divided by the power of the input signal $u(n)$. In case the input signal is zero, a positive constant α in the denominator prevents the step size from being infinite. In equation (8) we see the new update of the filter coefficients.[3]

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \hat{\mu}e(n)\mathbf{u}(n) \tag{8}$$

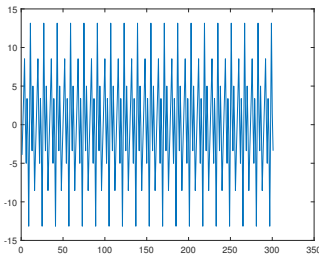

Figure 2: Sinusoidal input signal s(n)

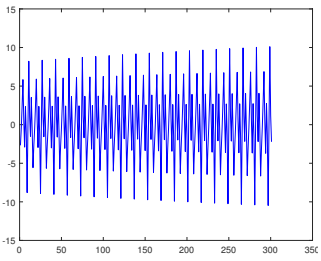

Figure 3: Filter output signal $d_w(n)$

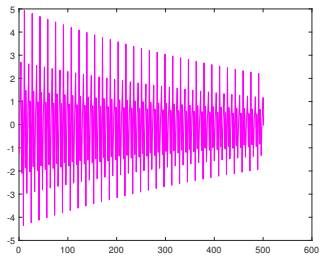

Figure 4: Error signal $e_w(n)$

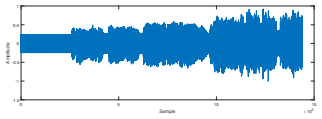

Figure 5: Input signal $u(n)$

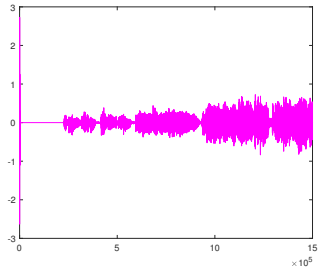

Figure 6: Error Signal $e_c(n)$

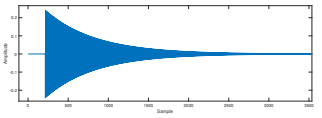

Figure 7: Initial transients of $e_c(n)$

Results

Configuration of the algorithms in Matlab

The parameters which provided a satisfactory result were chosen through initial runs and tweaks in MatLab. Before making the jump to implement our ALE on the DSP, a signal $s(n)$ corrupted with noise was created. As can be seen in *figure* 2, $s(n)$ contains 3 sinusoids combined with music. The aim was to filter out the tonal components which in our case is considered to be the signal of interest. This implementation can somehow be considered to be a Noise Cancellor, removing the music, in this considered to be the noise. The step size μ was selected to have a value of 0.01 and a protective constant, necessary to our NLMS algorithm, chosen as $a = 0.1$. Furthermore, our filter length was initially determined to be 300.

Using the NLMS algorithm, we were able to completely isolate the tonal component. In *figure* 3, the increasing amplitude of the signal is proof of convergence, if the window should be moved at a later time t_1, our signal will eventually reach the nominal amplitude of our original signal. In *figure* 4 the relationship is even more obvious, revealing the error signal decreasing as t_1 approaches the end of our signal.

Figure 5 shows the input signal to the filter, which is the music with sinusoidal noise. *Figure* 6 shows the error signal(music separated from noise) and *figure* 7 shows the first few samples of the error signal. Our filter had a large number of coefficients and a reasonably large step size leading to fast convergence. One can see the error had a large value in the beginning and then, as the filter proceeds to work, the error is minimized. On obtaining the coefficients which correctly reduce the error, the filter is able to extract our signal of interest with minimal error.

