Matthew Bremer
11 December 2020
ECE 551

Active Echo Cancellation using Frequency-Domain Spline Adaptive Filters

Abstract

The problem of active echo cancellation in VoIP programs has become of particular interest in the last few years. In this paper, a solution for Active Echo Cancellation using efficient nonlinear adaptive filters called Frequency Domain Spline Adaptive Filters is proposed, analyzed, and tested.

Introduction

Since the invention of the internet, a significant amount of business and social communication has moved online, including both text communication and Voice over IP (VoIP). In order for VoIP to be effective, people communicating must be able to clearly understand each other. This comes with several technical challenges. One of these is the problem of echo, which has existed in audio communication devices such as phones for a long time. However, this problem now must be solved for VoIP, in software. In this paper, we will examine the problem of Active Echo Cancellation, including what makes it so important and what properties a good solution should have. We will then introduce and evaluate a proposed solution using the novel Frequency-Domain Spline Adaptive Filter.

Motivation

In the year 2020, the SARS-CoV-2 virus has impeded the ability of many to continue working as normal. In a June 2020 interview with Stanford News, Stanford economist Nicholas Bloom reported that 42% of Americans were working from home at that time[1]. Even before the 2020 pandemic, the trend was clear: From 2005 to 2017, the number of people working from home in the United States grew by 159%[2]. Along with these workers, many K-12 students have had to begin learning online, using VoIP programs such as Zoom. This is important to note as disadvantaged children are likely to have significantly underpowered computers, and are still required to run a VoIP program, along with potentially other processor-intensive software such as a web browser. This means that an ideal VoIP program is as lightweight as possible, in order to be run smoothly and consistently on out-of-date hardware.

There are many technical challenges associated with VoIP software, with many focused on quality of service - the ability of each person on the call to understand the others. One of the quality-of-service issues is that of echo. In a general sense, an echo is just a copy of someone's audio input that is streamed back into their output with some changes (these changes will be specified in a later section). This is caused by leakage from one user's output device (i.e. headphones or speakers) into their input device (microphone). This problem is more common when using speakers such as built in laptop and phone speakers, but may also occur when someone uses headphones with poor isolation in close proximity to their microphone.

It is extremely important that the problem of echo be addressed in modern VoIP software. It is impossible to ensure that every member of a voice call will have an ideal setup with no

leakage, and so the possibility of echo must be resolved within the VoIP software. Echo can make it hard for someone to be understood, as their echo may overlap with their speech. However, a much bigger problem is that of speech jamming - when someone has a delayed copy of their speech fed back to them, it makes it extremely hard to speak. An example of this effect can be found at [3], where Leffen, the speaker on the left, has been "speech jammed", interfering with his speech.

Active Echo Cancellation

In order to remove echo from a call, each user's local output must be filtered in some way involving their local input. This is the process known as Active Echo Cancellation (AEC). Because of the nature of VoIP software, this filtering must be done in real time. This is one of the reasons that an ideal AEC algorithm must be as efficient as possible - to minimize signal delay and processor usage.

In order to understand the nature of an ideal filtering solution, we must first formally define the echo that needs cancelling. For the scope of this paper, we will assume echoes are generated in the following way:

1. User A's local input ("Far-end" signal) is received by user A. This may be the output of a single User B, or the linear combination of multiple users (User B, User C, etc.).
2. A copy of User A's input is created.
3. The copy is delayed.
4. The copy is filtered.
5. The copy is linearly added to User A's regular speech output.

An important note is that due to the nature of leakage from output to input, it is possible that the filtering done in step (4) is nonlinear in nature. For example, if User A's speaker's clip when playing their output, the echo will be clipped as well. It is also possible for a noise gate to affect the input such that it only plays back the audio peaks. In summary, an echo is a delayed, (potentially nonlinearly) filtered copy of a user's input added to their output.

Now that an echo is clearly defined, we can introduce the general AEC solution. Given that no two people sound exactly alike, and that no two echoes have exactly the same qualities, it may seem obvious that the solution involves adaptive filtering. Indeed, the solution uses an adaptive filter to replicate the delaying and filtering process on the far-end input.

The general scheme is as follows:

- The far-end signal serves as the input to the adaptive filter
- The near-end signal (Local user's speech with added echo) serves as the desired output of the filter

These two inputs along with the proper filter adaptation will have the following results:

- The output of the adaptive filter will mimic the delay and filtering effects present in the echo

- The error function of the adaptive filter will contain the near-end signal with the echo cancelled

Finally, the problem of AEC is laid out in full: the solution requires an adaptive filter that is efficient (since it is constantly being used on consumer hardware) and can perform real-time nonlinear filtering. This system is shown in Figure 1.

