Active Noise Cancellation: Fundamentals and Application

Mark Goldwater and Lauren Anfenson

January 2, 2019

Abstract

Noise cancellation has many practical uses in commercial and work settings, making vehicles and headphones more pleasant for consumers or removing the loud drone of heavy machinery to protect the long-term hearing of workers. Many of these applications require a real-time adaptive filter to remove their target frequencies – these adaptive filters use an error minimization algorithm to update the weights of a filter to produce a desired response. There are several kinds of error minimization algorithms that can be used to achieve this, of which we explore the Least Mean Squared (LMS) algorithm. Applying these methods to background noise reduction in speech clips, we prove the effectiveness of the LMS algorithm in reducing white noise in a pseudo-real time implementation as well as propose improvements to our methods such as adding speech detection to our weight updating algorithm.

1 Introduction and Contextualization

Active noise cancellation aims to remove some sort of unwanted sound while leaving the signal intact, such as removing engine noise from the cabin of a vehicle or taking background noise out of a sound clip. This technology is used in many commercial applications, like remducing the sound of heavy machinery to protect the hearing of workers that are around that type of irritating low-frequency noise often. Cancellation can be accomplished in real-time applications by playing an "anti-noise" that destructively interferes with the target noise to effectively cancel it. However, due to the processing and hardware needs of real-time noise cancellation we will be focusing on a pseudo-real time implementation that uses post-processing of a sound clip to remove background noise with an algorithm that could be potentially scaled to real-time.

The key concepts behind any active noise cancellation implementation are adaptive filtering and error minimization. While typical filters are set to attenuate a certain range of frequencies that is constant during operation, an adaptive filter adjusts its impulse response over time to cancel noise as it changes. The ideal adaptive filter weights can be computed using the Wiener-Hopf equations to minimize error. However, these equations can only be applied when you know the entire signal before processing. So, when implementing active noise cancellation in any sort of real-time capacity, you need an algorithm that converges on a minimized error over time. We've chosen to focus on the Least Mean Squared (LMS) algorithm for this purpose.

We've chosen to center our project around the application of active noise cancellation in cleaning audio clips, using a two-microphone setup to record a noisy sound clip of someone speaking and some other sound clip of noise correlated with that from the speaking clip. In this paper we will be explaining the mathematical background and quantitative calculations needed to turn these two recordings into isolated speech as well as recounting the results of our implementation of these concepts.

2 Mathematical Background

2.1 Random Processes and Correlation

The fact that the signals we're working with can be thought of as random processes allows us to make assumptions that simplify the calculations of values like cross-correlation and autocorrelation. A more complete explanation of random processes and their statistics can be found in Appendix C, but we can briefly make the following statements about random processes from background [1]:

1. $X(t, e)$ is a family of functions which is traditionally called an *ensemble*.
2. A single function $X(t, e_k)$ is selected by the outcome e_k. This is just a time function that we could call $X_k(t)$. Different outcomes give us different time functions.
3. If t is fixed, say $t = t_1$, then $X(t_1, e)$ is a random variable. Its value depends on the outcome e.
4. If both t and e are given then $X(t, e)$ is just a number.

Consider $X(t_1, e)$ and $X(t_2, e)$ which are samples from the same time function at different times This pair of random variables can conveniently be written in terms of the doublet (X_1, X_2) which allows us to write the joint probability density function as $f(x_1, x_2, t_1, t_2)$. From this joint density function we can compute the autocorrelation of the function at these two times with the following equation.

$$R(t_1, t_2) = E[X_1 X_2] = \int \int_{-\infty}^{\infty} x_1 x_2 f(x_1, x_2, t_1, t_2) dx_1 dx_2 \tag{1}$$

When a random processes' statistics do not depend on time, they are called stationary and can be written only as a function of the difference in time between the signals being compared. Therefore, the autocorrelation and cross-correlation (between two different functions of time) functions of a process $X(t)$ (and $Y(t)$ for the cross-correlation) can now be written, respectively, as a function of τ, the difference between t_1 and t_2, as shown in Equations 2 and 3 below.

$$R_{xx}(\tau) = E[X(t)X(t+\tau)] = \int_{-\infty}^{\infty} x(t)x(t+\tau)dt \tag{2}$$

$$R_{xy}(\tau) = E[X(t)Y(t+\tau)] = \int_{-\infty}^{\infty} x(t)y(t+\tau)dt \tag{3}$$

We will take advantage of this property of stationary processes in adaptive filtering as we create an instantaneous estimation of Equations 2 and 3. The equations for autocorrelation and cross-correlation look very similar to that of of convolution.

