Imperial College London

Department of Electrical and Electronic Engineering

Real Time Digital Signal Processing

Project - Speech Enhancement

Authors:
Eusebius Ngemera CID:00825003
Prahnav Sharma CID:00820722

Course Lecturer:
Dr. Paul Mitcheson

Declaration: We confirm that this submission is our own work. In it we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offenses policy.

March 25th, 2016

Contents

1 Introduction

The aim of this project is to implement a real-time speech enhancement system, capable of accurately estimating and eliminating the background noise in a given speech signal. The spectral composition of the background noise is unknown, and the noise sources vary from speeding cars, to factory environments, to lynx helicopters and phantom jets.

1.1 Speech Enhancement Algorithm

The algorithm that will be employed to achieve this task is called Spectral Subtraction, and will be implemented in the frequency domain. There are two key operations that are undertaken by the algorithm namely 1) noise estimation and 2) noise subtraction.

For this algorithm to work, it relies upon two fundamental assumptions. First, we assume that the input signal, to our DSK system, will be a composition of speech and noise, which have been summed together. This assumption allows for the noise component of the signal to be removed, leaving the original speech. The second assumption is that the average human does not exceed 10 seconds of continuous speech, before needing to take a breath. This assumption will help determine the estimate for the background noise.

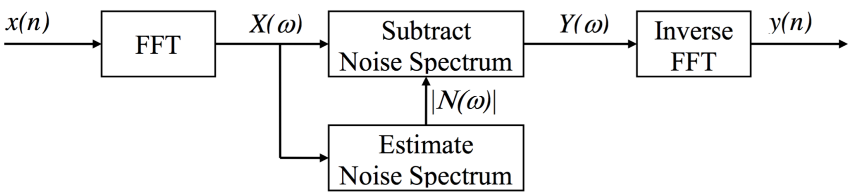

Figure 1: Spectral Subtraction Algorithm in Block Diagrams

The Spectral Subtraction algorithm is visually portrayed in figure 1. The Fast Fourier Transform, along with its inverse, are taken care of, and the focus of our project is to develop the noise estimation and noise subtraction procedures.

1.2 Input/Output Buffers

Processing 10 seconds of speech in one continuous attempt requires a huge amount of computational processing and storage; instead another alternative method is adopted. The 10 second range is divided into four 2.5 second ranges, and the incoming samples are stored in these buffers. These buffers are then rotated, post processing, and this process is repeated. With these 4 buffers, a minimum estimate for the noise can be obtained, within each buffer, and by estimating a minimum of the minimums, we can obtain a minimum estimate for the noise for the entire 10 seconds.

A further three buffers are required to carry out the processing in real time. The first buffer will be used as the input buffer. The second buffer will be an intermediate buffer, where all the processing will occur. The final buffer will be the output buffer, storing the samples to be sent to the DSK board. All time-domain samples are stored in the first buffer. Once the input buffer is full, the entire buffer's worth of data will be transferred to the intermediate buffer. The input buffer simultaneously takes in the next input, and this process is repeated. Processed data in the intermediate buffer is sent to the output buffer, and input data is shifted to the intermediate buffer for processing. Thus at any given point in the operation, there are three distinct actions being done.

1.3 Overlap Add Processing

All of the processing is undertaken in the frequency domain, thus to efficiently process the input samples, we segregate them into overlapping sections called frames. To avoid the occurrence of discontinuities, we apply a windowing function in the time domain, before taking the FFT of the signal. The Hanning windowing function is used, given by:

$$\sqrt{(}1 - 0.8515cos(\frac{(2k+1)\pi}{N})) \qquad fdor \quad k = 0, 1, ...(N - 14(1)$$

In this project, the oversampling ratio is 4, as each frame begins a quarter of a frame later. Upon return to the time domain post processing, we again apply windowing. This process is illustrated in figure 2 :

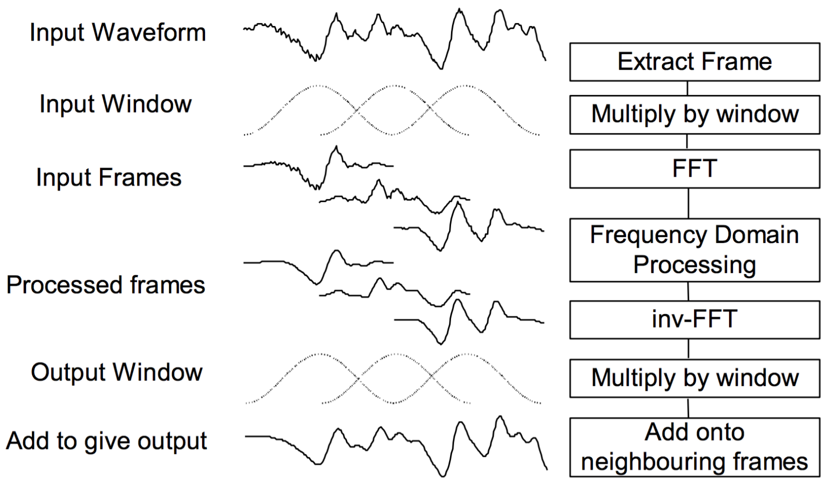

Figure 2: Overlap Add Processing

1.4 Noise Estimation and Subtraction

In the frequency domain, eliminating the noise is carried out by a simple subtraction:

$$Y(\omega) = X(\omega) - N(\omega) \tag{2}$$

However, as the phase of the noise is unknown, the subtraction is done for the magnitudes only, leaving the phase intact. This is carried out by $g(\omega)$, a frequency dependent gain factor

$$Y(\omega) = X(\omega) \cdot (\frac{|X(\omega)| - |N(\omega)|}{|X(\omega)|}) = X(\omega) \cdot (1 - \frac{|N(\omega)|}{|X(\omega)|}) = X(\omega) \cdot g(\omega) \tag{3}$$