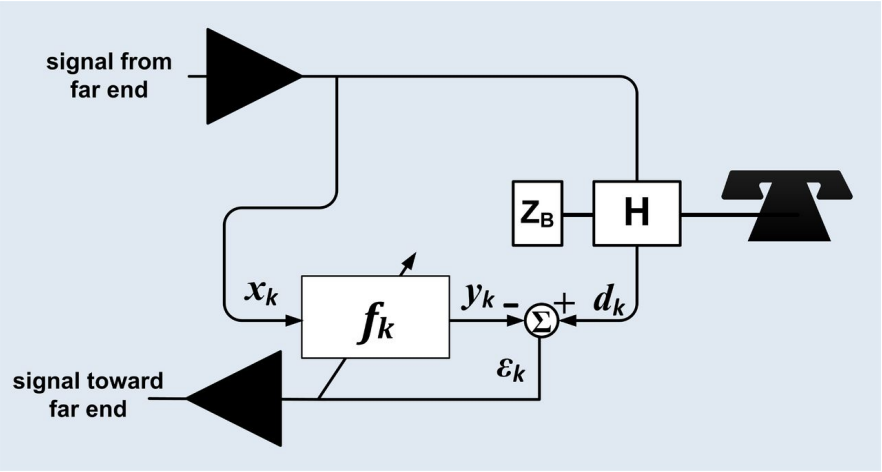

Fig. 1 *Constant314, CC0, via Wikimedia Commons*

Frequency-Domain Spline Adaptive Filter

One solution that is both capable of nonlinear filtering and performance oriented is the Frequency-Domain Spline Adaptive Filter, or FDSAF[4]. We begin with a description of the Weiner Type Spline Adaptive Filter (SAF)[5] on which the FDSAF is based.

The Spline Adaptive Filter is an adaptive filter capable of replicating nonlinear systems. It accomplishes this by first passing the input signal through an FIR Weiner filter, and then passing the filtered output through a nonlinear spline interpolator. This system is shown in Figure 2. The spline interpolation works using a Look-up table or LUT of spline function knots, or points in a 2D function. For each input, the four closest knots are calculated, and the output values are outputted from the LUT. This gives the vector $\mathbf{q}_i$. The value u is also calculated. This is the distance from the input value to the nearest control knot that is lower or equally valued. Then the vector $\mathbf{u} = [u^3\ u^2\ u\ 1]$ is calculated, and finally the interpolated value becomes $\varphi_i(t) = \mathbf{uCq}_i$, where $\mathbf{C}$ is a 4x4 matrix representing the interpolation function itself. For the FDSAF, the Catmul-Rom spline interpolation function was used.

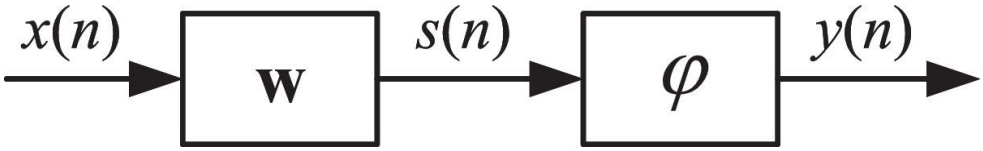

Fig.2 *[4]*

The FDSAF is based off of the Weiner Type SAF, with one key performance enhancement - the FIR filtering done in the first step is instead done in the frequency domain. Instead of performing time domain convolution, the filter instead maintains frequency-domain filter weights and filters the input in the frequency domain using element-wise multiplication. In order to do this, the input is first transformed into the frequency domain with a Fast Fourier Transform (FFT), and after the filtering, is transformed back into the time domain with an Inverse Fast Fourier Transform (IFFT). From there, the spline filtering and adaptation is done in the time domain, and the error signal is transformed into the frequency domain in order to adapt the frequency domain filter weights. A diagram of this system is shown in Figure 3.

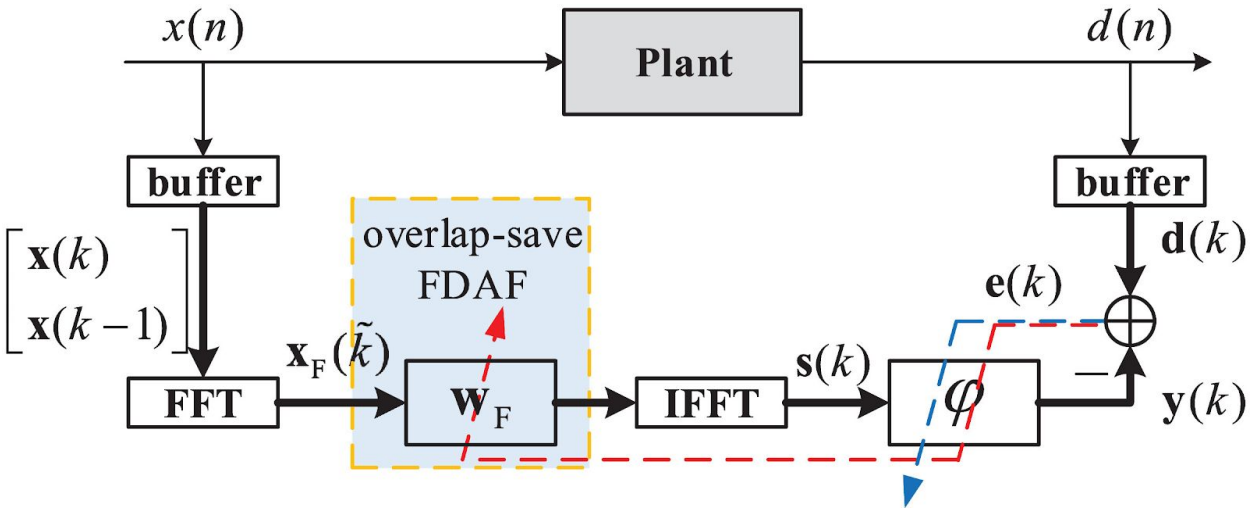

Fig 3. [4]

The reason that the FDSAF is so well suited to AEC is because of how it saves processing power. For an FIR filter with *M* taps, time domain convolution filtering takes $O(M^2)$ multiplication operations. However, frequency domain filtering only uses $O(2M\log M + M)$ multiplication operations. For some filtering problems, the decrease in algorithm complexity wouldn't be worth the tradeoffs of using frequency domain filtering. However, when discussing echoes, it isn't easy to define an upper bound on the echo delay. The echo delay time has many factors, both physical and digital. Even if we set an upper bound of 100ms delay and 20 tap FIR filter, at CD quality audio (44.1kHz sampling rate) our filter needs to be 0.1*(44100) + 20 = 4430 taps long. At such a large amount of needed taps, the algorithmic complexity decrease from using frequency domain filtering makes a huge difference.