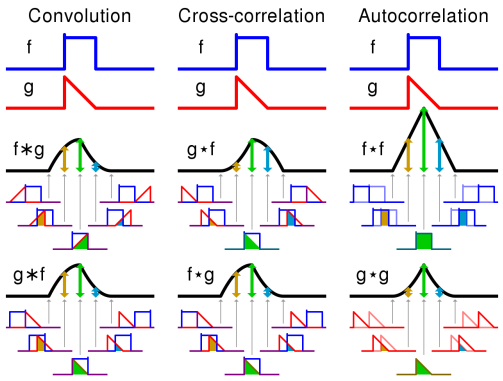

Figure 1: Visual representation of the differences and similarities between convolution, autocorrelation, and cross-correlation.

Figure 2 above shows the qualitative difference between taking the autocorrelation, convolution, and cross-correlation of a signal. We can see that there are only subtle differences that distinguish all three – we use cross-correlation and autocorrelation rather than convolution because these give us a quantative evaluation of the similarity between signals rather than how the shape of one signal is modified by the other.

2.2 The Discretized Wiener Filter

In signal processing, the Wiener filter is used to produce an estimate of a target unknown random process by linear time-invariant filtering of an observed noisy process **assuming known stationary spectra and additive noise**. The diagram below shows a basic diagram of an adaptive filter – we can see that it consists mainly of some filter with weights that are updated based on an adaptive algorithm. This algorithm usually acts on the error between the output of the filter and the desired signal, minimizing it. The LMS adaptive algorithm is based off of the Wiener filter and is a type of adaptive filter that descends an estimate of the Wiener filter exact gradient. We'll be using a Wiener filter with a finite impulse response, rather than an infinite impulse response, because it is more conducive to error minimization – a more thorough explanation of this can be found in Appendix A.

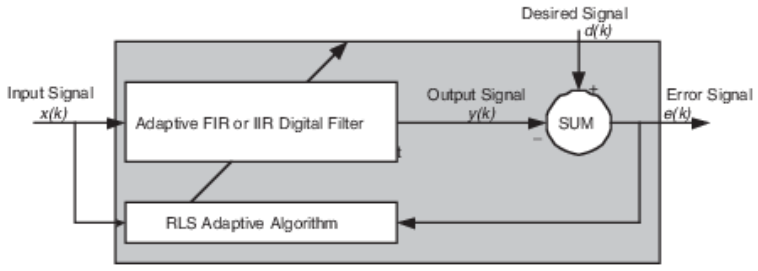

Figure 2: This is a block diagram of a basic adaptive filtering process. An input signal is filtered and produces an output with which an error is calculated by subtracting it from a desired signal. This error is then put through an adaptive algorithm which adjust the filter.

Let's dive into the exact optimal solution for a Wiener filter. Suppose we've sampled a signal

$x[n]$ and written its current value and its previous values in an M-dimensional vector, where M is the number of samples contained in the vector, or the size of the "window" of the signal we're currently evaluating.

$$x[n] = (x[n], x[n-1], x[n-2], ..., x[n-M+1])^T \tag{4}$$

The impulse response of the filter, then, can also be written in an M-dimensional vector. To get the output of the filter, $y[n]$, we must convolve the vector of samples of $x[n]$ with the impulse response of the filter. In our case this means operating on the signal with our filter weights, w, which is equivalent to using the impulse response directly.

$$y[n] = \sum_{k=0}^{M-1} w_k x[n-k] \tag{5}$$

The objective of applying the Wiener filter is to minimize the mean of the squared error between a desired signal, $d[n]$, and $y[n]$. The function that gives us the mean squared error between our desired and output signals is

$$J = E(d[n] - y[n])^2 \tag{6}$$

Where the function E is the mathematical expectation, a method of calculating a mean that takes into account the probability of each possible value of the signal. To minimize this function we substitute the equation for $y[n]$ into 6, solving for where the partial derivative of J with respect to its weights w is equal to zero. Because the error function is quadratic, there will be exactly one point where this is true. To find the partial derivative we go through the process detailed below

$$\begin{aligned}
\frac{\partial J}{\partial w_i} &= \frac{\partial J}{\partial w_i} E\left\{ \left(d[n] - \sum_{k=0}^{M-1} w_k x[n-k] \right)^2 \right\} \\
&= E\left\{ \frac{\partial J}{\partial w_i} \left(d[n] - \sum_{k=0}^{M-1} w_k x[n-k] \right)^2 \right\} \\
&= E\left\{ -2 \left(\frac{\partial J}{\partial w_i} \sum_{k=0}^{M-1} w_k x[n-k] \right) \left(d[n] - \sum_{k=0}^{M-1} w_k x[n-k] \right) \right\} \\
&= 2E\left\{ (-x[n-k]) \left(d[n] - \sum_{k=0}^{M-1} w_k x[n-k] \right) \right\} \\
&= -2E\{x[n-i]d[n]\} + 2 \sum_{k=0}^{M-1} w_k E\{x[n-k]x[n-i]\} \text{ for } i = 0, 1...M
\end{aligned} \tag{7}$$