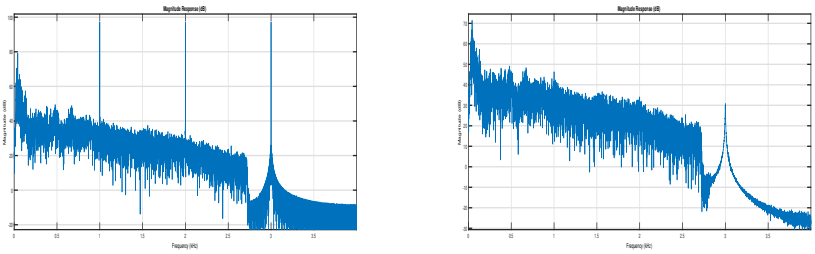

Figure 8: Magnitude response of the input & output signal respectively.

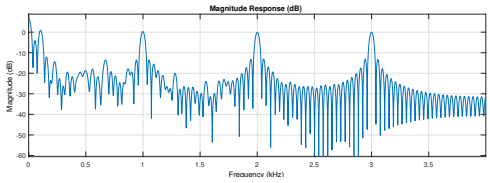

Figure 9: Magnitude response of the filter after convergence.

In *figure* 8 we can see the frequency response of the corrupted music before and after filtering. The spikes in the left figure represents the sinusoidal noise which our filter were able to eliminate(right figure). The frequency response of the filter after convergence can be seen in *figure* 9. The filter almost only lets through the frequencies corresponding to the sinusoids.

Discussion and Conclusions

Performance

If the filter length is too short, then the filter will be less precise. That means that frequencies close to the sinusoids that we try to remove will also be suppressed and this will affect the music. For that reason our original goal was to have a filter length of 300. We were unable to push the filter length past 220 when doing the implementation in the C programming language. This was a bit lower than our original aim of 300. What was holding us back was the computation time, as we had to do a number of computations proportional to the filter length. The goal was however reached when we did the implementation of the NLMS-algorithm in assembly. Almost all the computation time in the program takes place when we do convolution and updating of the filter. We were able to implement these two big for-loops with less half the number of instruction used by the assembly-code produced by the C-compiler. Hence we saw a big improvement after this and we are very satisfied with the final product.

Improvements

The NLMS-algorithm contains a division at every sample when normalizing the step-size with the power of the input signal. Since division is software emulated we are right now only doing it once in every 30 samples to save computation power. So our implementation is actually just an approximation of the real NLMS-algorithm. Further improvements would be to fully utilize the DSP by using both the memory banks and parallelize the assembly-code in the for-loops. If this optimization were done we would analyze if it is possible to increase the number of divisions, sampling rate and filter length to further improve quality.

Problems and experiences

Knowledge base

As we started up our project it was clear that we had the right knowledge base for the project, with a grading scale between signal processing and programming. Given that, a lot of the time we spent working we taught each other the part about the subjects that the others hadn't studied, as well as retutering our selfs. This gave rise to a surprising amount of problems, as we often assumed that the others knew what they were doing. And it took quite some time before we spoke in the same terms, as we looked at problems through different educational lenses, and conventions from different fields.

Programming

Although two of our three members had studied the C language, it was quite some time since either of us had used it, as such, much relearning was needed. Unfortunately, this was at the same point in time as we started to compile and run the program for the first time. And as we eventually learned, it was counterproductive to work on understanding the language and the VisualDSP program at the same time. It took quite some time until we were sure just what steps was necessary to upload a new program to the DSP. This, of course, caused major malfunction to our troubleshooting, as we made changes in the code but saw no changes on the DSP.

Troubleshooting

We never learned how to get any console information. And it took a long time until we fully understood what caused some of the effects that we were experiencing. One of our most recurring errors were that we got random noise that were clearly not the music that we had fed into the DSP, but it also clearly had the rhythm of the music. We were quite far into the project when we finally confirmed what it actually was. We thought we might have messed up with pointers, had type miss-match or implemented our algorithms wrong. But it turned out to be that we were too slow in processing the samples. As such we managed to process some samples, which caused us to hear the rhythm, but the remaining samples just became random noise.

References

[1] Haykin S, 2014 Adaptive filter theory, Essex: Pearson educational limited

[2] Adaptive Noise Cancellation, Aarti Singh

[3] Performance Comparison of Adaptive Algorithms for Adaptive line Enhancer