2 Enhancements

2.1 Enhancement 1: Low-pass Filtered Input

A single-pole, low-pass filter is used to generate the first enhancement. Here, we seek to filter out the high frequency variation, which causes large fluctuations in the spectral magnitude. This is carried out prior to estimating our minimum noise, present in the buffers. This is a commonly used technique in mobile phones today, which significantly improves the quality of the speech signal in the presence of noise. For a time constant τ, and frame rate T, the single-pole, low-pass filter has a pole at k, given by:

$$k = exp(\frac{-T}{\tau}) \tag{4}$$

After low-pass filtering the input signal, our noise estimates in the buffers can increase in accuracy. The output formula is given by:

$$P_t(\omega) = (1-k)\cdot|X(\omega)| + k \cdot P_{t-1}(\omega) \tag{5}$$

The effect of the parameter τ, can be noted here. As $\tau \to 0, k \to 0 \therefore P_t(\omega) \to X(\omega)$. On the other hand, as $\tau \to 1, k \to 1 \therefore P_t(\omega) \to P_{t-1}(\omega)$. Thus, the value of τ is a design trade off because a higher τ helps to combat musical noise, at the cost of distorting the speech.

```
// Low-pass filter the input frame magnitude
if (e1 == 1){
    for (k = 0; k < FFTLEN; k++){
        lpf_input[k] = (1-k_pole)*original_mag[k] + (k_pole*lpf_input[k]);
    }
    update_minimums(lpf_input);
}
```

When this enhancement was implemented, we observed a very noticeable removal of background noise. This enhancement alone had a substantial effect in removing noise, relative to other enhancements.

2.2 Enhancement 2: Low-pass Filtering in Power Domain

Enhancement 2 is a minor variant of enhancement 1, where instead of low-pass filtering our signal with regards to signal magnitude, we filter with respect to the signal power. We aim to reduce the high frequency variation of the signal power, within each frame. Thus, the governing equation modifies to:

$$P_t(\omega) = \sqrt{(1-k)\cdot|X(\omega)|^2 + k\cdot(P_{t-1})^2(\omega)} \tag{6}$$

Carrying out the low-pass filtering in the power domain is desirable because the human ear is more sensitive to changes in the power spectrum, over changes in signal magnitude [1].

[1] http://www.dspguide.com/ch22/1.htm

```
// Low-pass filter the input frame power
if (e2 == 1){
    for (k = 0; k < FFTLEN; k++){
        lpf_input[k] = sqrt((1-k_pole)*original_mag[k]*original_mag[k] + \
            (k_pole*lpf_input[k]*lpf_input[k]));
    }
    update_minimums(lpf_input);
}
```

2.3 Enhancement 3: Low-pass Filtered Noise

Enhancement 3 follows on from the previous two enhancements, where the low-pass filtering is now undertaken on the minimum noise estimation to avoid abrupt discontinuities. Consider a case where a low frequency noise signal suddenly fluctuates to a high frequency and returns. In such an event, the estimation for the noise minimum would change rapidly and this would result in discontinuities between frequency bins, when the noise is subtracted. Low-pass filtering the large noise variation is a solution to resolve this complication.

$$P_t(\omega) = (1-k) \cdot |N(\omega)| + k \cdot P_{t-1}(\omega) \tag{7}$$

```
// Low-pass filter the noise estimate magnitude
if (e3 == 1){
    for (k = 0; k < FFTLEN; k++){
        lpf_noise[k] = (1-k_pole)*noise[k] + (k_pole*lpf_noise[k]);
    }
    noise_mag = lpf_noise;
}
```

As an additional improvement to this enhancement, we seek to low-pass filter the variation in the minimum noise estimate power. While testing, we observed that this enhancement did not contribute any significant improvement, suggesting that the noise in our test files were not varying significantly.

$$P_t(\omega) = \sqrt{(1-k) \cdot |N(\omega)|^2 + k \cdot (P_{t-1})^2(\omega)} \tag{8}$$

```
if (e3 == 2){
    for (k = 0; k < FFTLEN; k++){
        lpf_noise[k] = sqrt((1-k_pole)*noise[k]*noise[k] + (k_pole*lpf_noise[k]*lpf_noise[k]));
    }
    noise_mag = lpf_noise;
}
```

