Jurgen Shiqerukaj - CID 00938154 Jason Yuan – CID 00954645 Due 24th March 2017

Project – Speech Enhancement

Contents

Overview 2
Spectral Subtraction Overview 2
Noise estimation 3
Spectral Subtraction 4
Obtaining Filtered Output 5
Initial Performance Assessment and Analysis 6
Enhancement Outlook 7
Enhancement One – Using a Low pass Filter 7
Enhancement Two – Low Pass Filter in Power Domain 8
Enhancement Three – Low pass filtering Noise 8
Enhancement Four – Setting $G(\omega)$ and Scaling Lambda 9
Enhancement Five - $G\omega$ in power domain 10
Enhancement Six – SNR & Over Subtracting Noise 10
Enhancement Seven – Different Frame lengths 11
Enhancement Eight – Residual noise reduction 11
Enhancement Nine – Using a Shorter Window 12
Final algorithm performance 12
Conclusion 13
Appendix 14
References 14

Declaration note:

We confirm that this submission is our own work. We have credited and referenced any published, unpublished work by others. We fully understand that we are bound by college examination policies.

Jurgen Shiqerukaj, Jason Yuan

Overview

The purpose of this project is to implement a real-time speech enhancer which removes unwanted noise from a speech signal leaving behind only the useful speech signal. This a integral part of how voice communication systems manage noise, with modern advancements of technology there is little excuse for such systems have sufficiently high noise reduction techniques. In this project main focus will be on a technique called spectral subtraction, which is commonly used for noise removal in mobile phones. There are multiple enhancement that can be used to improve the existing performance of this technique in which all these will be evaluated.

Spectral Subtraction Overview

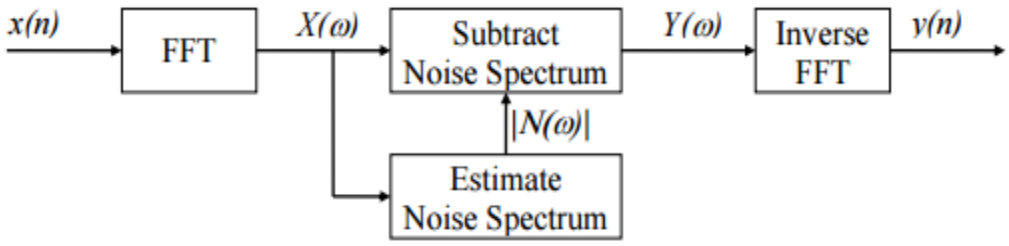

Figure 1: Block diagram to show how spectral subtraction works.

For spectral subtraction to work the FFT of the input signal must be taken to convert the signal from the time domain to the frequency domain. This is important as it allows processing to be done onto signal. The input signal is assumed to be the sum of the signal spectrum and noise spectrum. It is also worth to note that this technique does not improve on the phase spectrum of the signals as it is simply the magnitude of noise that is subtracted from the original signal.

```
for( i=0; i<FFTLEN ; i++)
       {
       C[i]=cmplx(inframe[i], 0);          /* input data into Buffer 'C' */
       }

fft(FFTLEN, C);                              /* perform the fft onto buffer 'C' */


for( i=0; i<FFTLEN ; i++)
       {
       absolute_C[i] = cabs(C[i]);    /* Find the absolute values of 'C' and place it
                                    into a new array for use in noise reduction*/
       }
```

For the processing to work in real-time, frames of data must be taken, however due to discontinuities at the frame boundaries this will result in unwanted frequency components when the FFT is taken on the frame, as a result the FFT is not the correct representation that is wanted. The approach to remove the

unwanted frequency component is to multiply the frame of data by a windowing function causing all these unwanted components to be removed. However, this affects the data in the time domain leading to a signal with varying amplitude. The solution to this is to overlap and add frames resulting in a much smoother signal, in this project an oversampling ratio of 4 is used.

Noise estimation

It is very difficult to detect whether is a signal is human speech or noise. One way of doing so is to design a voice activity detector (VAD), which detects whether speech is present within a signal, however it's extremely difficult to implement a reliable VAD thus an alternative approach was taken in this project.

