Real-Time Digital Signal Processing : Speech Enhancement Project

Jiyu Fang CID: 01054797 Deland Liu CID: 01066080

March 2018

1 Objectives

- Implement spectral subtraction technique
- Improve algorithm to obtain best possible performance on test files provided

2 Equipment

- Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator(DSK6713)
- Oscilloscope
- Software signal generator
- APX520 audio analyzer

3 Speech Enhancement

Speech enhancement to improve intelligibility of speech signal degraded by noise is widely used in different applications such as mobile phones, speech recognition and hearing aids. The reliability and efficiency of noise suppression algorithms are becoming more important as consumers increase demand for noise-free communications.

4 Basic Algorithm

The project involves designing and implementing a real-time speech enhancer which removes noise from a speech signal. There are no prior knowledge of noise and speech. The basic algorithm involves overlap add processing, noise estimation, noise subtraction and I/O buffering. Several enhancements are implemented and evaluated to improve the basic algorithm.

Assuming that the spectrum of input signal can be considered as sum of noise and speech spectrum, the **spectral subtraction** technique is adopted, which involves first performing FFT (Fast Fourier Transform) overlapped frames of input signal and subtracting estimated noise spectrum. Inverse FFT is done on processed frames of spectrum each of which is multiplied by the window again to eliminate spectrum discontinuities. The processed time-domain signal is then outputted to audio port.

To extract frames, input signal is multiplied by Hamming window with overlap of 3/4 of a frame (oversampling ratio of 4). Since Hamming window smoothly attenuates amplitude at frame edges, spectral artifacts due to discontinuities at frame boundaries are reduced. Since overlapping windows add to 1, this **overlap-add technique** solves the problem of windows deliberately amplitude modulating the signal. Extracted time-domain frames are then FFTed and noise estimation, subtraction and other enhancements are performed in frequency domain. Note that processed time domain frames may contain discontinuities, hence window multiplication is again required at output. In regards to **noise estimation**, we assume speaker stops at least once every 10s for a pause. Hence, the spectrum of input during this pause would be the noise spectrum. The program calculates the spectral minimum over four 2.5s intervals to estimate noise spectrum and stores them in noise minimum buffers. For benchmark program, frame length is set to 256 samples, hence each frame contains 32ms data of signal for sampling freq (F_s) of 8kHz.

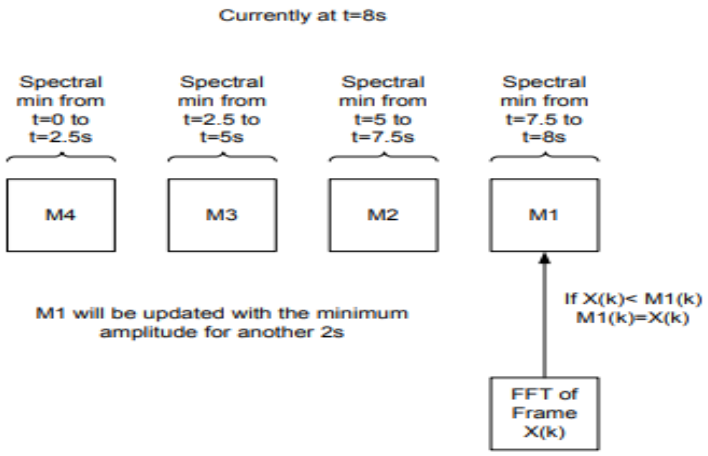


Figure 1: Operation of minimum buffers

There are 312 frames in each 2.5s ($\frac{(2.5\times4)}{0.032}$) and 256 frequency bins in spectrum of each frame. The additional multiplication with 4 is due to oversampling of 4. Shown in figure above, throughout 2.5s the spectrum is updated real-time based on description $M_1(\omega) = min(|X(\omega)|, M_1)$. Every 2.5s, buffer rotation is performed with M_{i+1}=M_i and $M_1 = X(\omega)$. Pointers are manipulated instead of copying data to improve efficiency. The noise estimate is determined for each frequency bin as the minimum magnitude present in any frame present over past 10s. This is described by the formula $|N(\omega)| = \alpha \times min(M_i(\omega))$ for i = 1...4. Since minimum tracking may underestimate average noise magnitude, compensating factor α is multiplied with estimate.

```
void process_noise(void){
    float *p;
    complex *pt;
    T=0;
    for (idx=0; idx< NFREQ;idx++){  //Note loop runs NFREQ times, taking advantage
                                    //of symmetrical property of real FFT
        X[idx] = cabs(inframe_temp[idx]); //Find magnitude spectrum of input frame
        x = X[idx];
        if ((enhancement1) == 1){
        lowpass_mag();
        }
        if ((enhancement2) == 1){
        lowpass_power();
        }
        if(frame_count == 0){   //Every 2.5s, set M1(w)=min(|X(w)|, M1(w))
            M1[idx] = X[idx];
        }
        else{
            if (M1[idx] > X[idx]){  //Find spectral minimum for current input frame
                M1[idx] = X[idx];
            }
        }
        N[idx] = min (min(M1[idx], M2[idx]), min(M3[idx], M4[idx])); //For each freq bin, noise estimate minimum of 4 buffers
        if(enhancement0 == 1){
            N[idx] = alpha*N[idx];
        }
        if ((enhancement3) == 1){
        lowpass_noise();

        }
```

Figure 2: Code implementation for noise estimation

Note that to improve code efficiency, we take advantage of the conjugate symmetry of real signal spectrum to halve the number of computations. This is shown when the frequency domain noise processing loop loops to NFREQ rather than FFTLEN.

```
if (++frame_count >= FRAMES_PER_SLOT_REFINED){ //If 2.5 seconds slot passed, rotate buffers
    frame_count = 0;
    p = M4;
    M4 = M3;
    M3 = M2;
    M2 = M1;
    M1 = p;
}
```

Figure 3: Buffer rotation after 2.5 seconds (312 frames) passed

As mentioned above, we perform spectral subtraction for noise removal before IFFT. Nonetheless, there is a lack of knowledge of phase of noise signal and the time minimum spectrum occurred; only magnitude is stored in minimum buffers. Hence, magnitude is subtracted and phase of input is maintained:

$$\begin{aligned} |Y(\omega)| &= |X(\omega)| - |N(\omega)| \\ &= X(\omega) \times (1 - \frac{|N(\omega)|}{|X(\omega)|}) \end{aligned}$$

$G(\omega) = (1 - \frac{|N(\omega)|}{|X(\omega)|})$ is a frequency-dependent gain factor. Note that G(k) can be negative due to overestimation of noise. We implement $G(\omega) = max(\lambda, (1 - \frac{|N(\omega)|}{|X(\omega)|})$, preventing $G(\omega)$ from reaching spectral floor parameter λ. λ should not be set to zero, since human ears are sensitive to "musical noise". This is explored more in the next section.

```
void gainfactor_calculate (void){
    //float X = cabs (inframe_temp[idx]);
    if ((enhancement4a) == 1){
    G[idx] =  (((lambda) > (1-(N[idx]/x))) ? (lambda) : (1-(N[idx]/x)));    //Frequency-dependent gain factor
    }
    if ((enhancement4b) == 1){
    G[idx] =  (((lambda*N[idx]/x) > (1-(N[idx]/x))) ? (lambda*N[idx]/x) : (1-(N[idx]/x)));
    }
    if ((enhancement4c) == 1){
    G[idx] =  (((lambda*P[idx]/x) > (1-(N[idx]/x))) ? (lambda*P[idx]/x) : (1-(N[idx]/x)));
    }
    if ((enhancement4d) == 1){
    G[idx] =  (((lambda*N[idx]/P[idx]) > (1-(N[idx]/P[idx]))) ? (lambda*N[idx]/P[idx]) : (1-(N[idx]/P[idx])));
    }
    if ((enhancement4e) == 1){
    G[idx] =  (((lambda) > (1-(N[idx]/P[idx]))) ? (lambda) : (1-(N[idx]/P[idx])));
    }
    if ((enhancement5) == 1){
    G[idx] =  (((lambda) > (sqrt(1-N[idx]*N[idx]/x*x))) ? (lambda) : (sqrt(1-N[idx]*N[idx]/x*x)));
    }
}
```

Figure 4: Gain factor calculation

```
outframe_temp[idx] = rmul(G[idx],inframe_temp[idx]);
//residual_noise_reduction ();
outframe_temp[FFTLEN-idx] = conjg(outframe_temp[idx]);
```

Figure 5: $Y(\omega) = X(\omega) \times G(\omega)$. Note real FFT magnitude spectrum is conjugate symmetric.

