Removing Echo

In this section we are given an audio file with an unwanted echo in it, and we discuss the methods used to remove the echo. First, we denote the desired audio file without the echo as s(n), D as the number of samples the echo is delayed from the original audio source, and a denotes the reflection coefficient representing the strength of the echo relative to the original sound source, where $0 \leq a \leq 1$. The file with the echo, x(n), can be written in the form:

(1) $x(n) = s(n) + as(n - D)$

Our task is to recover the original signal s(n).

(2) $s(n) = x(n) - as(n - D)$

To distinguish our recovered signal from the original sound source we let y(n) = s(n).

(3) $y(n) = x(n) - ay(n - D)$

To determine the appropriate filter, we take the z-transform of both sides to obtain:

(3) $Y(z) = X(z) - az^{-D}Y(z)$

(4) $Y(z) + az^{-D}Y(z) = X(z)$

(5) $Y(z)(1 + az^{-D}) = X(z)$

(6) $$\frac{Y(z)}{X(z)} = H(z) = \frac{1}{(1 + a^{z-D})}$$

Now we find the values of the a and D. To do this we'll be using the autocorrelation function in MATLAB. The autocorrelation function inputs a signal and outputs the correlation of each sample of the signal with itself as a function of delay. Naturally, the sample with zero delay will be the one with the highest correlation to itself, but since we know that there is a single echo in this file, the second largest value of autocorrelation will correspond to this echo, and the output of the function will tell us at which delay value the echo is occurring throughout the file. This is our value of D. Letting E{x} denote the expected value of x, the definition of autocorrelation (for our purposes) is:

(7) $R_{xx}(k) = E\{x(n)x(n + k)\}$

Using equation (1) to substitute for x(n) and x(n + k) we have:

(8) $R_{xx}(k) = E\{[s(n) + as(n - D)]*[s(n + k) + as(n + k - D)]\}$

Multiplying out the terms we have:

(9) $R_{xx}(k) = E\{[s(n)s(n + k) + as(n)s(n + k - D) + as(n - D)s(n + k) + a^2s(n - D)s(n + k - D)]\}$

Since the expected value of a sum is equal to the sum of the expected values, and since we can factor out constants from the expected value operator,

(10)

$R_{xx}(k) = E\{s(n)*s(n + k)\} + aE\{s(n)s(n + k - D)\} + aE\{s(n - D)s(n + k)\} + a^2E\{s(n - D)s(n + k - D)\}$

To simplify the third and fourth terms we take advantage of the following property of the autocorrelation function:

(11) $R_{xx}(k) = R_{xx}(-k)$,

which essentially says that the autocorrelation as a function of delay is symmetric, implying that it's the relative value of delay between any two samples that matters. So, shifting both of the input signals by D samples does not change the output of the function. Giving us:

(12)

$R_{xx}(k) = E\{s(n)*s(n + k)\} + aE\{s(n)s(n + k - D)\} + aE\{s(n)s(n + k + D)\} + a^2E\{s(n)s(n + k)\}$

which we can rewrite as,

(13) $R_{xx}(k) = (1 + a^2)R_{ss}(k) + aR_{ss}(k - D) + aR_{ss}(k + D)$

As we noted earlier, the highest values of autocorrelation will be $R_{xx}(0)$ and $R_{xx}(D)$. Plugging in 0 to equation (13) we get:

(14) $R_{xx}(0) = (1 + a^2)R_{ss}(0) + aR_{ss}(-D) + aR_{ss}(D)$

Since $R_{xx}(-D) = R_{xx}(D)$, we can rearrange this as:

(15) $R_{xx}(0) = (1 + a^2)R_{ss}(0) + 2a\ R_{ss}(D)$

Similarly, plugging in k = D, we get:

(16) $R_{xx}(D) = (1 + a^2)R_{ss}(D) + aR_{ss}(0) + aR_{ss}(2D)$

We now have two equations to represent the two values we'll be able to get from the autocorrelation function acting on x(n). However, we do not know any of the values of the autocorrelation function acting on s(n), but we can get around this with a simplifying approximation, namely that no matter what signal we're taking the autocorrelation of the value corresponding to zero delay will dominate. This gives us:

(17) $R_{xx}(0) = (1 + a^2)R_{ss}(0)$, and

(18) $R_{xx}(D) = aR_{ss}(0)$

Dividing equation (17) by equation (18), we get:

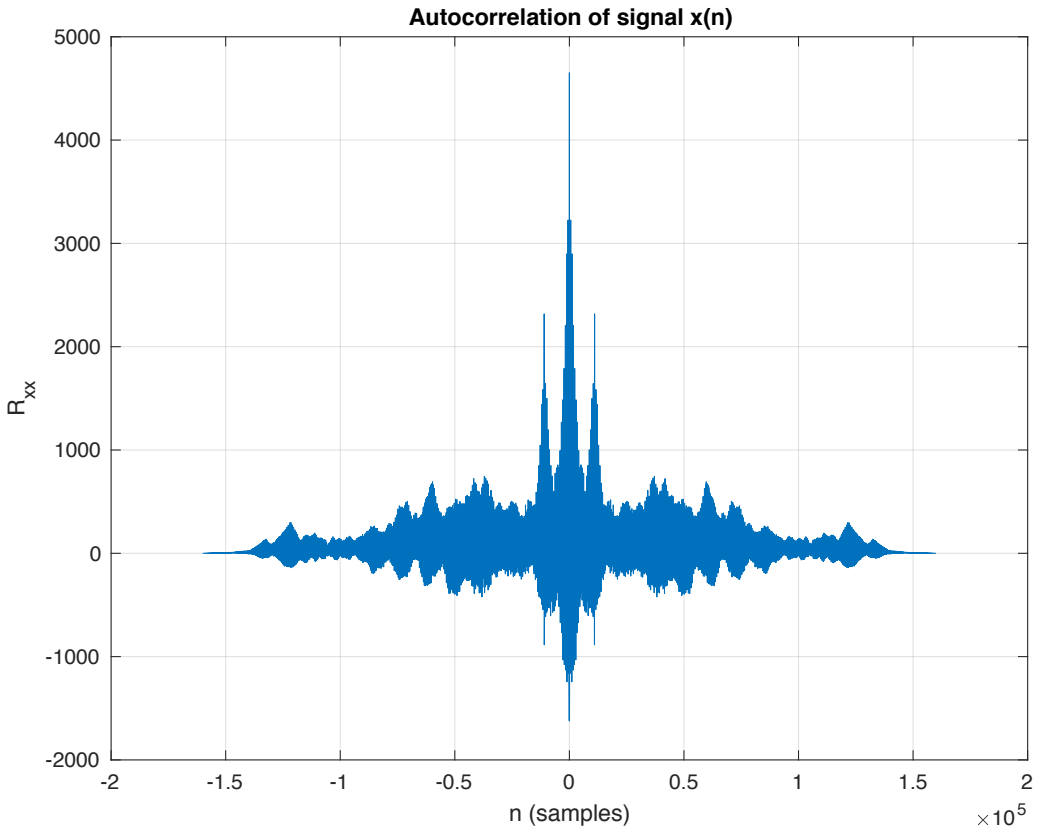
$$\frac{R_{xx}(0)}{R_{xx}(D)} = \frac{(1 + a^2)}{a}$$

(19)

Rearranging terms we find a solvable quadratic equation for a:

(20) $a^2 R_{xx}(D) - aR_{xx}(0) + R_{xx}(D) = 0$

Now we load the audio file into the vector x in MATLAB, compute the autocorrelation using the function xcorr, and graph it to approximate where Rxx(D) is. (See Appendix A.1 for MATLAB code).