2.4 Enhancement 4: Gain Factor

In enhancement 4, the gain factor is modified. Previously, the gain factor λ was initialised to a very small constant ($\lambda = 0.01$), however in enhancement 4, various options are suggested for the gain factor. In **4a**, the gain factor is proportional to the noise-to-signal ratio $\lambda \cdot \frac{N_\omega}{X_\omega}$, which implies that with more noise, the minimum gain will increase, which will help to reduce the noise. In **4b**, the minimum gain factor is proportional to the low-pass filtered signal, $\lambda \cdot \frac{P_\omega}{X_\omega}$, which will help in eliminating high frequency noise components, while still preserving the high frequency speech components, because the voice signal is low-pass filtered and not the noise. In **4c**, the low-pass filtered version for the minimum gain factor as well as the low-pass filtered version for the calculated gain factor are used: $max(\lambda \cdot \frac{N_\omega}{P_\omega}, 1 - \frac{N_\omega}{P_\omega})$. Finally in **4d**, the gain factor is re-set to λ, similar to enhancement 2. Of all the proposals in this enhancement, **4d** was observed to yield the best results.

```
switch(e4){
    case 2:
        factor = 1 - noise_mag[k]/original_mag[k];
        lambda_factor = lambda * noise_mag[k]/original_mag[k];
        break;

    // cases 3 to 5
        break;

    default: //1
        lambda_factor = lambda;
        factor = 1 - noise_mag[k]/original_mag[k];
}

// G(w)
mag[k] = max(lambda_factor, factor);
```

2.5 Enhancement 5: Subtraction in power domain

Enhancement 5 closely resembles enhancement 4, with the difference being the gain factor is modified in the power domain, rather than in the magnitude domain. After implementing each of the variations, it was found that they contributed very little to the improvement of the enhanced signal. Furthermore, there were some additional crackling sounds introduced, and the `cpufrac` was approaching its limit, due to the additional computation. Thus, this enhancement was neglected.

```
switch(e5){
    case 2:
        lambda_factor = lambda * sqrt(noise_mag[k]*noise_mag[k]/ (original_mag[k]*original_mag[k]));
        factor = sqrt(1 - noise_mag[k]*noise_mag[k]/(original_mag[k]*original_mag[k]));
        break;

    /// cases 3 to 5

    default: //1
        lambda_factor = lambda;
        factor = sqrt(1 - noise_mag[k]*noise_mag[k]/ (original_mag[k]*original_mag[k]));
}
```

2.6 Enhancement 6 Over-subtraction

Enhancement 6 aims to over-estimate the noise level for low frequency bins that have a poor signal-to-noise ratio (SNR). For a given frequency, if the SNR falls below a threshold, then the noise estimate is scaled up by a factor (`alpha_high` below). Musical noise arises when there are random spectral peaks in the frequency domain, which inevitably result in troughs. These valleys can increase the musical noise effect. Here, the affect of alpha can be noted, where and increased value of alpha helps to minimise the amplitudes of the peak. However, again a trade off arises between attenuating the musical noise, and distorting the speech signal.

```
if (e6 == 1){
    SNR = input[k]/ (min_noise); // (computationally) simplified SNR value
    if (SNR < SNR_threshold){
        // increase estimate of the noise by alpha_high scaling
        noise[k] = alpha * alpha_high * min_noise;
    }
    else{
        noise[k] = alpha * min_noise;
    }
}
else{
    noise[k] = alpha * min_noise;
}
```

2.7 Enhancement 7: Frame Length

Enhancement 7 seeks to enhance the speech by varying the frame length. Table 1 below illustrates the frame lengths tested and the results.

Table 1: Varying frame length

Frame length	Notes
128	Distorted speech
256	Default
512	No output

Having started with a frame length of 256, we reduced it to 128 and found our speech enhancement was worse off. Speech was distorted. We realised that 256 gave a good resolution in frequency domain, that made reliable inverse-FFT possible. On the other side of the scale, increasing 512 did not give us any output. We believe this was due to computation time exceeding the time requirement. Processing of each frame exceeded 16 ms (62.5 frames per second). Therefore frame length is a compromise between frequency resolution and frame computation time.

2.8 Enhancement 8: Residual Noise Reduction

Enhancement 8 aims to remove noise left after subtracting the estimate of the noise average from the original signal. The current processed frame is delayed further by a quarter to allow comparison between three adjacent output frames [2]. Frequency-wise, the output frame $Y(\omega)$, takes the complex value from the frame with the minimum magnitude, $|Y(\omega)|$, only if the noise fraction exceeds a threshold.

```
if (e8 == 1){
    if ((SNR_now[k]) > noise_threshold){
        // complex_array_now =
        // complex_with_smallest_mag(complex_array, complex_array_now, complex_array_prev)
        // three adjacent frames

        if (complex_mag(complex_array_now[k]) > complex_mag(complex_array[k])){
            complex_array_now[k] = complex_array[k];
        }
        if (complex_mag(complex_array_now[k]) > complex_mag(complex_array_prev[k])){
            complex_array_now[k] = complex_array_prev[k];
        }
    }
}
```

2.9 Enhancement 9: Detection Period

Various detection periods were tested. Table 2 illustrates our findings. A shorter detection period resulted in noise estimation adapting quicker, when background noise changed. However this came at the cost of increasing speech distortion with decreasing detection period. At 2.5 seconds, there's not enough time to catch non-speech activity, i.e. a pause to take a breadth occurs less often that this. A slightly smaller detection period coupled with low-pass filtered noise (enhancement 3) made a good combination.

Increasing the detection period maintained good noise cancellation, at the cost of slow noise readjusting.