The alternative approach is to assume the speaker will pause at least once during a 10 second interval in which during this pause the minimum amplitude of the background noise will be identified. This will result in an underestimation of the average noise within the 10 second interval thus the minimum noise is multiplied by a compensating factor known as **α**. This value **α** will be around 5-20 depending on the existing enhancement used.

The 10 second interval of the speech spectra could be stored in 1 buffer however this requires significantly more storage and processing thus the 10 second interval is split into 4 2.5 second intervals which results in faster processing but less accurate estimation of noise but this trade-off is worthwhile in real-time processing. This approach requires data to be shifted every 2.5 seconds, this is implemented below:

```
//Time count is defined as 312
if (time_counter >= time_count)
{

	time_counter=0; //Reset time_counter to 0

	//Pointer to rotate the set of data
	temp = M4;
	M4 = M3;
	M3 = M2;
	M2 = M1;
	M1= temp;

	//Making most recent M1 equal to the input
	for (i=0; i<FFTLEN; i++)
	{
		M1[i] = P[i];
	}

}
```

To ensure the rotation of data is done at correct time intervals time_count has to be calculated to a correct constant. It follows that a new set of data arrives every 8ms within a 2.5 second interval, around 312 sets

of data arrive resulting in the variable time_count to equal 312. The buffers are updated via the usage of pointers.

$$M_1(\omega) = \min(|X(\omega)|, M_1(\omega))$$

To find the minimum amplitude spectrum (background noise) within the last 10 seconds. The minimum of amplitude over the 4 buffers needs to be found, the equation and code implementation are below.

$$|N(\omega)| = \alpha \min_{i=1..4}(M_i(\omega))$$

```
//Updates the buffer with the information in the first 2.5s
for ( i=0; i<FFTLEN ; i++)
{
      M1[i] = min(M1[i],P[i]);
}
//Updates the noise buffer with the minimum amplitude spectrum over the past 10
seconds
for (i=0 ; i<FFTLEN ; i++)
{
      noise[i] = alpha*(min(min(M1[i], M2[i]),min(M3[i], M4[i])));
}
```

Spectral Subtraction

Following the noise estimation process, the noise associated with the signal is identified. It is necessary to proceed by subtracting it to the $X(\omega)$ resulting in a filtered output 'Y'. Mathematically it is

$$Y(\omega) = X(\omega) - Noise(\omega)$$

$$= X(\omega)\left(\frac{|X(\omega)| - |Noise\ (\omega)|}{|X(\omega)|}\right)$$

$$= X(\omega)\left(1 - \frac{|Noise\ (\omega)|}{|X(\omega)|}\right)$$

$$\therefore\ Y(\omega) = X(\omega)G(\omega) \quad where\ G(\omega)\ is\ a\ scaling\ factor.$$

However it is apparent that G(ω) can sometimes be negative if the noise has been over estimated ($|Noise(\omega)| > Absolute\ X(\omega)$). This could be especially problematic as it may result in a severely distorted output waveform. Hence in order to bypass this problem a floor value for $G(\omega)$ is set in the form of lambda (λ) which takes a value between 0.01 to 0.1. Mathematically this is implemented by:

$$\therefore\ G(\omega) = max\left\{\lambda, \left(1 - \frac{|Noise\ (\omega)|}{|X(\omega)|}\right)\right\}$$

The reason $\boldsymbol{\lambda}$ is not set to zero is because for a $\boldsymbol{\lambda}$ value of zero more musical noise is introduced (Rapid random musical notes in background) then if $\boldsymbol{\lambda}$ is just set slightly above zero. This has been implemented as below:

```
for( i=0; i<FFTLEN ; i++)
{

 g[i]=max(lamda*(P[i]/absolute_C[i]), 1-(noise[i]/absolute_C[i]));
/* Finds the scaling factor G(ω) of type float used for subtraction */

 Y[i] = rmul(g[i], C[i]);/* Does the multiplication to find the output Y of type complex */

}
```

To find the output $Y(\omega)$ the function `rmul()` is used which multiplies a type complex with a type float. As $G(\omega)$ is of type float and $X(\omega)$ is of type complex the `rmul()` function is necessary.