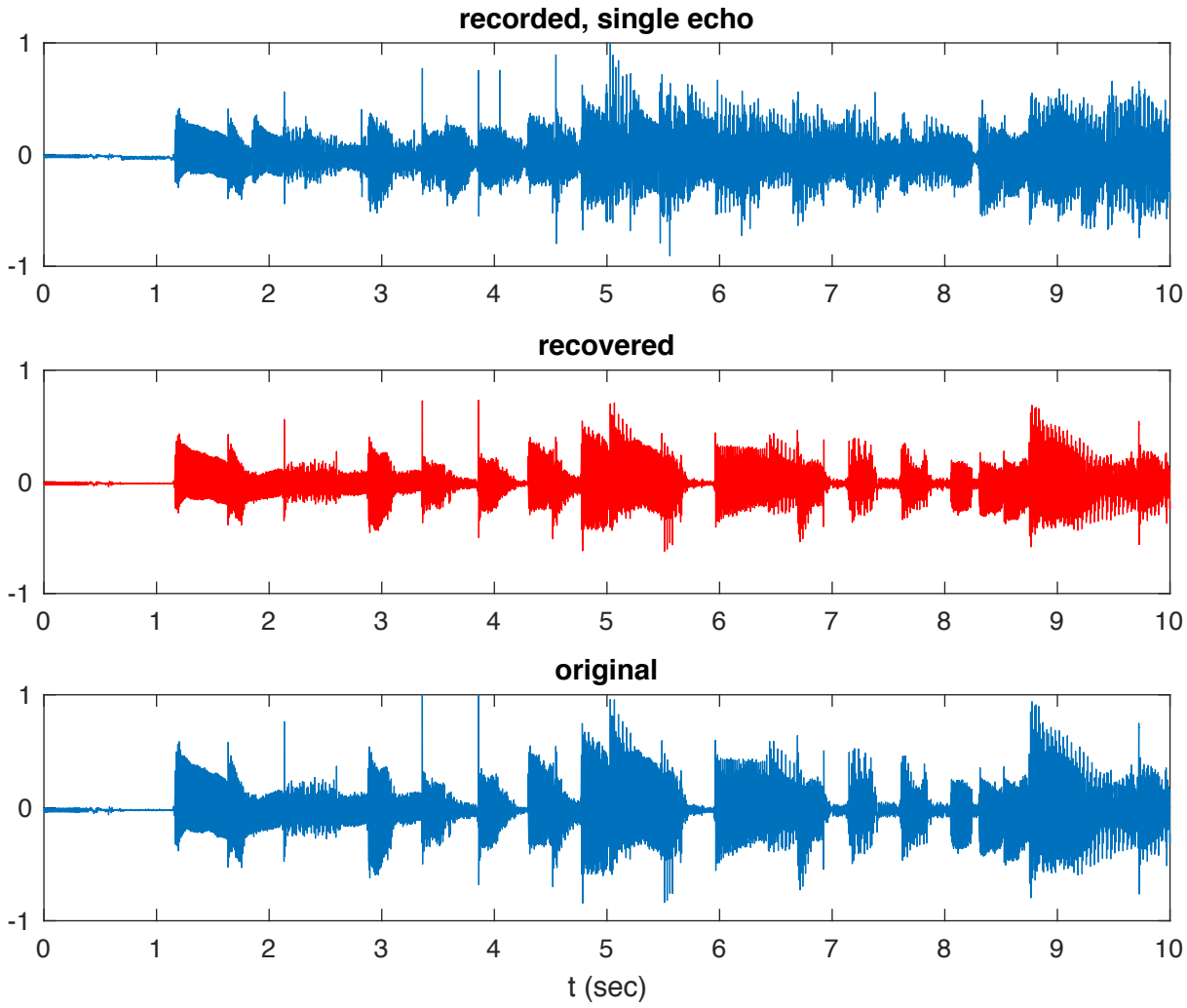
The maximum value, $R_{xx}(0)$, is about 4,651. Zooming in we can see that we should search the range between n = 8000 and n = 15,000. Doing so we find that Rxx(D) is about 2,318, verifying our assumption that $R_{xx}(0)$ dominates, and Rxx(D) occurs at a value of D = 11,000 samples. Plugging these values into equation (20) and solving for a, we get a = 0.9195 and a = 1.0875, but we know that a is a reflection coefficient, so we disregard the value greater than 1. We now have our transfer function of our inverse filter.

(21) $$H(z) = \frac{1}{(1 + 0.9195z^{-11000})}$$

In the time domain:

(22) $y(n) = x(n) - 0.9195y(n - 11000)$

To implement this we'll use the following sample-by-sample processing method. We know that we need to keep the current sample of x(n) along with the most recent 11,000 samples of the output to compute each current sample of the output. So we'll create a vector, w(n), of size 11,001 in memory and initialize it to zero for the computation of the first 11,000 samples. If we were to use the same memory location for each input sample and the same memory location to read the last sample, this would require 11,001 data shifts plus multiplication and addition (subtraction) for each sample. This can become impractical for signals with large delay, even for the fastest processors. So, we'll use what's known a circular delay line buffer. We'll still need the same amount of space in memory, but to avoid shifting all of the data by one register for each input sample, we'll just use a pointer (index variable in MATLAB) to move along the memory as we fill it with data, moving the pointer by one register for every input sample, and using another pointer (index variable) to access the sample with a delay of D = 11,000 relative to each sample we've just input. Of course, once we reach the end of the array we'll need wrap back around to the beginning to remain within the bounds of the array, hence the term circular delay line. We'll actually be decrementing our pointers, so it can be though of as moving backwards through the array. Once we've processed each output sample we can overwrite its 11,000th delayed version with new data. This process repeats until we move through the entire input file and calculate the output sample by sample. The code is shown in Appendix A.1, and results of running the signal x(n) through this filter to recover the original signal are shown on the next page.



As we can see, the recovered audio is practically the same as the original, By listening to the original and recovered audio, we hear that any small differences are inaudible.