The ideal solution of this equation is the set of weights w that satisfies the set of equations Eq. 6. Now, to further discretize and simplify this solution, we can recognize that the expectations in this equation are actually representing the cross and autocorrelation of our signals. Looking back at Eq. 2, we can see that $E\{x[n-i]d[n]\}$ is equivalent to the cross-correlation of the input signal and the desired signal. Let's call the vector resulting from this operation p which appears in matrix form as $p = (P_0, P_1..., P_{M-1})^T$. Likewise, Eq. 3 shows us that $E\{x[n-k]x[n-i]\}$ can

also be interpreted as the autocorrelation of the input signal. To represent the summation of the autocorrelation we can form the Toeplitz matrix of the autocorrelation vector, which we will call $\mathbf{R}$ and which will look something like this.

$$\mathbf{R} = \begin{bmatrix} R[0] & R[1] & \dots & R[M-1] \\ R[1] & R[0] & \dots & R[M-2] \\ \vdots & \vdots & \ddots & \vdots \\ R[M-1] & R[M-2] & \dots & R[0] \end{bmatrix}$$

The matrix is rectangular because we are summing over both k and i.
Rewriting 7 in terms of these new variables gives us

$$\begin{aligned} p &= \mathbf{R}w \\ w &= \mathbf{R}^{-1}p \end{aligned} \tag{8}$$

Which is the solution to the Wiener-Hopf equations that can now be solved for the ideal filter weights by simply multiplying each side by the inverse of the matrix $\mathbf{R}$.

It is also possible to solve the Wiener-Hopf equations for an unconstrained filter, an explanation of which can be found in Appendix B.

2.3 The Least Mean Squared Algorithm

In our aim to implement a pseudo-real time cancellation of noise, we can't use the solutions detailed above exactly because we won't know the entire signal in advance and will have to estimate the auto and cross-correlation values as we take more samples. Using the Least Mean Squared (LMS) algorithm, we can develop estimates of these values that improve continually over time until the algorithm converges on a value close to the true minimum. Figure 3 shows a visual of a two-weight filter descending the quadratic error surface, illustrating the difference between finding the exact gradient and using the LMS algorithm to estimate the gradient.

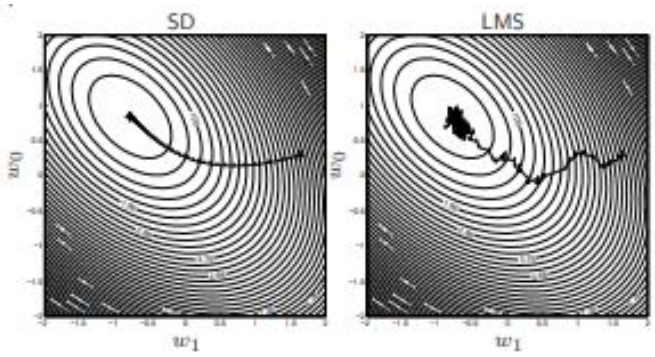

Figure 3: The left graph shows gradient descent using the steepest descent (SD) exact gradient calculation. On the right, a least mean squares (LMS) gradient approximation is used resulting in what appears to be Brownian motion down the quadratic surface.

Let's derive the algorithm for steepest descent. Looking back at our equation for the mean squared error, 7, we can substitute in with our new definitions of p and $\mathbf{R}$ and find the gradient

with respect to w. This results in the equation

$$\nabla_{w(n)} J(n) = -2\mathrm{P} + 2\mathrm{R}w(n) \tag{9}$$

Using this gradient to descend the error surface, we can find its minimum. By updating our weights with this gradient we can converge on the minimum error. The equation used to update the filter weights is given by

$$w(n+1) = w(n) + \mu(2p - 2\mathrm{R}w[n]) \tag{10}$$

Where μ is a constant that determines the step size that you use to traverse the gradient. The weights will be updated continually until the magnitude of the gradient is below some threshold. This parameter will have to be chosen when implementing the algorithm.