Table 2: Varying frame length

Detection Period	Notes
2.5 s	Faster noise adaptation; distorted speech
10 s	Default: Good
15 s	Good noise subtraction; slow to take effect

2.10 Extra Enhancements

Having applied enhancements to reduce noise, we noticed the speech had also been attenuated in the process, albeit still comprehensible. In our extra enhancement, we focused on

[2]Boll,S.F.,"Suppression of Acoustic Noise in Speech using Spectral Subtraction", IEEE Trans ASSP 27(2):113-120, April 1979.

amplifying the speech. We apply a passband filter in the expected speech range as well as a low-pass filter with ansharp transition. Since the input frame is Fourier Transformed, the required filtering can be performed while processing in the frequency domain.

```
if (e_speech_gain == 1){
    // mirror the right side to the left
    if (k > (FFTLEN/2)){
        kk = FFTLEN - k;
    }
    else{
        kk = k;
    }

    // Pass-band filter: amplify the speech frequency range
    if (kk >= low_freq && kk <= high_freq){
        complex_array[k].r = complex_array[k].r * speech_gain;
        complex_array[k].i = complex_array[k].i * speech_gain;
    }

    // Sharp, low-pass filtering
    else if (kk >= cut_off){
        complex_array[k] =  cmplx(0.0, 0.0);
    }
}
```

2.11 Enhancements Employed

Of all the enhancements suggested, our final algorithm made use of enhancements 1,3, default 4, and 6. Together, we observed that these enhancements enabled our algorithm to tackle different aspects of the encountered noise. Enhancements 1 and 3 helped to remove high frequency variation, in the signal and noise estimate, while enhancement 6 contributed to hindering the musical noise.

```
int e1 = 1;
int e2 = 0;
int e3 = 1;
int e4 = 1; // 0 to 4
int e5 = 0; // 0 to 4
int e6 = 1;
int e8 = 0;

float lambda = 0.01;
float alpha = 4;
float tau = 0.030;            // milliseconds
float k_pole;                // Low-pass filter pole, exp(-TFRAME/tau)

// enhancement 6
float alpha_high = 3;        // alpha scaling for low SNR
float SNR_threshold = 6;     // threshold for alpha scaling; simply |X(w)|/|N(w)|

```

```c
// enhancement 8
float *SNR_now;                // array of the previous SNR values
float noise_threshold = 1;     // |X(w)|/|N(w)| threshold
int i_threshold = 3;

// Extra enhancement; e_speech_gain
// FFT integer indices
int low_freq = 15;
int high_freq = 35;
int cut_off = 41;
float speech_gain = 4.0;
```

3 Verification

3.1 Buffers

After listening to our filtered output, we then look at the buffers as a step in verification.

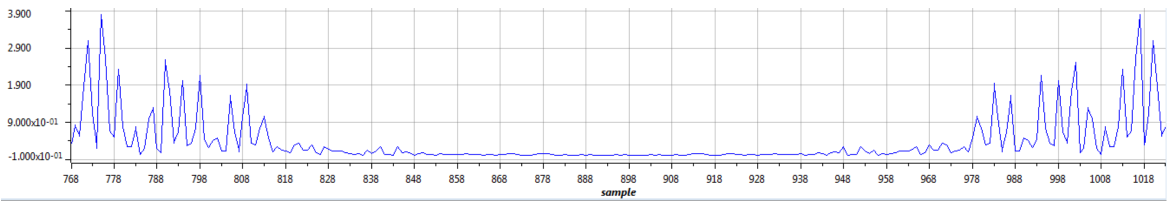

Figure 3: Input-frame frequency magnitude (lynx1 noise)

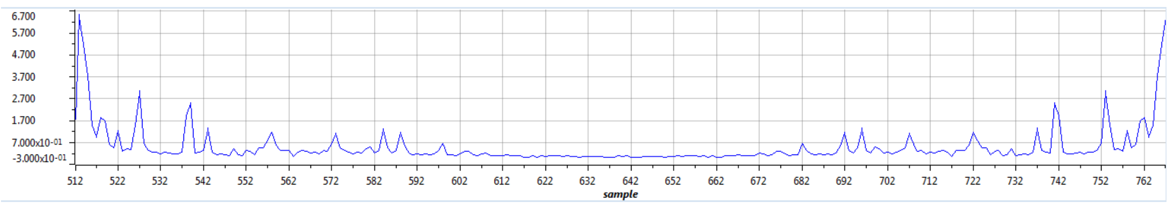

Figure 4: Input-frame estimated noise magnitude (lynx1 noise)

From the noise estimate above, we see the result from just using the minimum noise magnitude over 10 seconds. It shows a lot of noise less than 150 Hz. However calculating the inverse SNR, $\frac{|N(\omega)|}{|X(\omega)|}$, (enhancement 8) reveals a peak at 875 Hz. This peak is in turned attenuated, when compared to an inverse-SNR threshold.