The procedure to find the two initial values of $\boldsymbol{\alpha}$ and $\boldsymbol{\lambda}$ is through trial and error. It was noticed that higher values of $\boldsymbol{\alpha}$ would more aggressively target and remove noise, however the downside of high $\boldsymbol{\alpha}$ is that it would start attacking the speech signals itself heavily distorting and attenuating the signal, hence a balanced $\boldsymbol{\alpha}$ value of 25 was used. Furthermore increasing $\boldsymbol{\lambda}$ would remove more musical noise, however it did give rise to persistence static/white noise, and it was found initializing it to 0.01 had optimal performance.

Obtaining Filtered Output

After obtaining filter values of $Y(\omega)$ in the frequency domain, it has to be converted back into the time domain $Y(n)$. This is a simple procedure and involves performing an inverse Fourier transform. $Y(n)$ is then passed through to the output. The code implementation of this is as follows:

```
ifft(FFTLEN, Y);  /* Finds the Inverse Fourier transform of Y(w) as type complex */

for (k=0;k<FFTLEN;k++)
      {
      outframe[k] = Y[k].r;  /* input Real part of Y(n) straight into output frame */
      }
```

It is important to note that the `ifft()` function here returns a complex number when in theory it should only return a real number. This is due to there being only finite accuracy of floating point numbers and hence a small imaginary component is introduced. This small imaginary component is to be considered as redundant, therefore only the real part of $Y(n)$ is taken.

Initial Performance Assessment and Analysis

After performing a few tests, it was apparent that noise was significantly reduced, however by no means completely eliminated. It is evident that background noise was attenuated and suggests some filtering was taking place.

It was found that the value of alpha had a monstrous effect on how aggressive filtering of noise would take place. Here are the results.

Alpha	Effect
$\mathbf{1 \leq \alpha < 10}$	No noticeable or significant filtering effect as noise is heavily under estimated, musical noise remains highly present.
$\mathbf{10 \leq \alpha < 20}$	A more noticeable range where noise starts to noticeably be reduced, however it is still not aggressive enough as musical noise is still clearly persistent.
$\mathbf{20 \leq \alpha < 40}$	Musical Noise is significantly reduced, alpha values near 20 performed better on low musical noise background such as Car1, lynx1 as they don't require much aggressive filtering whilst alpha values near the 40 mark performed better for high background noise signals such as phantom2, lynx2.
$\mathbf{\alpha \geq 40}$	When alpha is too high this caused large distortion and attenuation of the signal as the filtering procedure heavily overestimates noise present, removing large sections of the signal.

The optimal value was found to be around the $\boldsymbol{\alpha = 25}$ as it provided the best balance between removing optimal noise and keeping the integrity of the signal intact.

A very apparent limitation is that it is very difficult to get a static alpha value that will deal with different types of background noise. This was clear though our initial tests as low alpha values performed better for low musical noise and worse for loud musical noise and vise-versa. A solution to this would be a variable alpha value that is tailored to each varying input signal. This will be discussed further in enhancement 6.

Enhancement Outlook

The preliminary filtering procedure could be improved heavily upon. Enhancements that will included will be various ranging from having a variable alpha using SNR, using low pass filters and changing the time interval between shifting the data.