Note that if proceed with frame length of 3.2ms and 312 frames per 2.5s, there would be a delay of 10s before impact of speech enhancement becomes recognizable. This is because minimum spectrum would be zero for starting 10 seconds of a speech when not all minimum buffers have been filled up.

5 Performance evaluation for basic implementation

White noise's power spectrum is characterized by peaks and valleys that exist in random frequency bins and vary in amplitude between frames. Wider peaks contribute to time varying broadband noise, and narrower peaks result in "musical noise", which sounds like random musical notes playing rapidly in

background. An $\alpha > 1$ shift wide spectral peaks downward, hence removing broadband noise. A $\lambda > 0$ "fills in" valleys, reducing sharpness of narrow peaks. This results in enhanced signal with lower "musical noise" [1].

This basic implementation of spectral subtraction technique is evaluated for different λ and α values. As α increases, background noise removal is improved, but speech attenuation becomes significant at $\alpha > 20$. This is because the current basic algorithm results in overestimation of noise level at frequency bins with relatively high SNR ratio. In addition, perceived "musical noise" reduces with increasing α. As λ reaches zero, there is an increase in "musical noise". Increasing λ (spectral floor) reduces "musical noise" but enlarges background white noise. Compromised values of $\alpha = 25$ and $\lambda = 0.001$ are chosen for this primitive technique, which removes most of the background noise for factory 1 and lynx 1 but introduces small amount of musical noise. The performance of this basic algorithm is evaluated from spectrograms in "Final Program" section (figure19). Enhancements are explored below to further improve speech enhancement performance.

6 Enhancement Evaluation

6.1 Enhancement 1

This technique involves low pass filtering the magnitude spectrum of each frequency bin of input frame over time with a first order exponentially weighted moving average (EMWA). The single pole, recursive algorithm can be characterized as a first order low pass filter. $P_t(\omega) = (1-k) \times |X(\omega)| + k \times P_{t-1}(\omega)$, where $P_t(\omega)$ is input estimate for frame t and k the smoothing factor. k is chosen $e^{\frac{-Frame\,rate}{\tau}}$ such that less weight is given to historical spectral estimates of input as τ decreases. τ represents the time frame for system, to an extent, to disregard the historical values. The motivation behind finding moving average over past few input frames to estimate current frame at specific frequency bin is that components at each bin fluctuate over time. Noise minimum buffers may hence store underestimated noise values that are in reality sharp, instantaneous drops in magnitude. To recognize this, one must realize that the speech signal at specific frame and frequency bin can be modelled as Gaussian random variables. Mean and variance fluctuate with respect to time and frequency bin index. The basic algorithm of minimum tracking tends to bias noise estimate towards lower values. Hence, with this enhancement, noise estimate is more accurate and closer to noise average magnitude. Also, it would be possible to reduce oversubtraction factor α. Frame rate is 8ms in this case since 4 frames are generated every 32ms. A k of ≈ 0.67 is chosen with frame rate = 8ms and τ = 20ms.[2]

```
void lowpass_mag (void){
    X[idx] = (1 - zpole) * X[idx] + zpole * P[idx]; //Implement EWMA to calculate low pass filtered
                                                    //version of |X(w)|
    P[idx] = X[idx];
}
```

Figure 6: Code implementation for low pass filtering of input

With this enhancement, background broadband noise is reduced significantly and α reduced to 1 to avoid over-attenuation of speech.

6.2 Enhancement 2

This enhancement is very similar to the previous one, with the only difference that the low pass filtering is done on power domain rather than magnitude domain. Intuitively, noise estimate would be more accurate and sensitive to input frame magnitude changes, since dynamic range of changes increase. In addition, human hearing tracks power of signal rather than than magnitude; hence noise estimate from filtering in power domain relates more to what human ears perceive as noise [3]. This technique is found to be more effective than enhancement 1.

```
void lowpass_power (void){
    X[idx] = sqrt((1 - zpole)*X[idx]*X[idx] + zpole*P[idx]*P[idx]); //Low pass filter in power domain
    P[idx] = X[idx];
}
```

Figure 7: Code implementation for low pass filtering of input in power domain

6.3 Enhancement 3

A similar low pass filter implemented in previous 2 enhancements is applied on noise estimate to avoid abrupt discontinuities in random noise. This is apparent because the low pass filter is implemented by an EWMA, smoothing out fluctuations of noise estimate over time. This allows a more accurate representation of average noise such that random peaks in background noise have less influence. It also smooths out discontinuities from buffer rotations. Code implementation is shown below, where N[idx] is extracted through minimum spectral tracking, $|N(idx)| = \alpha \times min(M_i(idx))$ for i = 1...4. Little improvement is achieved with this enhancement. The effect would be more apparent if noise level is very variable; whereas noise remains constant in test files such as "factory" and "car". One disadvantage is that when there are magnitude outliers the noise magnitude will be overestimated and speech will be severely attenuated.

```
void lowpass_noise (void){
    N[idx] = (1 - zpole) * N[idx] + zpole * P_N[idx]; //Low pass noise to avoid abrupt discontinuities
    P_N [idx] = N[idx];
    }
```

Figure 8: Code implementation for low pass filtering of noise estimate

6.4 Enhancement 4

Recall that λ sets the spectral floor parameter of $g(\omega)$ when noise has been over estimated and $1 - \frac{|N(\omega)|}{|X(\omega)|}$ becomes negative. Nonetheless, studying the original calculation $G(\omega) = max(\lambda, (1 - \frac{|N(\omega)|}{|X(\omega)|})$, a low energy speech signal may result in $(1 - \frac{|N(\omega)|}{|X(\omega)|}) << \lambda$, hence resulting in information loss if maximum of the two taken. Multiplying λ by a weighting factor allows λ to decrease accordingly. In addition, multiplication with weighting factor $\frac{|N(\omega)|}{|X(\omega)|}$ would allow phase of input signal be maintained even if noise magnitude overwhelms speech magnitude and $(1 - \frac{|N(\omega)|}{|X(\omega)|})$ becomes less than 0. This is represented in real multiplication $C(\omega) = \lambda \frac{|N(\omega)|}{|X(\omega)|} \times X(\omega)$. In several implementations, moving average processed signal $P(\omega)$ is used to replace $X(\omega)$, since sharp variations of signal over successive frames could result in fluctuations of gain factor.

```
void gainfactor_calculate (void){
    //float X = cabs (inframe_temp[idx]);
    if ((enhancement4a) == 1){
    G[idx] =  (((lambda) > (1-(N[idx]/x))) ? (lambda) : (1-(N[idx]/x)));    //Frequency-dependent gain factor
    }
    if ((enhancement4b) == 1){
    G[idx] =  (((lambda*N[idx]/x) > (1-(N[idx]/x))) ? (lambda*N[idx]/x) : (1-(N[idx]/x)));
    }
    if ((enhancement4c) == 1){
    G[idx] =  (((lambda*P[idx]/x) > (1-(N[idx]/x))) ? (lambda*P[idx]/x) : (1-(N[idx]/x)));
    }
    if ((enhancement4d) == 1){
    G[idx] =  (((lambda*N[idx]/P[idx]) > (1-(N[idx]/P[idx]))) ? (lambda*N[idx]/P[idx]) : (1-(N[idx]/P[idx])));
    }
    if ((enhancement4e) == 1){
    G[idx] =  (((lambda) > (1-(N[idx]/P[idx]))) ? (lambda) : (1-(N[idx]/P[idx])));
    }
    if ((enhancement5) == 1){
    G[idx] =  (((lambda) > (sqrt(1-N[idx]*N[idx]/x*x))) ? (lambda) : (sqrt(1-N[idx]*N[idx]/x*x)));
    }
}
```

Figure 9: Code implementation for gain factor variation

Little noticeable effect is recognized between the 5 different implementations to set $g(\omega)$. Enhancement 4a is chosen.