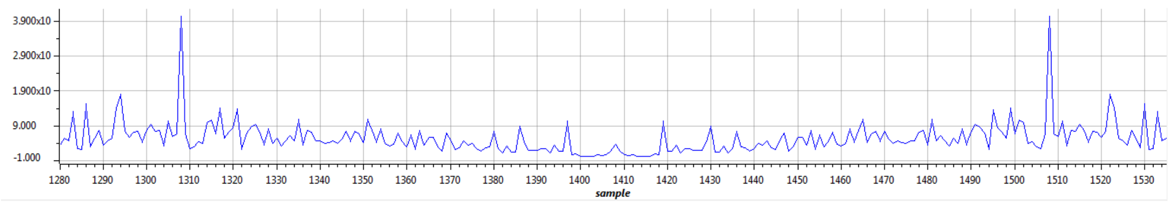

Figure 5: Input-frame simplified inverse-SNR: $\frac{|N(\omega)|}{|X(\omega)|}$ (lynx1 noise)

3.2 Spectrogram

To evaluate the performance of our speech enhancing algorithm, we use time-frequency analysis, in the form of a spectrogram. We compare and contrast two noise sources, namely the lynx helicopter and the phantom jet. For each case, we look at the speech signal with noise, the filtered speech signal with noise, and the noise free signal. These spectrograms are plotted in figures 6 and 7.

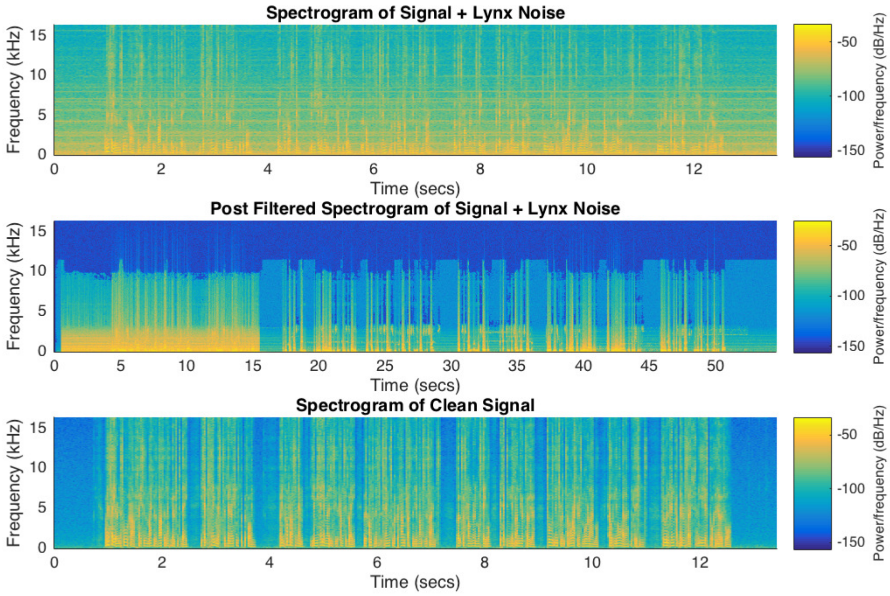

Figure 6: Real Time Processing on Signal + Lynx Noise

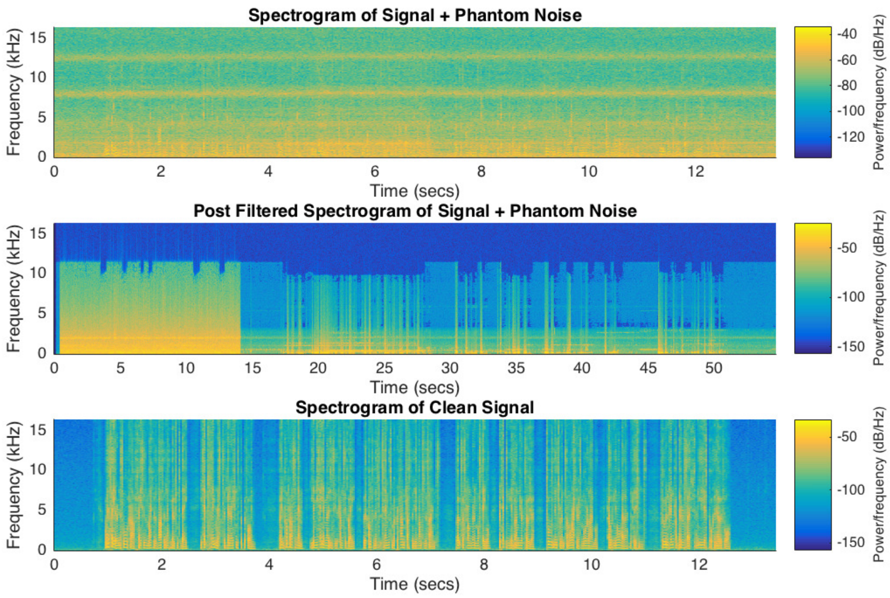

Figure 7: Real Time Processing on Signal + Phantom Noise

For both the spectrograms above, we see the filtered signal being affected by the DSK's anti-aliasing filter, which has removed very high frequency components. The first few seconds of both plots have high noise because the algorithm is in the process of estimating the noise, by going through each 2.5s bin. Thereafter, by looking at the colour of the plot, we can see that the noise has indeed been filtered to a significant extent.