There will be various enhancements, in order to have an organized and structured program switches are used to effortlessly switch between various different enhancements. The switching mechanism used is a define if (`#if`) and depending what is defined at the top of the program using a series of 1's for 'on' and 0 for 'off', different enhancements or multiple enhancements can be activated at a time. Example Code implementation can be seen to the right.

```
/*Define enhancement switchs*/
#define enhance_1 0
#define enhance_2 1
#define enhance_3 1
#define enhance_4_1 0
#define enhance_4_2 0
#define enhance_4_3 1
#define enhance_4_4 0
#define enhance_5 0
#define enhance_6 1
#define enhance_8 0
                …
#if enhance_x == 1 // where x is
number from 1-8

/* code implementation example*/

 #endif
```

Enhancement One – Using a Low pass Filter

In this enhancement a low pass filter will be used on the varying components in the frequency bins of the magnitude spectrum. It's important to note that the low pass filtering will only be performed on the magnitude spectrum and not within the time domain. The transforming magnitude spectrum introduces large peaks that occur for a very small amount of time (High frequency peaks) which occurs when there is noise in the background. To reduce the effect of this a moving average low pass filter was used to slow down the transforming magnitude spectrum. It is also very important to note that a static value of alpha is used, this slowing will have the same effect on all frequency bins, addressed in enhancement 8. The frequency bins has a range of frequencies, these can be assumed to have a Gaussian distribution:

$$N(\mu, \sigma^2) \; where \; \mu \; mean, \sigma \; standard \; deviation$$

This enhancement allows the estimation buffers to contain low spectrum components.
For this enhancement, time constant values ranging from 20ms to 80ms were considered. To obtain the most optimal time constant it's important to understand the fundamentals of human speech. Humans can speak at varying rates, these range from 110 to 150 words per minute or from 145 to 160 for a professional speaker or podcaster. Therefore a word is said every 375ms to 550ms when engaging in speech. Due to this we can see that using a 80ms time constant will simply be just too high to reduce musical noise whilst a time constant value of 20 ms will provide the most optimal tradeoff between musical noise reduction and being able to fully represent the changes in a user's speech. The code implementation of this enhancement is shown below:

```
#if enhance_1 == 1    /*enhancement switch start*/

for( i=0; i<FFTLEN ; i++)
{

        /*implementation of the low pass filter */
        P[i] = (1-K_enhance1)* absolute_C[i] + K_enhance1* opt1_temp[i];
        opt1_temp[i] = P[i]; /* use previous value of P for next iteration*/

}
#endif /*Enhancement Switch End*/
```

After testing this enhancement amongst all sound templates, it was clear that there is a significant reduction of noise. However it does not perform aswell for noise that varies greatly in the background such as the factory sound files. It was also noticed that using a time constant of 80ms would distort the signal, as expected and a lower time constant would provide a better trade off. The optimal time constant was found to be 35ms which corresponds to a k value off.

$$k = e^{\frac{-T}{\tau}} = e^{\frac{-(\frac{256}{4})/8000}{35ms}} \approx 0.8$$

Enhancement Two – Low Pass Filter in Power Domain

This second enhancement is very similar to enhancement 1 but has a very major differences. The moving average low pass filter in enhancement 1 is linear whilst in the power domain it is non-linear. The effects of this is that within the power domain higher amplitudes has significantly more power than lower amplitudes due to the nature of the squaring, in the power domain it would be very easy to spot these sharp peaks as they will be amplified and as a result the low pass filter will correct them whilst having minimal effect on lower amplitudes. Mathematically this is implemented by:

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

Theoretically this should be an improvement on enhancement 1, as humans hear in terms of power providing better results to human ears. The code implementation of this is as follows:

```
#if enhance_2 == 1    /*enhancement switch start*/

for( i = 0;  i < FFTLEN ; i++)
{
    /*Implement the moving average filter, squaring Input C*/
P[i] = (1-K_enhance1)* absolute_C[i]* absolute_C[i] + K_enhance1* opt1_temp[i];
P[i] = sqrt(P[i]);      /*obtain the square root of P*/
opt1_temp[i] = P[i];    /*Store to previous value of P*/
}
#endif  /*Enhancement switch end*/
```

As the input `absolute_C` is being squared, it is required for there to be changes in the value of **k** and $\boldsymbol{\alpha}$ as it will not operate in the same manner as enhancement one, in practice this was incredibly difficult as there didn't appear to be any optimal combination between the two.