The large challenge that arises from frequency domain filtering is that samples can no longer be processed one at a time - in order to reap the rewards from frequency-domain filtering, the input must be processed in chunks of samples (each with a length the same as the FIR filter length). For the most part, this simply means performing spline interpolation in batches with large matrices. In fact, this can lead to further performance enhancements on multiprocessor systems, due to the parallel nature of matrix multiplication. However, this may intuitively lead to

a concern about the learning rate of the program. This is because the filter weights and spline knots are updated per-chunk. This means that by the time we reach the end of a data chunk of length *M,* the filter hasn't had any adaptation from the previous *M-1* samples. However, according to [4], the learning rates for each filter are entirely equivalent. During testing, no discernable difference between the two algorithms' learning rates could be found.

Works Cited

1. Wong, May. “Stanford research provides a snapshot of a new working-from-home economy.” *Stanford News*, 29 June 2020, https://news.stanford.edu/2020/06/29/snapshot-new-working-home-economy/.
2. Reynolds, Brie W. “159% Increase in Remote Work Since 2005: FlexJobs & Global Workplace Analytics Report.” *FlexJobs*, 29 July 2019, https://www.flexjobs.com/blog/post/flexjobs-gwa-report-remote-growth/.
3. GRsmash. “Leffen has a stroke at BAM7.” *Youtube*, 25 May 2015, https://www.youtube.com/watch?v=UWcBMFI4FaQ
4. Liangdong Yang, Jinxin Liu, Qian Zhang, Ruqiang Yan, Xuefeng Chen. "Frequency domain spline adaptive filters.” *Signal Processing*, Volume 177, 2020, 107752, ISSN 0165-1684. http://www.sciencedirect.com/science/article/pii/S0165168420302954
5. Michele Scarpiniti, Danilo Comminiello, Raffaele Parisi, Aurelio Uncini. “Nonlinear spline adaptive filtering.” *Signal Processing*, Volume 93, 2013, ISSN 0165-1684. https://doi.org/10.1016/j.sigpro.2012.09.021.

Appendix A: Results

In order to verify the functionality of the FDSAF, I replicated Example 1 from [4] with the following results: Figure 4 shows that the spline adaptation was successful, as the result spline closely matches the target spline. Figure 5 shows that the resulting FIR filter's frequency response comes to closely match that of the specified IIR filter, although with some inaccuracy in the phase response. Most importantly, we can see that the Mean Square Error (Fig. 6) converges quickly to 0, with little variation after 20000 samples.

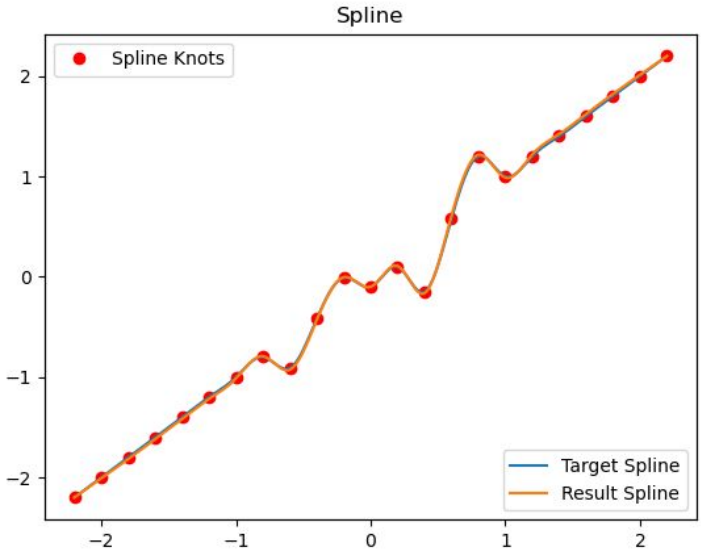

Fig. 4

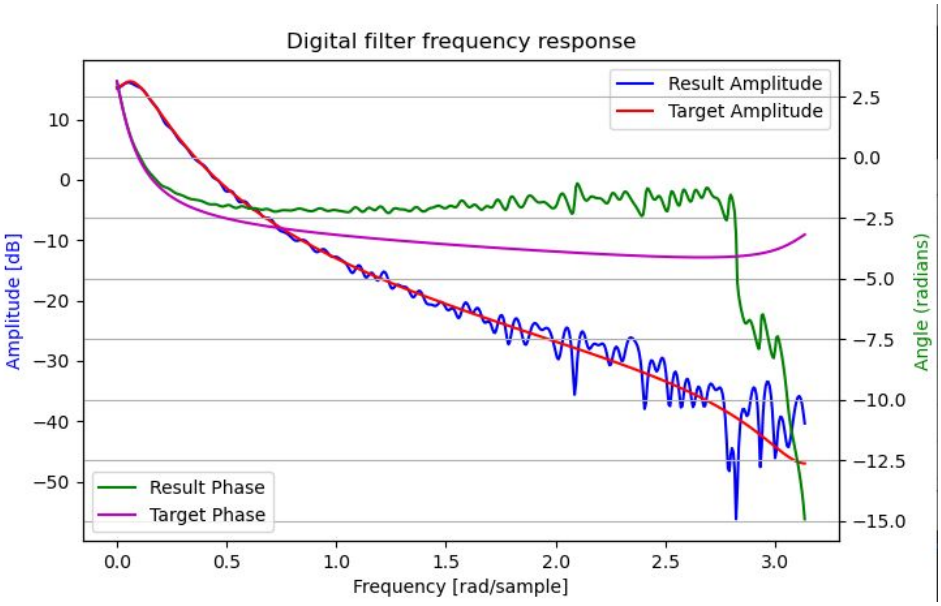
Digital filter frequency response
Result Amplitude
Target Amplitude
Result Phase
Target Phase
Amplitude [dB]
Angle (radians)
Frequency [rad/sample]
10
0
−10
−20
−30
−40
−50
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Fig. 5