However, Eq. 9 is still using the exact values of the auto and cross-correlation. To form our estimates, we can write the autocorrelation and cross-correlation, respectively, as

$$\hat{r}_{xx} = x[n-k]x[n-i] \tag{11}$$

$$\hat{r}_{dx} = x[n-i]d[n] \tag{12}$$

This calculates the correlation values using only the information we have about the signal so far. Substituting these new definitions into our updating equation, we get

$$\begin{aligned}
\hat{w}(n+1) &= \hat{w}[n] + \mu(x[n-i]d[n] - \sum_{k=1}^{p} \hat{w}_k x[n-k]x[n-i]) \\
&= \hat{w}[n] + \mu(d[n] - \sum_{k=1}^{p} \hat{w}_k x[n-k])x[n-i] \\
&= \hat{w}[n] + \mu(d[n] - y[n]x[n-i]) \\
&= \hat{w}[n] + \mu e[n]x[n-i]
\end{aligned} \tag{13}$$

Now we can use Equation 13 in our algorithm to update the filter weights based on estimates of autocorrelation and cross-correlation that change and become increasingly accurate as we receive more information about the signal. In real-time implementations of active noise cancellation that are in use today, an algorithm similar to this – that updates estimates of the correlations and filters the signal in small windows of samples as it is received – is used to find the weights that cancel the target noise. A common improvement to this algorithm that is made is the addition of a "forgetting factor" that weights more recent windows more heavily when determining the current filter weights.

3 Mathematical Application

Now that we've gone through the mathematical background of our project, we can apply these concepts to our specific noise cancellation scenario. As stated in the Introduction, this scenario is a clip of speech with a noisy background with which we're aiming to cancel the background noise

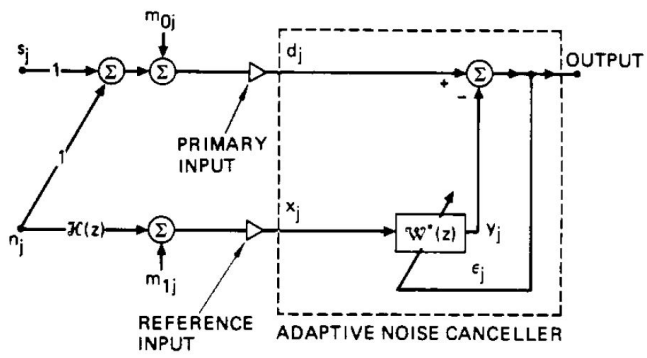

Figure 4: This is a block diagram illustrating how to implement adaptive filtering using an adaptive Wiener filter. s_j is the speech clip with noise, n_j is the isolated background noise that is operated on by some transfer function $X(z)$ before it is fed into the adaptive filter. The output of the system is the output of the adaptive filter subtracted from the noisy speech signal.

while retaining the speech. A block diagram of the system we're using to accomplish this with signal and noise sources labelled is shown in Figure 4.

The system in Figure 4 includes a signal that one wants to isolate (s_j), a noise correlated between the primary and reference noise (n_{0j} and n_{1j}) and uncorrelated noises (m_{0j} and m_{1j}). For the purpose of keeping the calculations clean and providing a clear understanding of this system, m_{0j} and m_{1j} will be ignored. These noises are caused by the vibration of electrons in the air close to the microphones and other things at this scale which are negligible for demonstrating the math behind the adaptive algorithm.

To apply an adaptive filter to this system, we adapt the filter weights to a small window of samples at a time n, filtering as we go. The filter weights, w, are initialized to a vector of zeros equal in size to our window of points. We operate on the window of the noise signal, u, with the current filter weights by multiplying the vector of weights and the vector of samples as shown below

$$y[n] = w(n)^T u(n) = \sum_{k=0}^{M-1} w_k(n) u(u-k) \tag{14}$$

This filter output is then subtracted from the speech signal to calculate the error.

$$e[n] = d[n] - y[n] \tag{15}$$

This error is then used to update the filter weights using the update equation derived from the LMS algorithm in Equation 13. The error is driven lower as the filter is adapted over each window of samples. Over time, ideally, this error converges to zero and must reconverge if there are changes in the system. The MATLAB code used to implement these steps can be found in Appendix D.

However, looking at Figure 4, it is not completely intuitive why minimizing the mean squared of the error would produce an output that contains the clean desired signal. Given that $d[n] = s[n] + n_0[n]$ the output of our error e is

$$e = s + n_0 - y \tag{16}$$

Squaring Equation 16 gives us

$$e^2 = s^2 + (n_0 - y)^2 + 2s(n_0 - y) \tag{17}$$

Taking the expectations of both sides of Equation 17, and realizing that s is uncorrelated with n_0 and y allows us to perform the following operations

$$\begin{aligned} E[e^2] &= E[s^2 + (n_0 - y)^2 + 2s(n_0 - y)] \\ &= E[s^2] + E[(n_0 - y)^2] + E[2s(n_0 - y)] \\ &= E[s^2] + E[(n_0 - y)^2] \end{aligned} \tag{18}$$

Since the signal power $E[s^2]$ is unaffected as the filter is adjusted to minimize $E[e^2]$ the minimum mean square error can be written as

$$\min\ E[e^2] = E[s^2] + \min\ E[(n_0 - y)^2] \tag{19}$$

Therefore when the filter is adjusted to minimize $E[e^2]$ the term $E[(n_0 - y)^2]$ is also minimized meaning that y is a best least squares estimate of the primary noise n_0. As a result, when the mean squared error is minimized the error of the system as seen in Equation 16 is s because $y \approx n_0$.