6.5 Enhancement 5

As an alternative to enhancement 4, $g(\omega)$ can be implemented in the power domain instead of the amplitude domain. Intuitively, if $g(\omega)$ is set with enhancement 5, a $\frac{|N(\omega)|^2}{|P(\omega)|^2}$ would be significantly smaller if noise amplitude lower than signal amplitude. Hence $\sqrt{1-\frac{|N(\omega)|^2}{|P(\omega)|^2}}$ increases closer to 1. This results in poorer spectral subtraction in very noisy environment. The code template for enhancement 5 is shown in previous subsection. To calculate power, the *pow* function is not used as we found that it carries huge processing and causes delays (echo) in speech signal. This enhancement is not used because it does not introduce any improvements but adds additional musical noise comparing to enhancement 4a.

6.6 Enhancement 6

Although enhancement 1 and 2 are effective in reducing broadband noise, it doesn't solve the issue of "musical" noise. "Musical noise" occurs when isolated narrow peaks are left in a spectrum after the spectrum subtraction algorithm. The larger the discrepancy in magnitude between pairs of frequency bins the worse the 'musical noise', and it produces a sort of audible 'warble' of the speech.
The method proposed by enhancement 6 thus aims to minimize those peaks remained in the spectrum by decreasing spectral excursions. The is done by oversubtraction of noise level at the frequency bins that have a poor signal-to-noise ratio (SNR), so that the remnants of noise peaks will be lower while the speech intelligibility is protected[1]. Define signal-to-noise ratio as:

$$SNR = 20\log\frac{|\,X(w)\,|-|\,(N(w)\,|}{|\,N(w)\,|} \quad where \;\; |\,N(w)\,| = min_{i=1..4}(M_i(w))$$

Currently, α is set to 1. In practice, when SNR = 0dB, a larger value of α, such as 2.65, does not distort the speech. This suggests that at frequency bins with low SNR, spectral subtraction of a noise estimate that is more than minimum average magnitude should be done to make sure most of the noise peaks have been removed. Define a new subtraction factor β, we establish the following relationship between β and SNR(dB) while setting α to 1:

$$\beta = \begin{cases} 2.32 - 0.066SNR & \text{if } -5 \leq SNR \leq 20 \\ 2.65 & \text{if } SNR < -5 \\ 1 & \text{if } SNR > 20 \end{cases}$$

The reason for allowing subtraction factor to vary is that SNR of a frequency bin varies from frame to frame in proportion to signal energy. This is based on assumption that broadband noise remains constant (background car, factory noise)[1]. When the SNR is too small β should stop changing otherwise the speech will be distorted, also when SNR is too large it is meaningless to further decrease the β.
Since the logarithm operation in C is time-consuming and delay in processing will be introduced (echo sound), we calculate SNR as magnitude ratio of noise subtracted signal and noise estimate. β is clamped in the range of $[1, 2.65]$ and varies linearly in the middle.We also used lower *beta* for high SNR conditions and for high frequencies than for low SNR conditions and for low frequencies, this is because information contained is mainly upon the pitch of the voices and a large β at high frequencies would distort the speech:

$$SNR = \frac{\mid X(w) \mid - \mid (N(w) \mid}{\mid N(w) \mid} \quad where \mid N(w) \mid = min_{i=1..4}(M_i(w))$$

$$\beta = \begin{cases} 2.65 - 0.0165(SNR)^2 & \text{if } k < \text{NFREQ}/2 \quad where\ k\ is\ frequency\ bin\ index \\ 2.5 - 0.015(\text{SNR})^2 & \text{otherwise} \end{cases}$$

alpha could also be used to calibrate the general level of subtraction.
A new function *oversubtraction*() is declared to the calculation above:

```
void oversubtraction(void){
    snr = (x/N[idx]-1);
    if(idx<NFREQ/2){                //for lower frequencies
    beta = 2.65-0.0165*snr*snr;     //calculation formula
    if(beta>2.65) beta = 2.65;      //clamp beta in range [1,2.65]
    if(beta<1) beta = 1;
    }
    else{                           //for higher frequencies
    beta = 2.5-0.015*snr*snr;       //calculation formula
    if(beta>2.5) beta = 2.5;        //clamp beta in range [1,2.5]
    if(beta<1) beta = 1;
    }
    N[idx] = beta*N[idx];           //multiplied to noise estimate of current frequency bin

}
```

Figure 10: Code implementation for noise oversubtraction

This enhancement turns out to be very useful as it effectively reduced 'musical noise' and the background noise is further attenuated.

6.7 Enhancement 7

This enhancement relies on changing the number of samples per frame. Originally the frame length is 256 samples, which means one frame contains information of a time period of $32ms$:

```
int FFTLEN = 256;                   /* fft length = frame length 256/8000 = 32 ms*/
```

Figure 11: Code implementation for changing frame lengths

By declaring $FFTLEN$ as an integer variable we could change frame length in real-time and observe the effect. Decrease it to 128(make sure it is still an integer power of 2 so that we could use FFT),

the audio sounds rough with more background noise and musical noise. As the number of samples in frequency domain are the same as in time domain after FFT, decreasing frame length reduces frequency resolution for each frame. The frequency resolution is $\Delta f = \frac{f_s}{FFTLEN}$, it is obvious that when FFTLEN decreases, the spectrum distance between each pair of samples increases and there are less samples of fundamental frequency components. As a result, quality of frequency domain processing deteriorates. When transformed back into time domain the hearing experience worsens.
On the other hand, increasing FFTLEN, say to 512 samples, introduces obvious delay between input and output and barely no enhancements could be implemented without distortion since there are more processing. Also, if we sample the signal too often we might not be able to capture the main trend(or dominant frequency components) of signal itself within a temporal time, since noise is fast-varying. The performance will be worse when noise is not stationary. Increasing frame length gives poor time resolution.
Based on testing, frame length of 256 samples is the best compromise.