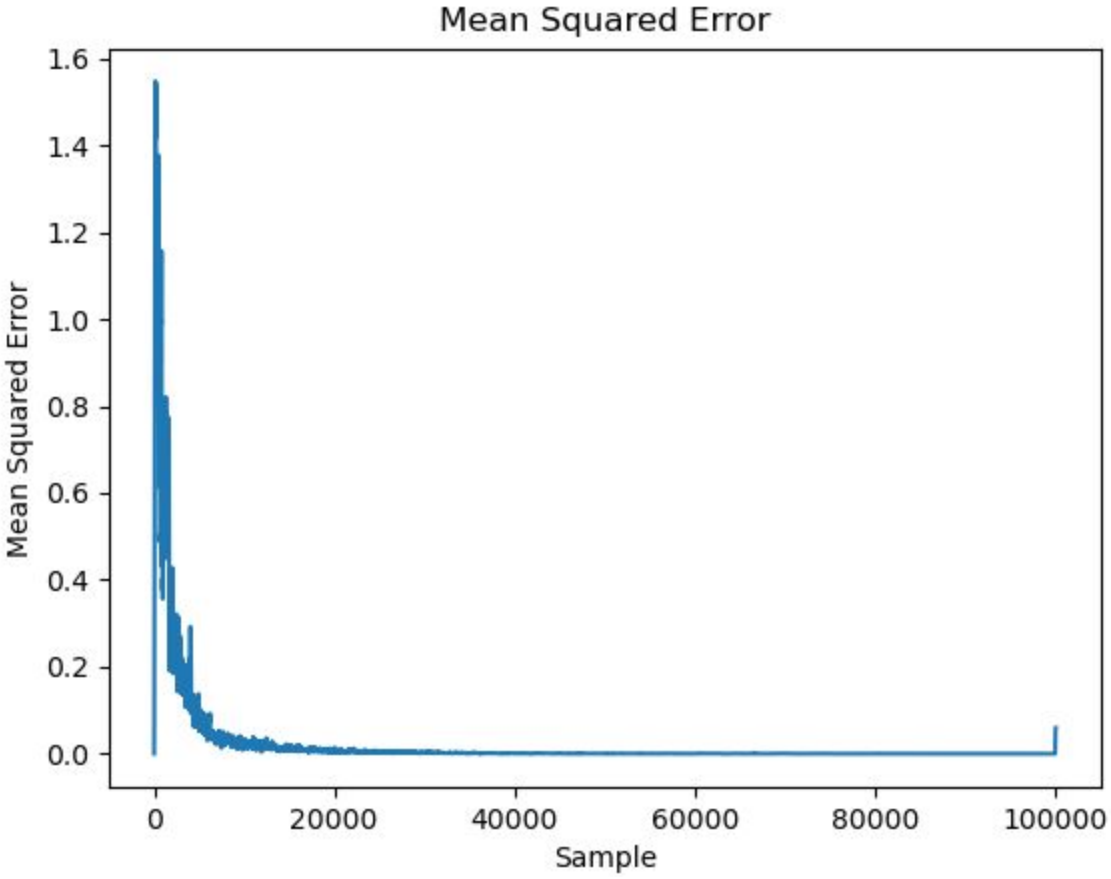
Mean Squared Error
Mean Squared Error
Sample
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000

Fig. 6

Next, I tested the FDSAF in an Active Echo Cancellation context. Using a recorded phone conversation from the CallHome Corpus, I simulated an echo with a 100ms delay time and -6dB amplification. Figure 7 shows the artificially added echo (top track) versus the residual echo extracted from the system output (bottom track). Visually, it is easy to see that while the echo hasn't been entirely cancelled, it has been significantly attenuated. In Figure 8, the Echo Return Loss Enhanced (The dB ratio of added echo to residual echo) is shown in red, overlaid with the echoed speaker's speech in blue. When the speaker is talking, the ERLE is largely positive, meaning the echo is being attenuated.