Comparing the 2 background noises, we see that phantom4 input has higher noise magnitude in both high frequencies and speech frequency range. This is observed by the strong horizontal lines and the yellowing of the spectrogram to represent a higher magnitude. This results in our filtered signal still containing noise for the speech frequency range. Furthermore comparing our filtered signal to the clean signal, it is evident that higher harmonics of speech are cut down and hence produce a lower speech quality.

4 Conclusion

In conclusion, our speech enhancement algorithm was able to successfully estimate, and substantially remove, the noise component present in all of the test signals. It was clear that the overall quality of the output signals, were audibly improved by our algorithm, most notably with the constant hum of car, and the lynx helicopter. The factory signal presented a challenge, with the periodic banging noises causing a sudden increase in noise magnitude. For the phantom signal, the background noise amplitude was significantly reduced, however the musical noise was still present. Perhaps with an improved implementation to enhancement 6 and 8, this musical noise could be further eliminated.

5 Appendix - Full Readable Code

Speech enhancement C file ran on the DSK, `enhance.c`

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

                        EE 3.19: Real Time Digital Signal Processing
                            Dr Paul Mitcheson and Daniel Harvey
                            Prahnav Sharma and Eusebius Ngemera

                                  PROJECT: Frame Processing

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

          Utilises overlap-add frame processing (interrupt driven) on the DSK.
          Enhances speech in noisy signal through several enhancements

 *************************************************************************************
                               By Danny Harvey: 21 July 2006
                              Updated for use on CCS v4 Sept 2010
                              Added core funtionality to skeleton Mar 2016
 ************************************************************************************/
//  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*/
#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 50000.0                /* 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 min(a,b) (((a) < (b)) ? (a):(b))
#define max(a,b) (((a) > (b)) ? (a):(b))
#define cmplx_min(a,b) (((cabs(a)) < cabs(b)) ? (a):(b))
```

```
#define cmplx_max(a,b) (((cabs(a)) > cabs(b)) ? (a):(b))
/******************************* 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;

float *inbuffer, *outbuffer;           /* Input/output circular buffers */
float *inframe, *outframe;          /* Input and output frames */
float *inwin, *outwin;              /* Input and output windows */

// float arrays for G(w) and |X(w)|
float *mag, *original_mag;

// output Y(w)=X(w)*|G(w)|

// complex arrays to hold FFT's and IFFT's
// enhancement 8 involves history of |Y(w)|
complex *complex_array, *complex_array_now, *complex_array_prev;

// Bins of minimums in frequency-domain magnitude
float *M1, *M2, *M3, *M4;

float *noise;         // noise estimate magnitude, |N(w)|

// Low-pass filtered versions
float *lpf_input, *lpf_noise;

float *noise_mag;    // simply a pointer to the noise buffer to use

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 */

int detection_period = 10;     // period for detection of minimum noise
int frame_count = 0;         // keep track of how many frames processed for current Minimum bin

// Parameters
float lambda = 0.01;
float alpha = 4;
float tau = 0.030;             // milliseconds
float k_pole;                 // Low-pass filter pole, exp(-TFRAME/tau)

// enhancement 6
```

```
float alpha_high = 3;         // alpha scaling for low SNR
float iSNR_threshold = 6;     // threshold for alpha scaling; simply |N(w)|/|X(w)|

// enhancement 8
float *SNR_now;                 // array of the previous SNR values
float noise_threshold = 1;     // |X(w)|/|N(w)| threshold
int i_threshold = 3;

// Extra enhancement; e_speech_gain
// FFT integer indices
int low_freq = 15;
int high_freq = 35;
int cut_off = 41;
float speech_gain = 4.0;

// enhancement switches
int allpass = 0;
int e1 = 1;
int e2 = 0;
int e3 = 1;
int e4 = 1; // 0 to 4
int e5 = 0; // 0 to 4
int e6 = 1;
int e8 = 0;
int e_speech_gain = 0;

 /******************************* 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 */

void update_minimums(float* input);    // update M bins and estimate noise
float complex_mag(complex input);    // custom implementation of cabs(.)
/********************************** Main routine ************************************/
void main()
{
      int k; // used in various for loops

      k_pole = exp(-TFRAME/tau);    // Low-Pass Filter pole

    /*  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 */

    mag           = (float *) calloc(FFTLEN, sizeof(float));
    original_mag= (float *) calloc(FFTLEN, sizeof(float));

    complex_array = (complex *) calloc(FFTLEN, sizeof(complex));
    complex_array_prev = (complex *) calloc(FFTLEN, sizeof(complex));
    complex_array_now = (complex *) calloc(FFTLEN, sizeof(complex));

    // Minimum bins
    M1            = (float *) calloc(FFTLEN, sizeof(float));
    M2            = (float *) calloc(FFTLEN, sizeof(float));
    M3            = (float *) calloc(FFTLEN, sizeof(float));
    M4            = (float *) calloc(FFTLEN, sizeof(float));
    noise        = (float *) calloc(FFTLEN, sizeof(float));
```

```

    lpf_input     = (float *) calloc(FFTLEN, sizeof(float));
    lpf_noise    = (float *) calloc(FFTLEN, sizeof(float));
    SNR_now     = (float *) calloc(FFTLEN, sizeof(float));

    noise_mag = noise;

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

      /* initialize hardware interrupts */
      init_HWI();

    /* initialize algorithm constants */

      for (k=0; k<FFTLEN; k++)
    {
        inwin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
        outwin[k] = inwin[k];
    }