4 Experiment and Discussion

4.1 Experimental Setup

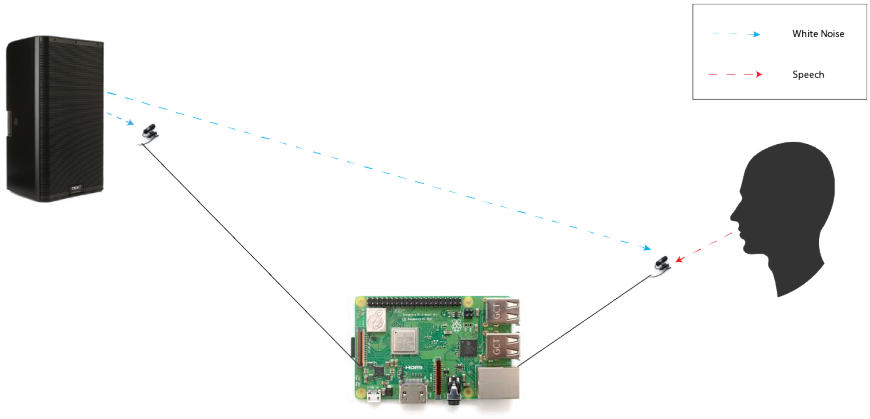

Figure 5: This image depicts our experimental setup. Two microphones recorded simultaneously on a Raspberry Pi. One picked up speech and white noise, while the other only recorded white noise.

To record our necessary input signals – the noisy speech signal and some correlated background noise – we used a stereo microphone setup that used a Raspberry Pi to record two channels of

sound simultaneously with a sample rate of 44100 Hz. The phase shift offset error was between the two signals is about one sample. We then played loud white noise out of a speaker[1]. Keeping the microphones as far apart as possible with the noise source about equidistant, we used one microphone to record someone speaking, making sure to have pauses in between speech to allow the filter weights to converge on the background noise. We used the second microphone to record just the white noise, being very careful to pick up as little of the speech as possible.

4.2 Experimental Results

Using the recording from the first microphone as d and the recording from the second as x which we split into windows u, we applied the noise reduction code detailed in Appendix D to the entire recording. Figure 6 shows plots of the speech signal before and after filtering.

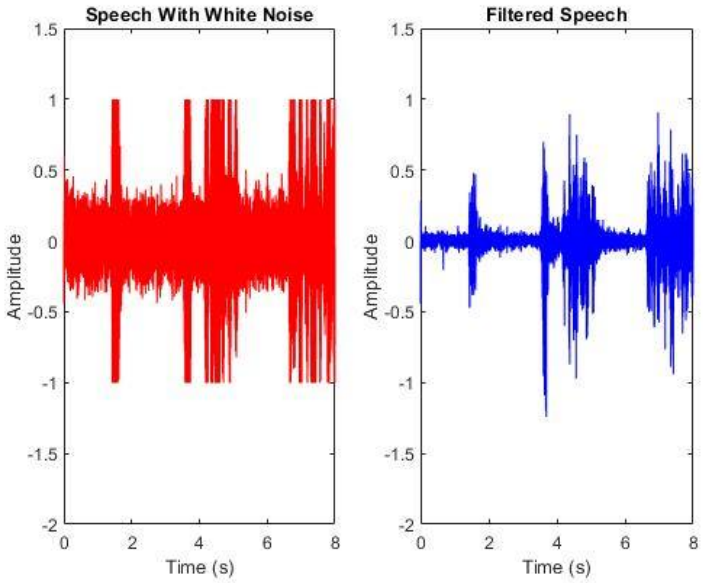

Figure 6: The plots above show a comparison of the unfiltered and filtered speech signal. We can see that magnitude of the noise between segments of speech has been considerably reduced.

We can see and hear that the background noise is noticeably reduced. However, it's not a perfect implementation. There's still a small amount of background noise that the filter can't remove, and the speech signal is attenuated slightly as well. We also noticed that our algorithm seemed to be more effective at removing low frequency noise than high frequency noise, which is somewhat expected because the filter would have more time to converge on relatively slow-changing low frequencies than on rapidly-switching high frequency noise. Moreover, we noticed that we actually produced noise out of the range of the recording capabilities of the microphones we were using. This is noticeably present in the unfiltered speech as no amplitude goes above one or below negative one. Below is the main code that we used to carry out the LMS algorithm.[2] The rest of the code can be found in Appendix D.