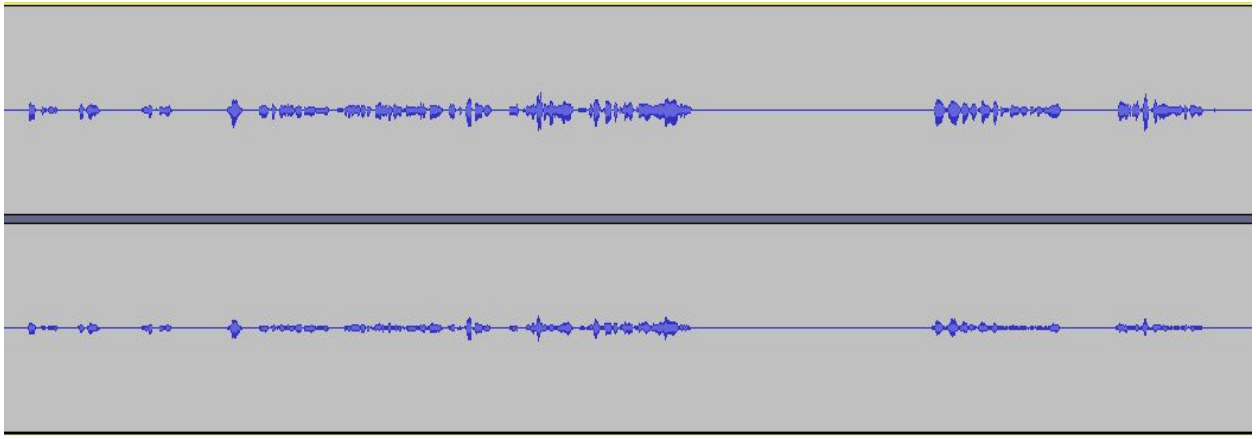

Fig. 7

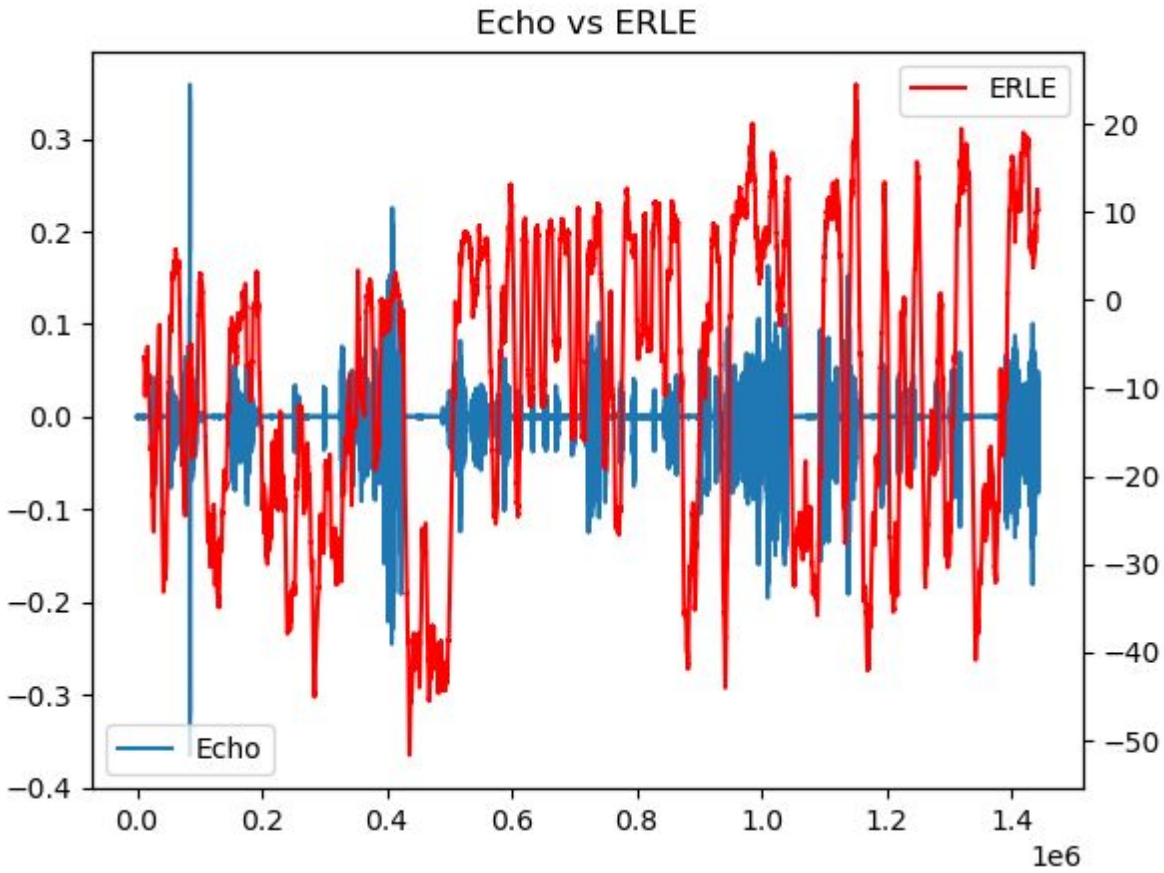

Fig. 8

Finally, performance of the FDSAF was analyzed. 3 minutes of 8kHz audio was fed through the above echo cancellation scheme with a filter length of 1200 taps, first by a normal SAF and then by an FDSAF. This is to show the difference between the FDSAF and another scheme with similar functionality that performed time-domain filtering. The FDSAF took a total of 31.3 seconds to filter the entire audio stream, while the SAF took 85.0 seconds. This shows that the FDSAF is extremely efficient compared to a similar time-domain scheme.