The performance of this enhancement was expected to be an improvement from enhancement 1. However it attenuated the speech signal heavily making it very difficult to hear, it also introduced a 'click' noise periodically. This is due to the addition of the square and square root, the complexity of enhancement two is that of $O(2N)$ which results in the enhancement running slower than enhancement one. Furthermore enhancement one had more audible difference in noise reduction than enhancement 2. After weighing the advantages and disadvantages for both enhancement 1 and 2, it's clear that enhancement one is an improvement and will be used for the final algorithm.

Enhancement Three – Low pass filtering Noise

Using the same method as enhancement 1, the aim of this enhancement is to avoid the sudden discontinuities of random background noise and hence reducing the effect of abrupt spikes have on the speech signal. The mathematical equation implementing this is:

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

This is implemented with the following code:

```
#if enhance_3 == 1    /*enhancement switch start*/

for ( i=0; i<FFTLEN ; i++)

{
    /*perform mathematical operation of low pass filter of noise */
noise[i] = (1-K_enhance1)* noise[i] + K_enhance1* opt3_temp[i];
opt3_temp[i] = noise[i]; /*Store Previous value of noise for next iteration*/
}
#endif /*Enhancement Switch Start*/
```

When this enhancement is tested there was a significant improvement in reducing noise that varied greatly, these included both factory1 and factory2 whilst not greatly effecting the speech signal. It is also important to note that this enhancement could have been expanded upon and just like enhancement two, perform the filtering in the power domain. However it was greatly suspected this would result in the same disadvantages as enhancement 2 thus was not implemented.

Enhancement Four – Setting $G(\omega)$ and Scaling Lambda

In this enhancement the floor value $G(\omega)$ will be adjusted to give different variable results. The purpose of this enhancement is that it's possible to adjust how much of the speech signal is lost due to lambda $\boldsymbol{\lambda}$ cutting out any signal due below the factor $G(\omega)$. Therefore by adjusting lambda by a scalar value it is clear how this cutting effect works. There are 4 different variations of $G(\omega)$.

Enhancement name	Enhancement	Effect on program
Enhance_4_1	$G(\omega) = max\left\{\lambda\frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}, 1-\frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}\right\}$	Little to no noticeable change.
Enhance_4_2	$G(\omega) = max\left\{\lambda\frac{\lvert P(\omega)\rvert}{\lvert X(\omega)\rvert}, 1-\frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}\right\}$	Little to no noticeable change.
Enhance_4_3	$G(\omega) = max\left\{\lambda\frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}, 1-\frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}\right\}$	Improvement on removing slightly more musical noise and echo on some sound files.
Enhance_4_4	$G(\omega) = max\left\{\lambda, 1-\frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}\right\}$	Slightly removes some music noise.

The final enhancement used was Enhance_4_3 which had a slight improvement. The code implementation of this is:

```
#if enhance_4_3 == 1 /*enhancement switch start*/
for(i= 0; i<FFTLEN ; i++)
{
g[i]=max(lamda*(noise[i]/P[i]), 1-(noise[i]/P[i]));
}
#endif       /*enhancement switch end*/
```

Enhancement Five - $G(\omega)$ in power domain

Similarly to enhancement 2, the gain factor $G(\omega)$ is found within the power domain using the equation:

$$G(\omega) = max\left\{\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|P(\omega)|^2}}\right\}$$

The code implementation is:

```
#if enhance_5 ==1  /*Enhancement Switch Start */
for (i = 0; i<FFTLEN; i++)
{
      /*Implementation of equation  */
      g[i] = max(lamda, sqrt(1 - (noise[i] * noise[i] / P[i] * P[i])));
}
#endif              /*Enhancement Switch End */
```

Upon testing this enhancement, the filtered signal was smoothened out resulting in a reduction in performance and quality of speech due to the increased difficulty in hearing the speech signal. Due to its inadequate performance, it will not be used over enhancement 4_3 in the final program.