      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, kk;
    int io_ptr0;
    float factor, lambda_factor;
    complex *temp_ca;

    /* 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;
    for (k=0;k<FFTLEN;k++)
    {
        inframe[k] = inbuffer[m] * inwin[k];
        complex_array[k] = cmplx(inframe[k],0.0);    // copy inframe into a complex array
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }

    /************************ PROCESSING OF FRAME  HERE ************************/

    // put FFT in complex_array
    fft(FFTLEN, complex_array);

    for (k = 0; k<FFTLEN; k++)
    {
        if (e8 == 1){
            // store previous SNR values
            SNR_now[k] = noise_mag[k]/original_mag[k];
        }
        original_mag[k] = cabs(complex_array[k]);
    }

    /*********** enhancement 1 ****************/
    // Low-pass filter the input frame magnitude
    if (e1 == 1){
        for (k = 0; k < FFTLEN; k++){
            lpf_input[k] = (1-k_pole)*original_mag[k] + (k_pole*lpf_input[k]);
        }
        update_minimums(lpf_input);
    }
    /*********** enhancement 2 ****************/
    // Low-pass filter the input frame power
    else if (e2 == 1){
        for (k = 0; k < FFTLEN; k++){
            lpf_input[k] = sqrt((1-k_pole)*original_mag[k]*original_mag[k] + (k_pole*lpf_input[k]*lpf_input[
        }
        update_minimums(lpf_input);
    }
```

```c
    else{
        // default case
        update_minimums(original_mag);
    }

    /****************** enhancement 3 ****************/
    // Low-pass filter the noise estimate magnitude
    if (e3 == 1){
        for (k = 0; k < FFTLEN; k++){
            lpf_noise[k] = (1-k_pole)*noise[k] + (k_pole*lpf_noise[k]);
        }
        noise_mag = lpf_noise;
    }
    // Low-pass filter the noise estimate power
    else if (e3 == 2){
        for (k = 0; k < FFTLEN; k++){
            lpf_noise[k] = sqrt((1-k_pole)*noise[k]*noise[k] + (k_pole*lpf_noise[k]*lpf_noise[k]));
        }
        noise_mag = lpf_noise;
    }
    else{
        noise_mag = noise;
    }

    // noise subtraction for-loop
    for (k = 0; k < FFTLEN; k++)
    {
        // calculate G(w)
        // G(w) = max(lamba_factor, lambda)
        /************** enhancement 4 ***************/
        if (e5 == 0){
            switch(e4){
                case 2:
                    factor = 1 - noise_mag[k]/original_mag[k];
                    lambda_factor = lambda * noise_mag[k]/original_mag[k];
                    break;

                case 3:
                    lambda_factor = lambda * lpf_input[k]/original_mag[k];
                    factor = 1 - noise_mag[k]/original_mag[k];
                    break;

                case 4:
                    lambda_factor = lambda * noise_mag[k]/lpf_input[k];
                    factor = 1 - noise_mag[k]/lpf_input[k];
                    break;

                case 5:
                    lambda_factor = lambda;
                    factor = 1 - noise_mag[k]/lpf_input[k];
                    break;

                default: //1
                    lambda_factor = lambda;
                    factor = 1 - noise_mag[k]/original_mag[k];
            }
        }
        /*************** enhancement 5 ***************/
        else{
            switch(e5){
                case 2:
                    lambda_factor = lambda * sqrt(noise_mag[k]*noise_mag[k]/ (original_mag[k]*original_mag[k
                    factor = sqrt(1 - noise_mag[k]*noise_mag[k]/(original_mag[k]*original_mag[k]));
```

```
                    break;

                case 3:
                    lambda_factor = lambda * sqrt(lpf_input[k]*lpf_input[k]/ (original_mag[k]*original_mag[k
                    factor = sqrt(1 - noise_mag[k]*noise_mag[k]/ (original_mag[k]*original_mag[k]));
                    break;

                case 4:
                    lambda_factor = lambda * sqrt(noise_mag[k]*noise_mag[k]/ (lpf_input[k]*lpf_input[k]));
                    factor = sqrt(1 - noise_mag[k]*noise_mag[k]/ (lpf_input[k]* lpf_input[k]));
                    break;

                case 5:
                    lambda_factor = lambda;
                    factor = sqrt(1 - noise_mag[k]*noise_mag[k]/ (lpf_input[k]*lpf_input[k]));
                    break;

                default: //1
                    lambda_factor = lambda;
                    factor = sqrt(1 - noise_mag[k]*noise_mag[k]/ (original_mag[k]*original_mag[k]));
            }
        }

        mag[k] = max(lambda_factor, factor);
        // output Y(w)=X(w)*|G(w)|
        complex_array[k].r = mag[k] * complex_array[k].r;
        complex_array[k].i = mag[k] * complex_array[k].i;

        /************ Enhancement8 ***************/
        // complex_array is the processed FFT that will be delayed to output
        // complex_array_now is the FFT to be output now
        // complex_array_prev is the FFT of the previous output

        if (e8 == 1){
            if ((SNR_now[k]) > noise_threshold){
                // complex_array_now = complex_with_smallest_mag(complex_array, complex_array_now, complex_a
                // three adjacent frames

                if (complex_mag(complex_array_now[k]) > complex_mag(complex_array[k])){
                    complex_array_now[k] = complex_array[k];
                }
                if (complex_mag(complex_array_now[k]) > complex_mag(complex_array_prev[k])){
                    complex_array_now[k] = complex_array_prev[k];
                }
            }
        }