Appendix B: Code

Python Imports (All Examples)

```
import numpy as np
from scipy import interpolate,signal
import matplotlib.pyplot as plt
import math
from scipy.io import wavfile
```

FDSAF / SAF Hybrid Class

Notes:
The function *est()* performs FDSAF using the frequency domain filter weights *w_f*
The function *estsingle()* performs SAF using time domain filter weights *w*
The functions *estsinglefilter()* and *estsinglespline()* perform the individual parts of SAF, FIR and spline filtering respectively. (These are for debugging purposes)

The T suffix means Transpose
The m suffix means matrix
Variables are named following the literature in [4], however there are some mistakes (i.e. deltadot is meant to mean phidot)

```
class FDSAF:
    def __init__( self,filterlen):
        self.M = filterlen
        #self.w_f = np.fft.fft(np.concatenate((np.ones(1),np.zeros(2*self.M-1))))
        self.w_f = np.fft.fft(np.zeros(2*self.M))
        #self.w_f =
np.fft.fft(np.concatenate((np.ones(self.M)/self.M,np.zeros(self.M))))
        self.last_buffer = np.zeros(self.M, dtype='float')
        self.Cm = np.matrix(0.5 * np.array([[-1,3,-3,1], # Row major
                                [2,-5,4,-1],
                                [-1,0,1,0],
                                [0,2,0,0]]))

        # Based on paper example 1
        self.mu_w = 0.001
        self.mu_q = 0.001
        self.delta_x = 0.2
        self.ordinates = np.arange(-2.2,2.3,self.delta_x)
        self.abscissas = np.arange(-2.2,2.3,self.delta_x) # Only for graphing

        self.N = len(self.ordinates)-1
        self.e = 0

        # For sample-wise filtering
        #self.w = np.concatenate((np.ones(1),np.zeros(self.M-1)))
        self.w = np.zeros(self.M)
        self.single_buffer = np.zeros(self.M)
```

```
def est(self, x, d):
    if len(x) != self.M or len(d) != self.M:
        print("Wrong input length")
        exit()

    full_buffer = np.concatenate((self.last_buffer,x))

    x_f = np.fft.fft(full_buffer)

    s = np.fft.ifft(x_f*self.w_f)[-self.M:]

    UT = []
    UdotT = []
    QT = []
    IT = []

    for j in range(self.M):
        i_j = int(np.floor(s[j].real/self.delta_x) + (self.N/2))
        u_j = s[j].real/self.delta_x - np.floor(s[j].real/self.delta_x)
        u = [math.pow(u_j,3),math.pow(u_j,2),u_j,1]
        udot = [3*math.pow(u_j,2),2*u_j,1,0]

        if (abs(s[j])>2.0):
            q = np.asarray([(y-11)*self.delta_x for y in range(i_j-1,i_j+3)])
            IT.append([-1,-1,-1,-1])
        else:
            q = np.asarray(list(self.ordinates[i_j-1:i_j+3]))
            IT.append([i_j-1,i_j,i_j+1,i_j+2])

        UT.append(u)
        UdotT.append(udot)
        QT.append(q)

    Um = np.matrix(UT).T
    Udotm = np.matrix(UdotT).T
    Qm = np.matrix(QT).T
    Im = np.matrix(IT).T

    y = np.matmul(self.Cm,Qm)
    y = np.multiply(y, Um)
    y = np.asarray(y)
    y = np.sum(y, axis=0)

    self.e = d - y

    deltadot = np.matmul(self.Cm,Qm)
    deltadot = np.multiply(deltadot,Udotm)
    deltadot = np.asarray(deltadot)
    deltadot = np.sum(deltadot,axis=0)
```

```
        e_s = np.multiply(deltadot/self.delta_x,self.e)
        e_f = np.fft.fft(np.concatenate((np.zeros(self.M),e_s)))
        deltaW = np.fft.ifft(np.multiply(e_f,np.conjugate(x_f)))[:self.M]

        self.w_f = self.w_f + self.mu_w *
np.fft.fft(np.concatenate((deltaW,np.zeros(self.M))))

        temp = np.asarray(np.matmul(self.Cm.T,Um))
        deltaq = self.mu_q * self.e * temp
        Qm = Qm + deltaq
        Qm = np.reshape(Qm,-1)
        Im = np.reshape(Im,-1)
        deltaq = np.reshape(deltaq,-1)

        for i in range(np.shape(Im)[1]):
            if Im[0,i] != -1:
                self.ordinates[Im[0,i]] += deltaq[i]

        self.last_buffer = x
        return y

    def estsingle(self, x, d):

        # Filter
        self.single_buffer[1:] = self.single_buffer[:-1]
        self.single_buffer[0] = x
        s = np.dot(self.single_buffer,self.w)

        i_j = int(np.floor(s/self.delta_x) + (self.N/2))

        u_j = s/self.delta_x - np.floor(s/self.delta_x)

        u = np.asarray([math.pow(u_j,3),math.pow(u_j,2),u_j,1])
        udot = np.asarray([3*math.pow(u_j,2),2*u_j,1,0])

        if (abs(s)>2.0):
            q = np.asarray([(y-11)*self.delta_x for y in range(i_j-1,i_j+3)])
        else:
            q = np.asarray(list(self.ordinates[i_j-1:i_j+3]))

        y = np.matmul(u,self.Cm)
        y = np.matmul(y, q)
        y = float(y)

        self.e = d - y

        # Train filter
        deltaw = np.matmul(self.mu_w*self.e*udot,self.Cm)
        deltaw = float(np.matmul(deltaw, q))
        self.w = self.w + deltaw*self.single_buffer

        if abs(s) <= 2.0:
```

```
            deltaq = np.matmul(self.mu_q*self.e*self.Cm.T,u)
            q = q + deltaq
            for i in range(np.shape(q)[1]):
                self.ordinates[i_j+i-1] = float(q[0,i])

        return y

    def estsinglefilter(self, x, d):

        # Filter
        self.single_buffer[1:] = self.single_buffer[:-1]
        self.single_buffer[0] = x
        # print(self.w)
        s = np.dot(self.single_buffer,self.w)


        i_j = int(np.floor(s/self.delta_x) + (self.N/2))
        q = np.asarray([(y-11)*self.delta_x for y in range(i_j-1,i_j+3)])

        u_j = s/self.delta_x - np.floor(s/self.delta_x)
        u = np.asarray([math.pow(u_j,3),math.pow(u_j,2),u_j,1])
        udot = np.asarray([3*math.pow(u_j,2),2*u_j,1,0])

        y = np.matmul(u,self.Cm)
        y = np.matmul(y, q)

        y = float(y)

        self.e = d - y

        # Train filter
        deltaw = np.matmul(self.mu_w*self.e*udot,self.Cm)
        deltaw = float(np.matmul(deltaw, q))
        self.w = self.w + deltaw*self.single_buffer

        # print(self.w[0])
        return y

    def estsinglespline(self, x, d):
        i_j = int(np.floor(x/self.delta_x) + (self.N/2))

        u_j = x/self.delta_x - np.floor(x/self.delta_x)

        u = np.asarray([math.pow(u_j,3),math.pow(u_j,2),u_j,1])

        q = np.asarray(list(self.ordinates[i_j-1:i_j+3]))

        y = np.matmul(u,self.Cm)
        y = np.matmul(y, q)
        y = float(y)

        self.e = d - y
```

```
        deltaq = np.matmul(self.mu_q*self.e*self.Cm.T,u)
        q = q + deltaq

        for i in range(np.shape(q)[1]):
            self.ordinates[i_j+i-1] = float(q[0,i])

        return y
```