[1]Provided by Tony Nathan Estill.

[2]Note that the syntax of the code that allows this to be a method in a class have been removed for clarity.

```
w = zeros(order,1); % initialize weights vector to zeros

for n = order : length(d)
    u = x(n:-1:n-order+1);
    y(n)= w' * u;
    e(n) = d(n) - y(n);
    w = w + mu * u * e(n);
end
```

With every iteration, a small window u of the noisy input signal x is taken and has the filter weights applied to it, producing one sample of the output signal y. From here the error is taken, the weights are updated and then another window is taken by shifting the current one over one sample and the process repeats itself until it reaches the end of the input signal.

We then improved our noise reduction algorithm by adding voice activity detection so that we only adapt our filter weights on sample windows that don't contain speech. This helps because we want the output of the adaptive filter to closely resemble the background noise in the speech signal. Our measure of this similarity is the error term e, but when we try to subtract a filter output from a section of the speech signal that is not just background noise, we get an inaccurate measurement. This inaccuracy is then reflected in the update of the filter weights for that window. Because of this, the noise-reduced clip has a noticeable echo around segments of the clip that contain speech.

The algorithm works by taking the mean of the magnitude of the signal in the time domain for each window. We determined some threshold between the mean of a segment of noise and a segment of speech and said that the window contained speech is the mean was above that threshold. If speech is detected, then the step size for the update equation for that window of samples, μ, is set to zero so the filter weights do not adapt in response to speech segments. This improved both overall noise reduction and significantly reduced the "echo" that would occur after speech in our initial noise reduction attempts, as we can see in the figure below.

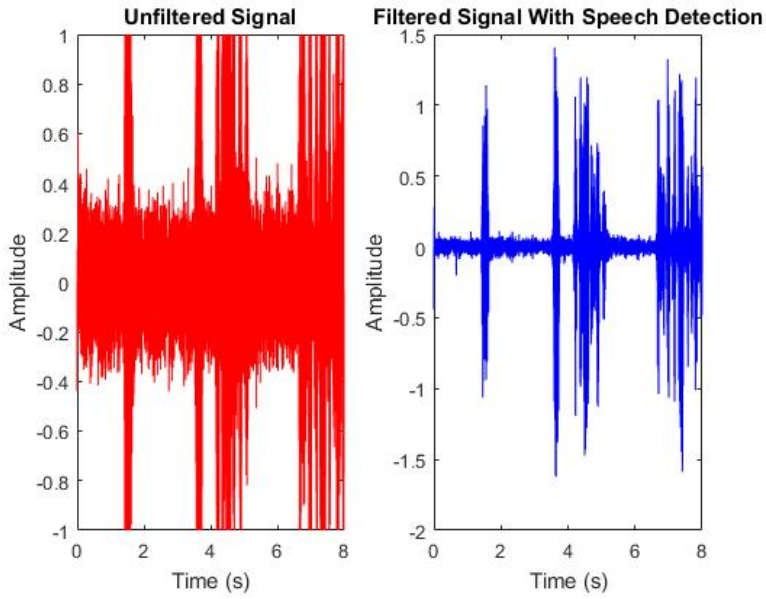

Figure 7: The plots above show a comparison of the unfiltered and filtered speech signal with speech detection. The background noise is even further reduced and the speech signal is less attenuated than it was in the noise reduction attempts without speech detection.

The code used to run this filtering can be found in Appendix E.

5 Conclusion

In summary, noise cancellation is widely applicable in commercial products, voice recording, and preserving the long-term health of workers because of its effectiveness in removing low-frequency, relatively constant background noise. It works on the principle of adaptive filtering, where a filter's weights are changed based on the error between the output of the filter and some desired signal. We've demonstrated many of the necessary components of active noise cancellation implementation in real-time: an adaptive filter that updates its weights using the LMS algorithm for error minimization that changes based on estimates of autocorrelation and cross-correlation that are updated as more samples are taken.

We've also proposed some ways to improve our initial implementation, such as adding speech detection to prevent the filter from adapting to speech and attentuating the signal we're trying to preserve. To move our project from pseudo-real time into true real time, however, even more improvements would have to be made. The speed of our algorithm would have to be greatly increased to keep up with the rapidly changing sound. However, our LMS algorithm shows that the filter can be adapted effectively without knowing the entire signal, suggesting that, given proper hardware, noise cancellation in real time using this method is potentially feasible.