        /****** Extra enhancement: Frequency-domain filtering *******/
        if (e_speech_gain == 1){

            // mirror the right side to the left
            if (k > (FFTLEN/2)){
                kk = FFTLEN - k;
            }
            else{
                kk = k;
            }

            // Pass-band filter: amplify the speech frequency range
            if (kk >= low_freq && kk <= high_freq){
                complex_array[k].r = complex_array[k].r * speech_gain;
                complex_array[k].i = complex_array[k].i * speech_gain;
            }
```

```

            // Sharp, low-pass filtering
            else if (kk >= cut_off){
                complex_array[k] =  cmplx(0.0, 0.0);
            }
        }
    }

    // Enhancement 8: rotate array pointers of Y(w) history
    if (e8 == 1){
        temp_ca = complex_array;
        complex_array = complex_array_prev;
        complex_array_prev = complex_array_now;
        complex_array_now = temp_ca;

        ifft(FFTLEN, complex_array_prev);
    }
    else{
        ifft(FFTLEN, complex_array);
    }


    for (k=0; k<FFTLEN; k++)
    {
        if (allpass == 1){
            // copy input straight into output
            outframe[k] = inframe[k];
        }
        else if (e8 == 1){
            // enhancement 8: one frame delay
            outframe[k] = complex_array_prev[k].r; // _now before c
        }
        else{
            outframe[k] = complex_array[k].r;
        }
    }

    /**********************************************************************************/

    /* 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 */
          outbuffer[m] = outbuffer[m]+outframe[k]*outwin[k];
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }
    for (;k<FFTLEN;k++)
    {
        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;
}

/*************************************************************************************/

void update_minimums(float* input)
{
    int k;
    float *temp;
    float SNR, min_noise;

    if (frame_count == 0){
        // first frame input for this M bin (every 2.5s)
        for (k = 0; k < FFTLEN; k++){
            M1[k] = input[k];
        }
    }
    else{
        for (k = 0; k < FFTLEN; k++){
            // M1 = min(M1, input_frame_mag) for each f
            if (input[k] < M1[k]){
                M1[k] = input[k];
            }
        }
    }

    frame_count++;

    // if 10s of minumums collected
    if (frame_count >= ((int) (detection_period/(4*TFRAME))) ){
        frame_count = 0;

        // rotate pointers to Minimum bins
        temp = M4;
        M4 = M3;
        M3 = M2;
        M2 = M1;
        M1 = temp;

        // M1 is made to always be the current minimum bin

        k_pole = exp(-TFRAME/tau);     // Low-Pass Filter pole

        for (k = 0; k < FFTLEN; k++){
            min_noise = min(M1[k], min(M2[k],min(M3[k],M4[k])));

            if (e6 == 1){
                SNR = input[k]/ (min_noise); // (computationally) simplified SNR value
                if (SNR < iSNR_threshold){
                    // increase estimate of the noise by alpha_high scaling
                    noise[k] = alpha * alpha_high * min_noise;
                }
                else{
                    noise[k] = alpha * min_noise;
                }
            }
```

```
            else{
                noise[k] = alpha * min_noise;
            }

        }
    }
}

float complex_mag(complex c){
    return sqrt(c.r * c.r + c.i * c.i);
}
```

MATLAB was used for acquiring the various spectograms:

```
%% MATLAB Code for RTDSP Spectrograms
%-------------------------------------------------------------------------%
Fs = 44.1e3;

[Clean,Fs] = audioread('clean.wav');
[Lynx1,Fs] = audioread('lynx1.wav');
[Phantom4,Fs] = audioread('phantom4.wav');

[Lynx1_Recorded,Fs] = audioread('lynx1filtered.wav');
[Phantom4_Recorded,Fs] = audioread('phantom4filtered.wav');

%-------------------------------------------------------------------------%

figure
subplot(3,1,1)
spectrogram(Lynx1, hanning(1024), 0, 8192, 32768, 'yaxis');
title('Spectrogram of Signal + Lynx Noise');
subplot(3,1,2)
spectrogram(Lynx1_Recorded(:,1), hanning(1024), 0, 8192, 32768, 'yaxis');
title('Post Filtered Spectrogram of Signal + Lynx Noise');
subplot(3,1,3)
spectrogram(Clean, hanning(1024), 0, 8192, 32768, 'yaxis');
title('Spectrogram of Clean Signal');

figure
subplot(3,1,1)
spectrogram(Phantom4, hanning(1024), 0, 8192, 32768, 'yaxis');
title('Spectrogram of Signal + Phantom Noise');
subplot(3,1,2)
spectrogram(Phantom4_Recorded(:,1), hanning(1024), 0, 8192, 32768, 'yaxis');
title('Post Filtered Spectrogram of Signal + Phantom Noise');
subplot(3,1,3)
spectrogram(Clean, hanning(1024), 0, 8192, 32768, 'yaxis');
title('Spectrogram of Clean Signal');
```