Audio Test with Graphing

```
if __name__ == "__main__":

    sr, data = wavfile.read("4065long.wav")

    speaker1 = data[:,0]
    speaker2 = data[:,1]

    echo = np.copy(speaker1)

    echo_amp = 0.5 # ratio
    echo_delay = 0.1 #seconds

    roll_amt = int(echo_delay * sr)

    echo = signal.lfilter(signal.firwin(50,0.3,pass_zero='lowpass'),1,echo)

    echo = echo * echo_amp
    echo = np.roll(echo,roll_amt)
    echo[:roll_amt] = np.zeros(roll_amt)

    speaker2wecho = speaker2+echo

    output = np.asarray([[speaker1[i],speaker2wecho[i]] for i in
range(len(speaker1)) ])
    wavfile.write("bad.wav",sr,output)

    x = speaker1
    d = speaker2wecho

    filterlen = int(roll_amt*1.5)

    FD = FDSAF(filterlen)

    y = []
    e = []


    do_fdsaf = True
    if do_fdsaf:
        # Perform filtering with FDSAF
        zext_len = filterlen - (len(x)%filterlen) if len(x)%filterlen !=0 else 0
```

```
        x = np.pad(x,(0,zext_len),'constant')
        x = np.reshape(x,(-1,filterlen))
        d = np.pad(d,(0,zext_len),'constant')
        d = np.reshape(d,(-1,filterlen))
        for k in range(len(x)):
            y.append(FD.est(x[k], d[k]))
            e.append(FD.e)
        y = np.asarray(y).flatten()
        e = np.asarray(e).flatten()
        e = e[:-zext_len]
    else:
        # Perform normal SAF
        for k in range(len(x)):
            y.append(FD.estsingle(x[k], d[k]))
            e.append(FD.e)


    ERLE =
signal.lfilter(np.ones(10000)/10000,1,20*np.log(abs(echo/(e-speaker2))))
    plt.title("Echo vs ERLE")
    ax = plt.subplot(111)
    ax.plot(echo,label="Echo")
    ax2 = plt.twinx(ax)
    ax2.plot(ERLE,'r',label="ERLE")
    plt.legend()
    ax.legend(loc="lower left")
    plt.show()




    plt.show()
    plt.title("FIR Filter Weight")
    plt.ylabel("Filter Weight")
    plt.xlabel("Tap")
    plt.plot(np.fft.fft(FD.w_f).real[:int(len(FD.w_f)/2):-1]/240)
    plt.show()

    plotx = np.linspace(-2.2,2.2,100000)
    resultspline = interpolate.splrep(FD.abscissas, FD.ordinates, s=0)
    resulty = interpolate.splev(plotx, resultspline, der=0)

    plt.plot(plotx,resulty)

    plt.title("Result Spline")

    plt.show()

    output = np.asarray([[speaker1[i],e[i]] for i in range(len(speaker1)) ])
    wavfile.write("good.wav",sr,output)

    residual = e - speaker2
```

```
    wavfile.write("residualecho.wav",sr,residual)

    comparison = np.asarray([[echo[i], residual[i]] for i in range(len(e))])
    wavfile.write("compare.wav",sr,comparison)
```