6.8 Enhancement 8

After subtracting the noise estimation, the noise above the estimation remains, which is called the noise residual. Noise residual exhibits itself in the spectrum as randomly spaced narrow bands of magnitude spikes[3]. It sounds like tone generators with random fundamental frequencies. This undesired audio effect could be reduced by replacing $Y(\omega)$ with minimum magnitude calculated in three adjacent frames at that frequency bin, when $\mid Y(\omega) \mid$ at that frequency bin is less than the maximum magnitude of noise residual. This is because when the signal-to-noise ratio is smaller than the maximum noise residual the variation of magnitudes in adjacent frames at a certain frequency bin is mostly due to noise remained in the spectrum, assuming the signal spectrum magnitude is constant. By taking the minimum magnitude we could minimize the noise residual in the output. When SNR is larger than the threshold we do not bother taking the minimum since subtracting the noise estimation is enough.
To implement this algorithm we need two more complex arrays of size FFTLEN to store data of previous and future frames, which are *previous* and $future$ respectively. Data of two frames are taken at the start of the the program by checking the flag variable *start*, and store them into $inputframe\_temp$ and $future$ respectively. FFT are done on both arrays at the start of the program and *process_noise*() is only done on the current frame. When the current frame output is available the function *residual_noise_reduction* is called to compare and take the minimum magnitude output from adjacent three frames when $\frac{|N(w)|}{|X(w)|}$ exceeds some threshold value *residue*. $\frac{|N(w)|}{|X(w)|}$ approximates the magnitude ratio of maximum noise residual to actual underlying signal described above, and a suitable value of 0.68 of *residue* is obtained through testing. After the final output of each frequency bin is computed buffers are copied around to age the data by rotating the array pointers. After the start of the program, where *start* is set to 0, each time we get a new frame we will store it to $future$ and FFT is done on it only since $inframe\_temp$ already got data via buffer rotation.

```
if ((enhancement8) == 1){                                   //if enhancement8 is enabled
 if(start==0){                                              //if it is not the start of programe,
     for(k=0; k<FFTLEN;k++){                                //store the new frame into future array
             future[k].r = inbuffer[m] * inwin[k];
             future[k].i = 0;
             if (++m >= CIRCBUF) m=0; /* wrap if required */
     }
 }
 else{                                                      //if it is the start of programe
         for (k=0;k<FFTLEN;k++){                            //get two frames at a time and store into
             inframe[k] = inbuffer[m] * inwin[k];           //inframe_temp(current frame) and future
             inframe_temp[k].r = inframe[k];                //respectively
             inframe_temp[k].i=0;
             if (++m >= CIRCBUF) m=0; /* wrap if required */
         }
         while((io_ptr/FRAMEINC) != frame_ptr);
         if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;
         io_ptr0=frame_ptr * FRAMEINC;
         m=io_ptr0;
         for(k=0; k<FFTLEN;k++){
             future[k].r = inbuffer[m] * inwin[k];
             future[k].i = 0;
             if (++m >= CIRCBUF) m=0; /* wrap if required */
         }
         //start = 0;
 }
}
 else{                                                      //if enhancement8 is not enabled
 for (k=0;k<FFTLEN;k++)                                     //do the same thing as before
 {
     inframe[k] = inbuffer[m] * inwin[k];
     inframe_temp[k].r = inframe[k];
     inframe_temp[k].i=0;
     if (++m >= CIRCBUF) m=0;
 }
 }
```

Figure 12: Store frame data differently at and after the start of the program

```
void residual_noise_reduction (void){

    float a, b, c, m;
    if((N[idx]/x)>residue){                  //do this only when |N/X| is smaller than some threshold
        a = cabs(previous[idx]);             //magnitude of previous output
        x = cabs(future[idx]);               //obtain future output using same noise estimation
        gainfactor_calculate();
        b = cabs(rmul(G[idx],future[idx]));
        c = cabs(outframe_temp[idx]);
        m = min(min(a,b),c);                 //get the minimum
        if(a == m) outframe_temp[idx] = previous[idx];//update current output with output of minimum magnitude
        else if(b==m) {                              //in thrre adjacent frames
        outframe_temp[idx] = rmul(G[idx],future[idx]);
    }
}
}
```

Figure 13: Function for replacing output when $\frac{|N(w)|}{|X(w)|}$ is smaller than *residue* by that with minimum magnitude from three adjacent frames

```
if ((enhancement8) == 1){               //rotate the buffer to age the data
    pt = &previous[0];                  //note that the updated inframe_temp
    previous = &outframe_temp[0];       //already contains data in frequency domain
    inframe_temp = &future[0];
    future = pt;
}
```

Figure 14: Buffer rotation to age the data

This enhancement can effectively reduce the 'musical noise' with audible decrease in those 'warble' sound.

6.9 Enhancement 9

Previously we are estimating noise magnitude by looking at data for past 10 seconds, which gives relatively accurate estimation since normally the speaker will at least stop once during a 10-second period. As mentioned before, the disadvantage is that noise reduction becomes apparent after 10s. We could change the estimation period by changing FRAMES_PER_SLOT. We change it to a variable of type integer so that we could change it in real-time. Since there is no speech activity in the starting 2.5 seconds of all the test audio files and speaking speed is relatively slow, it is enough to decrease FRAMES_PER_SLOT to $\frac{312}{4} = 78$, so that noise subtraction commences before speech starts. Noise estimation is then the minimum magnitude of past 2.5s($78x4x0.08ms \approx 2.5s$). However, decreasing the estimation period too much will introduce distortion since the noise magnitude may be overestimated and the speech would be attenuated.

```
if ((enhancement9) == 1){
    FRAMES_PER_SLOT = 78;
}
```

Figure 15: Code implementation for changing noise estimation period

6.10 Additional Enhancements

Additional Signal Attenuation During Nonspeech Activity
The energy content of output signal relatively to noise could indicate whether there is a speech activity. Let's define a measure T:

$$T = 20\log_{10}[\frac{1}{2\pi}\int_0^{2\pi}\frac{|\,\mathrm{Y(w)}\,|}{|\,\mathrm{N(w)}\,|}\,\mathrm{d}w]$$

Practically a frame is classified as having no speech activity if $T < -12dB$[3]. In the program we only loop through half of the spectrum image since the magnitude spectrum is symmetric, hence we define the following algorithm:

$$T < -12dB \implies \int_0^{\pi}\frac{|\,\mathrm{Y(w)}\,|}{|\,\mathrm{N(w)}\,|}\,\mathrm{d}w < 0.789$$

$$Y(w) = \begin{cases} Y(w) & \text{if } T \geq -12dB \\ cX(w) & \text{otherwise} \quad where\ 20\log_{10} c = -30dB \implies c = 0.0316 \end{cases}$$

If the T is smaller than the -12dB, we can consider there is no speech activity and only a small portion of the input will be outputted. The code implementation is as below:

```
if(enhancement10 ==1){                    //calculate the average power ratio of output
    T+= cabs(outframe_temp[idx])/N[idx];  //and noise when the output is available
}
```

Figure 16: Add up the magnitude ratio of output and noise to T for each frequency bin

```
void nonspeechattenuation(void){
    int j;                                                  //check the energy measure and update
    if(T<0.789){                                            //the output of each frequency bin based
        for(j=0; j<NFREQ; j++){                             //on the defined algorithm
            outframe_temp[j] = rmul(0.0316, inframe_temp[j]);
            outframe_temp[FFTLEN-j] = conjg(outframe_temp[j]);
        }
    }
}
```

Figure 17: Nonspeech detection and output update

However when adding this enhancement there no audible difference and when using it other enhancements delay is introduced since it is time-consuming as it requires another loop to do the output update.

7 Final Program

The final program with optimal performance adopted **enhancements 2,4a,6,8,9** together with the basic noise subtraction algorithm and the following parameters:

α	λ	τ	$residue$
1.5	0.001	0.02	0.68

During the listening test we could hear that the system is very effective at removing the background and 'musical' noise, while maintaining good speech intelligibility even for 'phantom4.wav'.
We analyzed final speech enhancement performance using spectrogram, comparing difference between "clean", "lynx2" and "phantom4" files after enhancement. This is a more objective method to analyze performance.