References

[1] "Chap7-Random-Process.Pdf." Accessed December 4, 2018. https://www.cis.rit.edu/class/simg713/notes/chap7-random-process.pdf.

[2] "Chapter 5.Pdf." Accessed December 6, 2018. http://een.iust.ac.ir/profs/Farrokhi/Neural%20Networks/NNSH/chapter%205.pdf.

[3] Widrow, B., J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, R. H. Hearn, J. R. Zeidler, J. Eugene Dong, and R. C. Goodlin. "Adaptive Noise Cancelling: Principles and Applications." *Proceedings of the IEEE 63,* no. 12 (December 1975): 1692–1716. https://doi.org/10.1109/PROC.1975.10036.

[4] "Wiener_filter.Pdf." Accessed December 6, 2018. http://www.cnel.ufl.edu/courses/EEL6502/wiener_filter.pdf.

Appendix A FIR vs IIR Filter

In our research, we used a finite impulse response (FIR) Wiener filter as opposed to a digital filter with an infinite impulse response (IIR). In short, an FIR filter uses only current and past digital samples to obtain a current output sample value while an IIR filter uses the current output sample value, past input, and future output samples to obtain the current output sample value. We chose to use an FIR filter in our project because it is able to be adapted via the minimization of a cost function. In order to prove this the z-transform is utilized. The general equation for the z-transform is shown below, and can also be thought of as the discrete-time counterpart of the Laplace transform.

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{20}$$

where z is a variable with both a real and imaginary component. For convenience the z-transform of $x[n]$ is sometimes written as $\mathcal{Z}\{x[n]\}$ and the relationship between $x[n]$ and its z-transform is indicated as

$$x[n] \overset{\mathcal{Z}}{\Longleftrightarrow} X(z) \tag{21}$$

Keeping in mind the previously stated definitions of an FIR vs. IIR filter the difference equations for each can be written respectively as

$$y[n]_{FIR} = \sum_{k=0}^{M} b_k x[n-k] \tag{22}$$

$$y[n]_{IIR} = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=0}^{N} a_k y[n-k]_{IIR} \tag{23}$$

The FIR filter difference equation in Equation 22 only depends on the the last M outputs and their associated constant b_k while the IIR filter depends on the the last M outputs and their respective constants b_k and the next N outputs and their associated constants a_k (for k = 0, 1, 2,...,M).

In order to learn more about these systems let's take the z-transform of each and calculate their transfer functions. Starting with the FIR filter we get (note that the FIR and IIR subscripts have been removed for these calculations)

$$\begin{aligned}
\mathcal{Z}\{y[n]\} &= \sum_{k=0}^{M} b_k X(z) z^{-k} \\
Y(z) &= X(z) \sum_{k=0}^{M} b_k z^{-k} \\
H(z) &= \frac{Y(z)}{X(z)} = \sum_{k=0}^{M} b_k z^{-k}
\end{aligned} \tag{24}$$

The impulse response for an FIR filter is determined by the set of weights b. Because of the simplicity of this form, if one has a desired impulse response $H_d(z)$, one can calculate what the weights of an FIR filter must be in order to achieve this desired impulse response by, for example, minimizing the mean square error between $H_d(z)$ and $H(z)$. This is useful for a system that adaptively adjusts a filter for noise cancellation, for it can change its weights over time in order to produce the desired H_d. One does not have to train an error minimization algorithm on the entire signal which makes it ideal for an adaptive filtering implementation.

Carrying out the calculations for the impulse response of the IIR filter we get

$$\begin{aligned}
\mathcal{Z}\{y[n]\} &= \sum_{k=0}^{M} b_k X(z) z^{-k} + \sum_{k=0}^{N} a_k Y(z) z^{-k} \\
Y(z) - Y(z) \sum_{k=0}^{N} a_k z^{-k} &= X(z) \sum_{k=0}^{M} b_k z^{-k} \\
Y(z) \left(1 - \sum_{k=0}^{N} a_k z^{-k} \right) &= X(z) \sum_{k=0}^{M} b_k z^{-k} \\
H(z)_{IIR} = \frac{Y(z)}{X(z)} &= \frac{\sum_{k=0}^{M} b_k z^{-k}}{\left(1 - \sum_{k=0}^{N} a_k z^{-k} \right)}, \text{ given } a_0 = 1
\end{aligned} \tag{25}$$

This impulse response is more complicated than that of the FIR filter because the a weights in the denominator of the equation. If one wanted to create a cost function to minimize in order to bring $H(z)_{IIR}$ as close as possible to H_d you would end up with local maxes and mins and the one that an algorithm converges to may not be close to it. This is the reason why an FIR adaptive Wiener filter is best suited for a system that uses LMS.