Example 1 using FDSAF

```
if __name__ == "__main__":
    np.random.seed(0)
    filterlen = 120
    FD = FDSAF(filterlen) # initialize our FDSAF

    y = []
    e = []

    # FILTER COEFFICIENTS
    a = [1,-2.628,2.3,-0.6703]
    b = [0,0.1032,-0.0197,-0.0934]

    # SPLINE KNOTS
    spline_y = [-2.2,-2.0,-1.8,-1.6,-1.4,-1.2,-1.0,-0.8,
                -0.91,-0.42,-0.01,-0.1,0.1,-0.15,0.58,1.2,
                1.0,1.2,1.4,1.6,1.8,2.0,2.2]
    spline_x = np.linspace(-2.2,2.2,len(spline_y))
    targetspline = interpolate.splrep(spline_x, spline_y, s=0)

    ### TEST FOR EX 1
    x = np.random.uniform(-1,1,100000)
    d = signal.lfilter(b,a,x)
    d = interpolate.splev(d, targetspline, der=0)

    signal_power = np.mean(d**2)
    d = d + np.random.normal(loc=0.0,scale= signal_power/1000,size=len(d))
     #FOR BLOCKWISE TESTING
    zext_len = filterlen - (len(x)%filterlen) if len(x)%filterlen !=0 else 0

    x = np.pad(x,(0,zext_len),'constant')
    x = np.reshape(x,(-1,filterlen))

    d = np.pad(d,(0,zext_len),'constant')
    d = np.reshape(d,(-1,filterlen))

    for k in range(len(x)):
        y.append(FD.est(x[k], d[k]))
        e.append(FD.e)

    y = np.asarray(y).flatten()
```

```
    e = np.asarray(e).flatten()

    plotx = np.linspace(-2.2,2.2,100000)
    targety = interpolate.splev(plotx, targetspline, der=0)
    resultspline = interpolate.splrep(FD.abscissas, FD.ordinates, s=0)
    resulty = interpolate.splev(plotx, resultspline, der=0)

    fig = plt.figure()
    ax = plt.subplot(111)
    ax2 = plt.twiny(ax)
    ax2.plot(targety,label="Target Spline")
    ax2.get_xaxis().set_visible(False)
    ax.plot(spline_x,spline_y,'or',label="Spline Knots")
    ax2.plot(resulty,label="Result Spline")
    ax.legend()
    ax2.legend(loc="lower right")
    plt.title("Spline")
    plt.show()

    plt.title("Mean Squared Error")
    plt.xlabel("Sample")
    plt.ylabel("Mean Squared Error")

    plt.plot(signal.lfilter(np.ones(100)/100,1,np.asarray(e)**2))
    plt.show()

    neww = np.fft.fft(FD.w_f).real[:int(len(FD.w_f)/2):-1]
    neww = neww/240

    # plt.plot(neww)
    # plt.show()
    # exit()

    wprime,hprime = signal.freqz(b,a)
    w, h = signal.freqz(neww)
    fig = plt.figure()
    plt.title('Digital filter frequency response')
    ax1 = fig.add_subplot(111)

    ax1.plot(w, 20 * np.log10(abs(h)), 'b',label="Result Amplitude")
    ax1.plot(wprime, 20 * np.log10(abs(hprime)), 'r',label="Target Amplitude")
    plt.ylabel('Amplitude [dB]', color='b')
    plt.xlabel('Frequency [rad/sample]')

    ax2 = ax1.twinx()
    angles = np.unwrap(np.angle(h))
    anglesprime = np.unwrap(np.angle(hprime))
    ax2.plot(w, angles, 'g',label="Result Phase")
    ax2.plot(wprime,anglesprime, 'm',label="Target Phase")
    plt.ylabel('Angle (radians)', color='g')
    plt.grid()
    plt.axis('tight')
```

```
    ax1.legend()
    ax2.legend(loc="lower left")
    plt.show()
```

Example 1 using SAF

```
    ### WORKING TEST TO SHOW SAMPLE LEVEL FILTER AND SPLINE WITH RANDOM INPUT
(WITH NOISE)

    x = np.random.uniform(-1,1,100000)
    d = signal.lfilter(b,a,x)
    d = interpolate.splev(d, targetspline, der=0)

    signal_power = np.mean(d**2)
    d = d + np.random.normal(loc=0.0,scale= signal_power/1000,size=len(d))

    for k in range(np.shape(x)[0]):
        y.append(FD.estsingle(x[k], d[k]))
        e.append(FD.e)

    plotx = np.linspace(-2.2,2.2,100000)
    targety = interpolate.splev(plotx, targetspline, der=0)
    resultspline = interpolate.splrep(FD.abscissas, FD.ordinates, s=0)
    resulty = interpolate.splev(plotx, resultspline, der=0)

    fig = plt.figure()
    ax = plt.subplot(111)
    ax2 = plt.twiny(ax)
    ax2.plot(targety,label="Target Spline")
    ax2.get_xaxis().set_visible(False)
    ax.plot(spline_x,spline_y,'or',label="Spline Knots")
    ax2.plot(resulty,label="Result Spline")
    ax.legend()
    ax2.legend(loc="lower right")
    plt.title("Spline")
    plt.show()

    plt.title("Mean Squared Error")
    plt.xlabel("Sample")
    plt.ylabel("Mean Squared Error")

    plt.plot(signal.lfilter(np.ones(100)/100,1,np.asarray(e)**2))
    plt.show()


    wprime,hprime = signal.freqz(b,a)
    w, h = signal.freqz(FD.w)
    fig = plt.figure()
    plt.title('Digital filter frequency response')
    ax1 = fig.add_subplot(111)

    ax1.plot(w, 20 * np.log10(abs(h)), 'b',label="Result Amplitude")
    ax1.plot(wprime, 20 * np.log10(abs(hprime)), 'r',label="Target Amplitude")
```

```
    plt.ylabel('Amplitude [dB]', color='b')
    plt.xlabel('Frequency [rad/sample]')

    ax2 = ax1.twinx()
    angles = np.unwrap(np.angle(h))
    anglesprime = np.unwrap(np.angle(hprime))
    ax2.plot(w, angles, 'g',label="Result Phase")
    ax2.plot(wprime,anglesprime, 'm',label="Target Phase")
    plt.ylabel('Angle (radians)', color='g')
    plt.grid()
    plt.axis('tight')
    ax1.legend()
    ax2.legend(loc="lower left")
    plt.show()
```