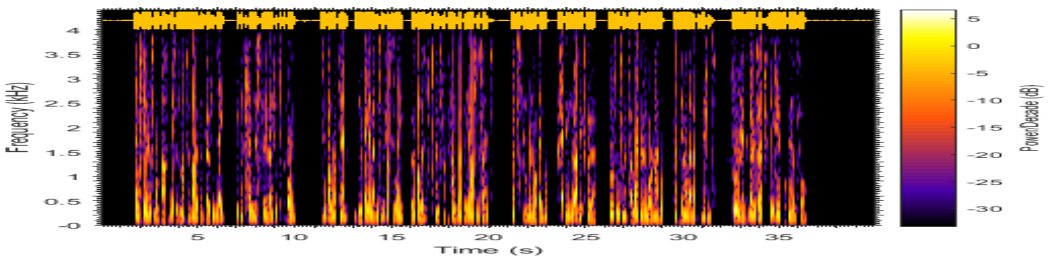
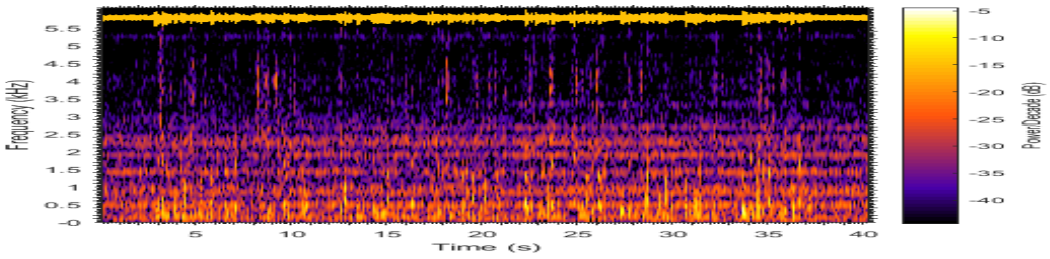


Figure 18: "Clean" file spectrogram



ynx2" file spectrogram

ynx2" file spectrogram

Figure 19: "l
ynx2" file spectrogram

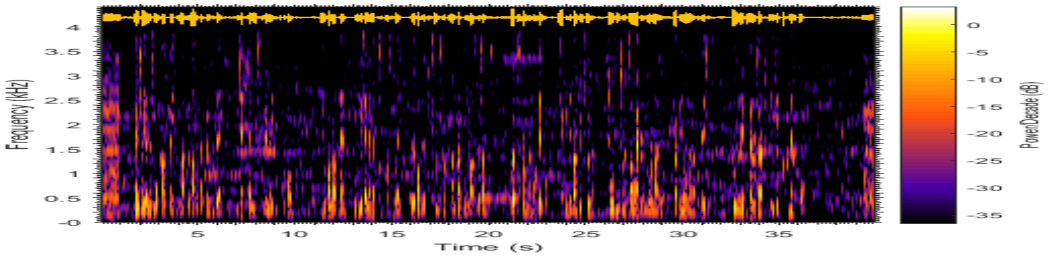


Figure 20: lynx2 after basic algorithm

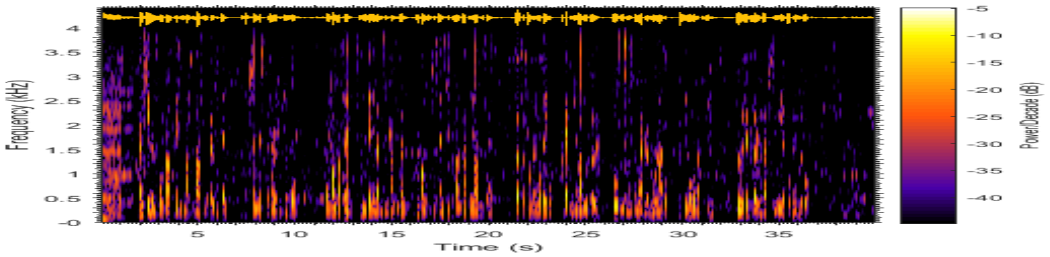


Figure 21: lynx2 after enhancement

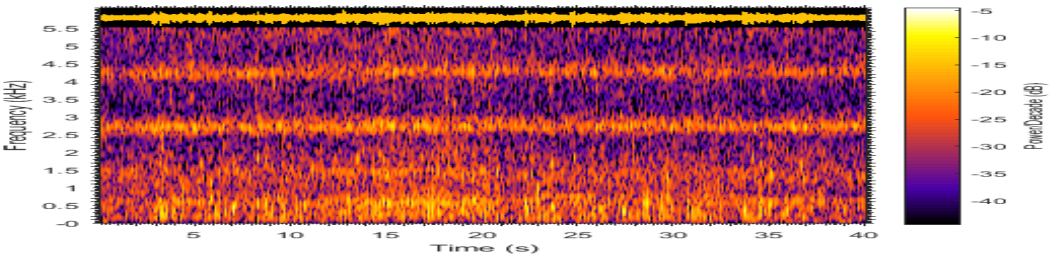


Figure 22: "phantom4" file spectrogram

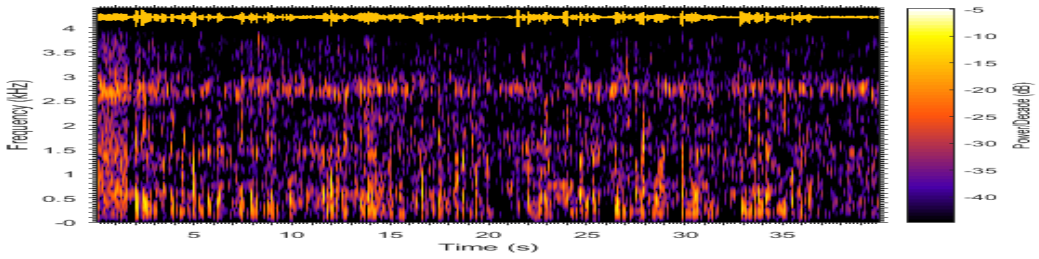


Figure 23: Phantom4 after enhancement

Significant removal of background broadband noise is achieved for both test files, seen in black spaces at periods of silence (11-12.5s, 20-21s). In "lynx2", the basic algorithm is limited to remove "musical" noise, seen in sharp fluctuation of magnitude between frequency bins. In addition, the use of large over-subtraction factor α of 25 to compensate for underestimation of speech removes speech signal. With addition of final enhancements, for "Phantom4", despite minor attenuation of speech signal, noise introduced at 2.5kHz is reduced significantly. In "lynx2", background noise at very low and high frequency bins are largely removed. SNR tend to be low at these bins, since usable voice frequency ranges from 300Hz to 3.4kHz. Also, during speech pauses, "musical" noise is decreased, shown in less magnitude jumps across frames at particular frequency bins. The effect of each enhancement chosen is analyzed on "lynx2" test file and spectrogram results presented within appendix (figure 27,28,29).