Enhancement Six – SNR & Over Subtracting Noise

The aim of enhancement 6 is to remove spectral peaks. In previous enhancements a moving average lowpass filter was implemented to reduce the effects of spectral peaks by slowing down the rate of change of the magnitude spectrum as much as possible. A method to completely eliminate these rouge spectral peaks is to adjust alpha. The higher the alpha value the more spectral peaks are reduced, there is however a limit to how much you can increase alpha before the speech signal gets heavily distorted. Furthermore intensity of spectral peaks changes throughout hence resulting in certain alpha values being more optimal then other values, thus having a fixed value of alpha is suboptimal. The issue here stems from the knowledge that all spectral peaks are treated the same when ideally, large spectral peaks should be affected by higher value alphas and lower peaks affected by lower value alphas to maintain integrity of the speech signal.

A solution is to use SNR to scale alpha. SNR is a powerful tool used in analyzing the signal-to-noise ratio of a signal and can metaphorically be perceived as the ratio of useful information to miscellaneous/irrelevant information(noise). The SNR of a signal is simply:

$$SNR_{db} = 20\log\left(\frac{Signal}{Noise}\right)$$

Ideally using SNR, alpha should be scaled linearly. However this attempt didn't improve our existing performance by much thus an alternative custom approach was taken, which resulted in significant improvement in performance for loud background noise e.g. lynx, phantom.

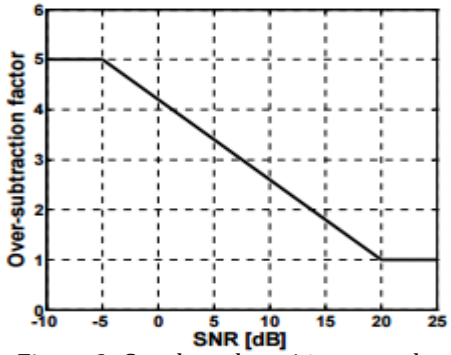

Figure 2: Graph to show 1st approach

```
#if enhance_6 ==1
for( i = 0 ; i <FFTLEN; i++)
{
	if(SNR > 20*log(P[i]/noise[i]))
	{
	SNR = 20*log(P[i]/noise[i]);
	}
}
	if (SNR <-15)
	{
		variable_alpha = 5*alpha/4;
	}
	else if (-15 <SNR < -5)
	{
		variable_alpha = alpha;
	}
	else if (-5 < SNR < 20)
	{
		variable_alpha = alpha/2;
	}
	else
	{
		variable_alpha = 1;
	}
#endif
```

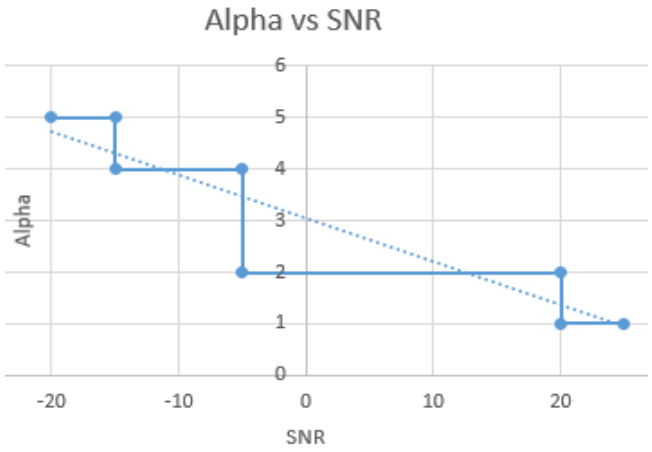

Figure 3: Graph to show alternative approach

The first 'if' statement calculates the lowest SNR value between all the frequency bins then using this lowest SNR a corresponding alpha is chosen.

Implementation of this custom enhancement can be viewed above. Alpha was adjusted according to SNR values as seen in the if statements. Upon testing this had significant audible improvements to largely varying SNR signals and will be implemented in the final algorithm.