Appendix B The Unconstrained Wiener Filter

Having the solution to the discrete version of the Wiener-Hopf equations, it would also be helpful to solve for the exact weights of an ideal Wiener filter continuously. The discrete convolution can more simply as

$$w(n) * R_{xx}(n) = R_{xd}(n) \tag{26}$$

This form of the Wiener solution is unconstrained, for the impulse response $w(n)$ may be causal or non causal as well as finite or infinite length to the left or right of the current time. In order to derive the transfer function of the Wiener filter one must first take the z-transform of the

autocorrelation matrix of the input signal which is that signals power-density spectrum. This is a measure of the distribution of signal energy over frequency.

$$\mathcal{S}_{xx}(z) = \sum_{n=-\infty}^{\infty} R_{xx}(n) z^{-n} \tag{27}$$

Therefore the cross power spectrum between the input signal and the desired response is

$$\mathcal{S}_{xd}(z) = \sum_{n=-\infty}^{\infty} R_{xd}(n) z^{-n} \tag{28}$$

The transfer function of the Wiener filter is then yielded by susbstituting into Equation 26 and exploiting the fact that convolution is multiplication in the z domain.

$$\mathcal{W}(z) = \frac{\mathcal{S}_{xd}(z)}{\mathcal{S}_{xx}(z)} \tag{29}$$

Appendix C Random Processes

In understanding the mathematical derivation and function of the Wiener filter, a significant concept is the expected value of random processes which in our case are random signals.

A random process is a rule that maps every outcome e of an experiment to a function $X(t, e)$ the domain of e is the set of outcomes of the experiment and the domain of t is a set, $\mathcal{T}$, of real numbers which is the real axis if $X(t, e)$ is a *continuous-time* random process or a set of $\mathcal{T}$ integers of the function $X(t, e)$ is a *discrete-time* random process. Note that we are also assuming that we know the probability density function for the set of e. In continuous time, the integral of of the probability density across some interval A is the probability that x is in said interval as written in Equation 30. This fact can be derived through the use of Bayesian statistics; however, as this is not completely relevant to this paper it will not be covered.

$$Pr(X \in A) = \int_A f(x)dx \tag{30}$$

Random processes whose statistics do not depend on time are called *stationary*. Random processes can have joint statistics and, if the process is stationary, they are independent of time shift. First order statistics are described by the first order cumulative distribution function $F(x, t) = Pr(X \leq x, t)$ is independent of time. Second order statistics are described by the joint statistics of random variables $X(t_1$ and $X(t_2$ with the second order cumulative distribution function: $F(x_1, x_2, t_1, t_2) = Pr(X(t_1) \leq x_1, X(t_2) \leq x_2)$. When the process is stationary, this function only depends of the separation of the signals ($\tau = t_2 - t_1$) but not the location of the interval.

Appendix D MATLAB Implementation: No Voice Detection

```
classdef LMSFilter
    properties
        mu % step size
        order % filter order
    end
    methods
        function obj = LMSFilter(mu,order)
            if nargin == 2
                obj.mu = mu;
                obj.order = order;
            else
                error('Missing Parameters: Be Sure to Enter Mu and Filter Order')
            end
        end
        function [y,e,w] = lms(obj,x,d)
            w = zeros(obj.order,1); % initialize weights vector to zeros
            for n = obj.order : length(d)
                u = x(n:-1:n-obj.order+1);
                y(n)= w' * u; % calculate one output sample
                e(n) = d(n) - y(n); % calculate error
                w = w + obj.mu * u * e(n); % update weights
            end
        end
    end
end
```

Appendix E MATLAB Implementation Without Sound Detection

```
classdef LMSFilter
    properties
        mu % step size
        order % filter order
    end
    methods
        function obj = LMSFilter(mu,order)
            if nargin == 2
                obj.mu = mu;
                obj.order = order;
            else
                error('Missing Parameters: Be Sure to Enter Mu and Filter Order')
            end
        end
        function [y,e,w] = lms(obj,x,d)
            w = zeros(obj.order,1); % initialize weights vector to zeros
            thresh = 0.25; % set mean threshold for speech detection
```

```
            for n = obj.order : length(d)
                u = x(n:-1:n-obj.order+1);
                y(n)= w' * u; % calculate one output sample
                e(n) = d(n) - y(n); % calculate error
                avg(n) = mean(d(n:-1:n-obj.order+1)); % find mean magnitude of window

                if avg(n) > thresh
                    w = w;
                else
                    w = w + obj.mu * u * e(n); % update weights if no speech detected
                end
            end
        end
    end
end
```

Appendix F GitHub Repository

The GitHub repository withe the code for this project can be found here.