8 References

[1] M.Berouti, R.Schwartz, J.Makhoul. (1979, April). Enhancement of speech corrupted by acoustic noise. Presented at IEEE International Conference on ICASSP. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=1170788

[2] R.Martin.(1994). Spectral Subtraction Based on Minimum Statistics. Presented at EUSIPCO. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.472.3691

[3] S.Boll. (1979, April). Suppression of acousitc noise in speech using spectral subtraction. IEEE Transactions on Acoustics, Speech and Signal Processing. Volume 27, Issue: 2, p113-120. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1163209

9 Appendix

Speech Enhancer Implementation in C:

```
/*************************************************************************
                        DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                                                IMPERIAL COLLEGE LONDON

                                EE 3.19: Real Time Digital Signal Processing
                                        Dr Paul Mitcheson and Daniel Harvey

                                                PROJECT: Frame Processing

                                        ********* ENHANCE. C **********
                                                Shell for speech enhancement

                        Demonstrates overlap-add frame processing (interrupt driven) on the DSK.

 *************************************************************************
                                        By Danny Harvey: 21 July 2006
                                                Updated for use on CCS v4 Sept 2010
 *************************************************************************/
/*
 *        You should modify the code so that a speech enhancement project is built
 *  on top of this template.
 */
/************************** Pre-processor statements ***************************/
//Set up software switches for different enhancements
int enhancement0        =1;
```

```
int enhancement1                =0;
int enhancement2                =1;
int enhancement3                =0;
int enhancement4a                =1;
int enhancement4b                =0;
int enhancement4c                =0;
int enhancement4d                =0;
int enhancement4e                =0;
int enhancement5                =0;
int enhancement6                =1;
int enhancement7                =1;
int enhancement8                =1;
int enhancement9                =1;
int enhancement10       =0;

//  library required when using calloc
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"
/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

/* Some functions to help with Complex algebra and FFT. */
#include "cmplx.h"
#include "fft_functions.h"

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

#define WINCONST 0.85185                        /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0                            /* sample frequency, ensure this matches Config for AIC */
//#define FFTLEN 256                            /* fft length = frame length 256/8000 = 32 ms*/
int FFTLEN = 256;                   /* fft length = frame length 256/8000 = 32 ms*/
int FRAMES_PER_SLOT = 312;          /*frames_per_slot = 2.5*4/32ms = 312.5*/
#define NFREQ (1+FFTLEN/2)                      /* number of frequency bins from a real FFT */
#define OVERSAMP 4                                      /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP)        /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC)        /* length of I/O buffers */

#define OUTGAIN 16000.0                             /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)               /* Input gain for ADC  */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP        /* time between calculation of each frame */
// #define FRAMES_PER_SLOT 312       //312 frames per 2.5s slot


/****************************** Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
                        /*****************************************************************/
                        /*   REGISTER                  FUNCTION                  SETTINGS         */
                        /*****************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off     */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on     */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit              */\
    0x008d,  /* 8 SAMPLERATE Sample rate control        8 KHZ-ensure matches FSAMP */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
                        /*****************************************************************/
};

// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

int idx = 0;
float *inbuffer, *outbuffer;                    /* Input/output circular buffers */
float *inframe, *outframe;          /* Input and output frames */
float *inwin, *outwin;              /* Input and output windows */
float *M1;                                              //define noise minimum buffers
float *M2;
float *M3;
float *M4;
float *X;                           //Holds magnitude spectrum of current input frame (waiting to be compared)
float *N;                           // Estimate of noise spectrum
```

```
float *G;                                                   //  Frequency Dependent Gain Factor
float *P;                                                   //low-pass filtered input estimate for current frame
float *P_N;                                                   //low-pass filtered estimate for noise
float ingain, outgain;                                  /* ADC and DAC gains */
float cpufrac;                                             /* Fraction of CPU time used */
volatile int io_ptr=0;                 /* Input/ouput pointer for circular buffers */
volatile int frame_ptr=0;              /* Frame pointer */
complex *inframe_temp;                                  //Temporary holding of input frame to compute FFT
complex *outframe_temp;                                  //Temporary holding of output frame to compute FFT
complex *future, *previous;                            //Stroes data of the previous and the next frames
volatile int frame_count = 0;              //Counter recording when 312th frame arrives (2.5s), then can perform buffer rotation
float beta = 1;                                            //Set beta that corrects underestimation
float lambda = 0.001;                  //lower bound for gain factor
float tc = 0.02;                                        //time constant used for low-pass filter
float residue = 0.68;                                  //threshold value for enhencement 8
float zpole;                                                  //Z-pole plane, zpole = exp (-T/tc)
//int SLOTS_FIRST10S = 1;
float snr=0;                                               //Stores signal-to-noise ratio(not in dB)
float x=0;                                                       //Temporary storage for input magnitude
int start = 1;                                               //flag only set for the start of the program
float T =0;                                                 /*average power ratio of output and noise*/
float alpha = 1.5;                                        //overal noise subtration factor
 /****************************** Function prototypes ****************************/
void init_hardware(void);              /* Initialize codec */
void init_HWI(void);               /* Initialize hardware interrupts */
void ISR_AIC(void);                /* Interrupt service routine for codec */
void process_frame(void);          /* Frame processing routine */
float min(float a, float b);       /*return minimum among two values*/
void process_noise(void);          /*noise estimation and subtratioin*/
void lowpass_mag (void);           /*low pass filter for input magnitude*/
void lowpass_power (void);         /*low pass filter for input power*/
void lowpass_noise (void);         /*low pass filter for noise magnitude*/
void gainfactor_calculate (void);          //variation of gain factor
void oversubtraction (void);     //oversubtraction technique to overestimate noise levels at freq bins with low SNR
void residual_noise_reduction (void); //residual noise reduction technique to rid musical noise
void nonspeechattenuation(void);       // attenuate the noise during non-speech activity
/********************************* Main routine ***********************************/
void main()
{

          int i; // used in various for loops


/*  Initialize and zero fill arrays */

        inbuffer        = (float *) calloc(CIRCBUF, sizeof(float));        /* Input array */
    outbuffer        = (float *) calloc(CIRCBUF, sizeof(float));        /* Output array */
        inframe                = (float *) calloc(FFTLEN, sizeof(float));        /* Array for processing*/
    outframe        = (float *) calloc(FFTLEN, sizeof(float));        /* Array for processing*/
    inwin                = (float *) calloc(FFTLEN, sizeof(float));        /* Input window */
    outwin                = (float *) calloc(FFTLEN, sizeof(float));        /* Output window */
    inframe_temp = (complex *) calloc(FFTLEN,sizeof(complex));
    outframe_temp = (complex *) calloc(FFTLEN,sizeof(complex));
    M1  = (float *) calloc(NFREQ, sizeof(float));
    M2  = (float *) calloc(NFREQ, sizeof(float));
        M3  = (float *) calloc(NFREQ, sizeof(float));
        M4  = (float *) calloc(NFREQ, sizeof(float));
        X  = (float *) calloc(NFREQ, sizeof(float));
        N  = (float *) calloc(NFREQ, sizeof(float));
        G  = (float *) calloc(NFREQ, sizeof(float));
    if ((enhancement8) == 1){
    future = (complex *) calloc(FFTLEN,sizeof(complex));
    previous = (complex *) calloc(FFTLEN,sizeof(complex));
    }
           if ((enhancement1) == 1 || (enhancement2) == 1){
        P  = (float *) calloc(NFREQ, sizeof(float));
        }
        if ((enhancement3) == 1){
        P_N = (float *) calloc(NFREQ, sizeof(float));
        }

        /* initialize board and the audio port */
          init_hardware();

          /* initialize hardware interrupts */
          init_HWI();

/* initialize algorithm constants */
    zpole = exp(-TFRAME/tc);
          for (i=0;i<FFTLEN;i++)
        {
        inwin[i] = sqrt((1.0-WINCONST*cos(PI*(2*i+1)/FFTLEN))/OVERSAMP);
        outwin[i] = inwin[i];
        }
          ingain=INGAIN;
          outgain=OUTGAIN;
```

```
          /* main loop, wait for interrupt */
          while(1) {

                  process_frame();
}
}
/********************************* init_hardware() ******************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

        /* Function below sets the number of bits in word used by MSBSP (serial port) for
        receives from AIC23 (audio port). We are using a 32 bit packet containing two
        16 bit numbers hence 32BIT is set for  receive */
        MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

        /* Configures interrupt to activate on each consecutive available 32 bits
        from Audio port hence an interrupt is generated for each L & R sample pair */
        MCBSP_FSETS(SPCR1, RINTM, FRM);

        /* These commands do the same thing as above but applied to data transfers to the
        audio port */
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
        MCBSP_FSETS(SPCR1, XINTM, FRM);


}
/********************************* init_HWI() ***********************************/
void init_HWI(void)
{
        IRQ_globalDisable();                        // Globally disables interrupts
        IRQ_nmiEnable();                              // Enables the NMI interrupt (used by the debugger)
        IRQ_map(IRQ_EVT_RINT1,4);                 // Maps an event to a physical interrupt
        IRQ_enable(IRQ_EVT_RINT1);                 // Enables the event
        IRQ_globalEnable();                                 // Globally enables interrupts

}

/******************************** process_frame() ***********************************/
void process_frame(void)
{
        int k,m;
        int io_ptr0;

        /* work out fraction of available CPU time used by algorithm */
        cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;

        /* wait until io_ptr is at the start of the current frame */
        while((io_ptr/FRAMEINC) != frame_ptr);

        /* then increment the framecount (wrapping if required) */
        if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;

         /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where the
         data should be read (inbuffer) and saved (outbuffer) for the purpose of processing */
         io_ptr0=frame_ptr * FRAMEINC;

        /* copy input data from inbuffer into inframe (starting from the pointer position) */

        m=io_ptr0;

   if ((enhancement8) == 1){                    //if enhancement8 is enabled
        if(start==0){                                                                   //if it is not the start of programe,
                for(k=0; k<FFTLEN;k++){                                         //store the new frame into future array
                                future[k].r = inbuffer[m] * inwin[k];
                                future[k].i = 0;
                                if (++m >= CIRCBUF) m=0; /* wrap if required */
                }
        }
        else{                                                                                                   //if it is the start of programe
                for (k=0;k<FFTLEN;k++){                 //get two frames at a time and store into
                                inframe[k] = inbuffer[m] * inwin[k]; //inframe_temp(current frame) and future
                                inframe_temp[k].r = inframe[k];                 //respectively
                                inframe_temp[k].i=0;
                                if (++m >= CIRCBUF) m=0; /* wrap if required */
                    }
                    while((io_ptr/FRAMEINC) != frame_ptr);
                    if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;
                    io_ptr0=frame_ptr * FRAMEINC;
                    m=io_ptr0;
                    for(k=0; k<FFTLEN;k++){
                                future[k].r = inbuffer[m] * inwin[k];
```

```
                                future[k].i = 0;
                                if (++m >= CIRCBUF) m=0; /* wrap if required */
                        }
                        //start = 0;
        }
   }
        else{                                                                   //if enhancement8 is not enabled
    for (k=0;k<FFTLEN;k++)                                          //do the same thing as before
        {
                inframe[k] = inbuffer[m] * inwin[k];
                inframe_temp[k].r = inframe[k];
                inframe_temp[k].i=0;
                if (++m >= CIRCBUF) m=0;
        }
        }
        /************************ DO PROCESSING OF FRAME  HERE ************************/
        //for (k=0; k<FFTLEN; k++)
        //{
                //inframe_temp[k].r = inframe[k];
                //inframe_temp[k].i = 0;
        //}

        if ((enhancement8) == 1){
        if(start) {                                             //if it is the start,
                fft(FFTLEN, inframe_temp); //FFT is done on both arrays
                start = 0;                 //if not only on the future array
        }
        fft(FFTLEN, future);
        }
        else{
        fft(FFTLEN, inframe_temp);    //if enhancement8 is not enabled
        }                                                       //same as before
        process_noise(); //noise estimation and speech processing


        /*****************************Passing On to Output************************************/

        ifft(FFTLEN, outframe_temp);
        //for (k=0; k<FFTLEN; k++)
        //{
                //outframe[k] = outframe_temp[k].r;
                //outframe[k] = inframe[k];
        //}

    /* multiply outframe by output window and overlap-add into output buffer */

        m=io_ptr0;

    for (k=0;k<(FFTLEN-FRAMEINC);k++)
        {                                                                                       /* this loop adds into outbuffer */
                outframe[k] = outframe_temp[k].r;
                  outbuffer[m] = outbuffer[m]+outframe[k]*outwin[k];
                if (++m >= CIRCBUF) m=0; /* wrap if required */
        }
    for (;k<FFTLEN;k++)
        {
                outframe[k] = outframe_temp[k].r;
                outbuffer[m] = outframe[k]*outwin[k];   /* this loop over-writes outbuffer */
            m++;
        }
}
/************************* INTERRUPT SERVICE ROUTINE  ****************************/

// Map this to the appropriate interrupt in the CDB file

void ISR_AIC(void)
{
        short sample;
        /* Read and write the ADC and DAC using inbuffer and outbuffer */

        sample = mono_read_16Bit();
        inbuffer[io_ptr] = ((float)sample)*ingain;
                /* write new output data */
        mono_write_16Bit((int)(outbuffer[io_ptr]*outgain));

        /* update io_ptr and check for buffer wraparound */

        if (++io_ptr >= CIRCBUF) io_ptr=0;
}

/*******************************************************************************/
float min(float a, float b){

        return ((a<b)? (a):(b));
}

void process_noise(void){
```

```
        float *p;
        complex *pt;
        T=0;
        for (idx=0; idx< NFREQ;idx++){        //Note loop runs NFREQ times, taking advantage
                                                                        //of symmetrical property of real FFT
                X[idx] = cabs(inframe_temp[idx]); //Find magnitude spectrum of input frame
                x = X[idx];
                if ((enhancement1) == 1){
                lowpass_mag();
                }
                if ((enhancement2) == 1){
                lowpass_power();
                }
                if(frame_count == 0){        //Every 2.5s, set M1(w)=min(|X(w)|, M1(w))
                        M1[idx] = X[idx];
                }
                else{
                        if (M1[idx] > X[idx]){  //Find spectral minimum for current input frame
                                M1[idx] = X[idx];
                        }
                }
                N[idx] = min (min(M1[idx], M2[idx]), min(M3[idx], M4[idx])); //For each freq bin,
                if(enhancement0 == 1){
                        N[idx] = alpha*N[idx];
                }
                if ((enhancement3) == 1){
                lowpass_noise();

                }

                if ((enhancement6) == 1){
                oversubtraction();

                }


                gainfactor_calculate();
                outframe_temp[idx] = rmul(G[idx],inframe_temp[idx]);

                if(enhancement10 ==1){                                          //calculate the average power ratio of output
                        T+= cabs(outframe_temp[idx])/N[idx];        //and noise when the output is available
                }
                if ((enhancement8) == 1){
                residual_noise_reduction ();
                }
                outframe_temp[FFTLEN-idx] = conjg(outframe_temp[idx]);

        }
        if((enhancement10)==1 ){
                        nonspeechattenuation();
                }

        if ((enhancement8) == 1){                               //rotate the buffer to age the data
                pt = &previous[0];                                      //note that the updated inframe_temp
                previous = &outframe_temp[0];               //already contains data in frequency domain
                inframe_temp = &future[0];
                future = pt;
        }

        if ((enhancement9) == 1){
                FRAMES_PER_SLOT = 78;
        }

        if (++frame_count >= FRAMES_PER_SLOT){ //If 2.5 seconds slot passed, rotate buffers
                frame_count = 0;
                p = M4;
                M4 = M3;
                M3 = M2;
                M2 = M1;
                M1 = p;
        }
}
void lowpass_mag (void){
        //Implement EWMA to calculate low pass filtered version of |X(w)|
        X[idx] = (1 - zpole) * X[idx] + zpole * P[idx];
        P[idx] = X[idx];
}

void lowpass_power (void){
        //Low pass filter in power domain
        X[idx] = sqrt((1 - zpole)*X[idx]*X[idx] + zpole*P[idx]*P[idx]);
        P[idx] = X[idx];
}
void lowpass_noise (void){
        //Low pass noise to avoid abrupt discontinuities
        N[idx] = (1 - zpole) * N[idx] + zpole * P_N[idx];
        P_N [idx] = N[idx];
```

```
        }

void gainfactor_calculate (void){

        if ((enhancement4a) == 1){
        G[idx] =  (((lambda) > (1-(N[idx]/x))) ? (lambda) : (1-(N[idx]/x)));        //Frequency-dependent gain factor
        }
        if ((enhancement4b) == 1){
        G[idx] =  (((lambda*N[idx]/x) > (1-(N[idx]/x))) ? (lambda*N[idx]/x) : (1-(N[idx]/x)));
        }
        if ((enhancement4c) == 1){
        G[idx] =  (((lambda*P[idx]/x) > (1-(N[idx]/x))) ? (lambda*P[idx]/x) : (1-(N[idx]/x)));
        }
        if ((enhancement4d) == 1){
        G[idx] =  (((lambda*N[idx]/P[idx]) > (1-(N[idx]/P[idx]))) ? (lambda*N[idx]/P[idx]) : (1-(N[idx]/P[idx])));
        }
        if ((enhancement4e) == 1){
        G[idx] =  (((lambda) > (1-(N[idx]/P[idx]))) ? (lambda) : (1-(N[idx]/P[idx])));
        }
        if ((enhancement5) == 1){
        G[idx] =  (((lambda) > (sqrt(1-N[idx]*N[idx]/x*x))) ? (lambda) : (sqrt(1-N[idx]*N[idx]/x*x)));
        }
}

/*void oversubtraction (void){ //300 and 3400

        N[idx] = (1.08 - (0.54 -0.46*cos (2*PI*(idx+128)/(NFREQ-1)/2.2)))*N[idx];
}
*/
void oversubtraction(void){
        snr = (x/N[idx]-1);
        if(idx<NFREQ/2){                 //for lower frequencies
        beta = 2.65-0.0165*snr*snr;            //calculation formula
        if(beta>2.65) beta = 2.65;              //clamp beta in range [1,5]
        if(beta<1) beta = 1;
        }
        else{                           //for higher frequencies
        beta = 2.5-0.015*snr*snr;       //calculation formula
        if(beta>2.5) beta = 2.5;        //clamp beta in range [1,2.5]
        if(beta<1) beta = 1;
        }
        N[idx] = beta*N[idx];           //multiplied to noise estimate of current frequency bin

}
void residual_noise_reduction (void){

        float a, b, c, m;
        if((N[idx]/x)>residue){                      //do this only when |N/X| is smaller than some threshold
                a = cabs(previous[idx]);        //magnitude of previous output
                x = cabs(future[idx]);      //obtain future output using same noise estimation
                gainfactor_calculate();
                b = cabs(rmul(G[idx],future[idx]));
                c = cabs(outframe_temp[idx]);
                m = min(min(a,b),c);                                            //get the minimum
                if(a == m) outframe_temp[idx] = previous[idx];    //update current output with output of minimum magnitude
                else if(b==m) {                                                 //in thrre adjacent frames
                outframe_temp[idx] = rmul(G[idx],future[idx]);
        }
}
}

void nonspeechattenuation(void){
        int j;                                                                                  //check the energy measure and update
        if(T<0.789){                                                                    //the output of each frequency bin based
                for(j=0; j<NFREQ; j++){                                                         //on the defined algorithm
                        outframe_temp[j] = rmul(0.0316, inframe_temp[j]);
                        outframe_temp[FFTLEN-j] = conjg(outframe_temp[j]);
                }
        }
}
```

Matlab files for spectrogram plotting:

```
close all;

recObj = audiorecorder(8000,16,1); %Create 8000Hz,16Bit, 1-channel
                                    %audiorecorder object
disp('Start speaking.');
recordblocking(recObj, 40); %Record audio for 40s
disp('End of Recording.');
play(recObj);
y = getaudiodata(recObj);    %Returns recorded audio data associated
                             %with audiorecorder object
filename = 'benchmark_lynx2.wav';
audiowrite(filename, y, 8000,'BitsPerSample',16);%Writes matrix of audio data to file
```

Figure 24: Record speech

```
%Spectrograms plot
[s,fs] = audioread('benchmark_lynx2.wav');%Reads data from lynx2 and returns sampled data and sample rate
spgrambw (s, fs, 'pJcw');%Plot Spectrogram
```

Figure 25: Plot spectrogram

Spectrogram for speech enhancement results on "lynx 2":

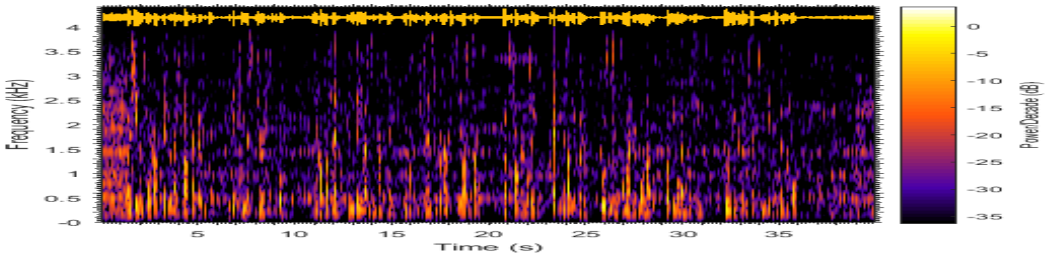


Figure 26: Lynx2 after enhancement 2 added to basic algorithm

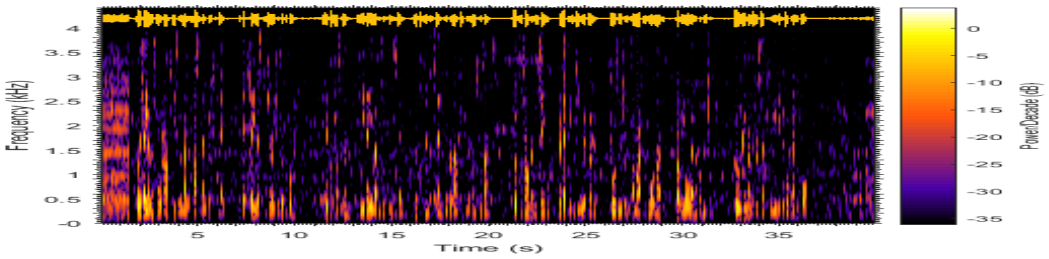


Figure 27: Lynx2 after enhancement 2 and 6 added to basic algorithm.

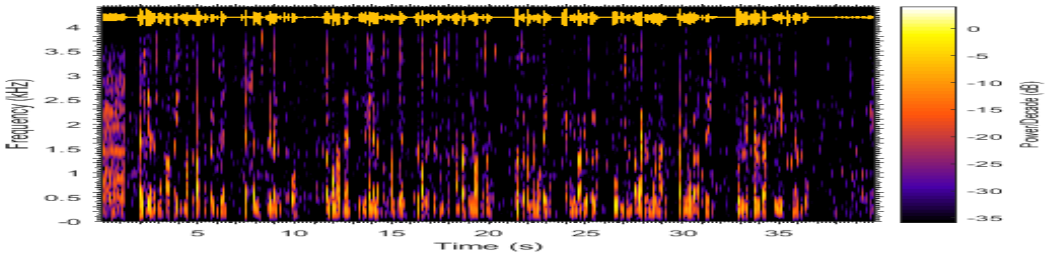


Figure 28: Lynx2 after enhancement 2,6 and 8 added to basic algorithm. Musical noise artifacts introduced by spectral subtraction are clearly reduced.