Enhancement Seven – Different Frame lengths

Implementing this enhancement involved changing the frame length off [`#define FFTLEN 256`]. By adjusting the frame length it gave various results. By decreasing the frame length by half to 128 it caused the intensity of the speech signal to decreases drastically with added clicks and musical noise on top. The theory behind this is that decreasing the frame length results in greater quantization therefore resulting in the return of less number of fundamental frequencies. By decreasing the frame length it leads to increasing the frequency bin size. A length of 128 results in a frequency bin of 0 Hz to 62.5 Hz, length 64 gives 0 Hz to 125 Hz. It is apparent increasing the frequency bin range is undesirable as even though time resolution is good, it has poor frequency resolution. It's also worth noting that the frame length should be ideally a power of two in order to increase speed and efficiency when doing a FFT as processing speed is critical in real time processing.
Increasing frame length caused an amplified delaying effect between the input and the output and distorted the speech signal heavily. The reason for this is because the rate of change of the magnitude spectrum has been slowed down drastically.

Upon further testing, the most optimal frame length found was to be that of 256.

Enhancement Eight – Residual noise reduction

This enhancement is used to reduce the musical noise in the signal by checking if $\frac{|N(\omega)|}{|X(\omega)|}$ exceeds a certain threshold. If it does go above the threshold value set this means that the effect of noise on the original signal is relatively large, thus the next output $Y(\omega)$ is replaced with the minimum of the 3 adjacent frames resulting in reduction in musical noise. The C code for this is implemented below.

```
#if enhance_8 == 1     /*Enhancement Switch Start */

for ( i = 0; i < FFTLEN; i++) /*shifting  previous outputs*/
{
       Y_past2[i] = Y_past1[i];
       Y_past1[i] = Y[i];
}
#endif


for (i =0; i<FFTLEN; i++)
{
       Y[i] = rmul(g[i],C[i]);
}

#if enhance_8 == 1
for ( i = 0; i < FFTLEN; i++)
{
       if (noise[i]/P[i] > residue) //check if the ratio of noise to signal is
                                                    //above residue value
       {
       compareY = min(min(cabs(Y[i]),cabs(Y_past1[i])),cabs(Y_past2[i]));
       //Find the minimum between the current and past 2 outputs
              if(compareY == cabs(Y_past1[i])) //if past1 is minimum

              {
                     Y[i] = Y_past1[i];
              }
              else if(compareY == cabs(Y_past2[i])) //if past2 is minimum
                                                   //make output equal to past2
              {
                     Y[i] = Y_past2[i];
              }
       }
}
#endif      /*Enhancement Switch end */
```

Enhancement Nine – Using a Shorter Window

The aim of this enhancement is to use a shorter window where the noise estimates and SNR values are updated at a faster rate to improve upon noise estimations and current SNR values to adjust alpha. Initially the window was updated every 2.5s with filtering occurring after an initial 10 seconds. This limited real world application as people don't wait 10 seconds before speaking. This is implemented very easily through:

```
if (time_counter >= time_count/4)  // Window shorter by a factor of 4
```

With an iterative testing procedure on various window lengths we found that scaling 'time_count' by a factor of 4 (like seen above) provided the best estimate of background noise. This is because the first 2-4 seconds of each sound file contained just pure noise, hence it allowed the algorithm to more accurately identify and assess the noise where as a longer window would provide a less accurate noise estimation due to the speech signal being mixed in with the background noise. This shorter window frame would also work well in a real world environment as initially the caller doesn't speak for roughly 2-3 seconds allowing for optimal noise estimation.

Final algorithm performance

After thoroughly testing all enhancements the final enhancement that was used are as follows: 1 (low pass filtering the input in the magnitude domain), 3 (low pass filtering the noise in the magnitude domain), 4_3, 6 (changing the value of alpha depending on the lowest SNR value) and 9 (shortening the

time interval shifting data). Enhancement 1 & 3 were used to help remove the spectral peaks from the speech signal that causes the musical noise. Additionally , enhancement 6 improves on this furthers by applying a suitable value of alpha based on SNR such that the estimation of the average noise is more accurate thus the speech signal does not get distorted or attenuated. Enhancement 9 decreases the window length by rotating the buffer and updating the SNR more frequently, this results in the noise cancelling algorthim to be performed quicker than the usual 10 seconds in addition it allows the algorthim to react quicker to a sudden change in background noise. Here are the results:

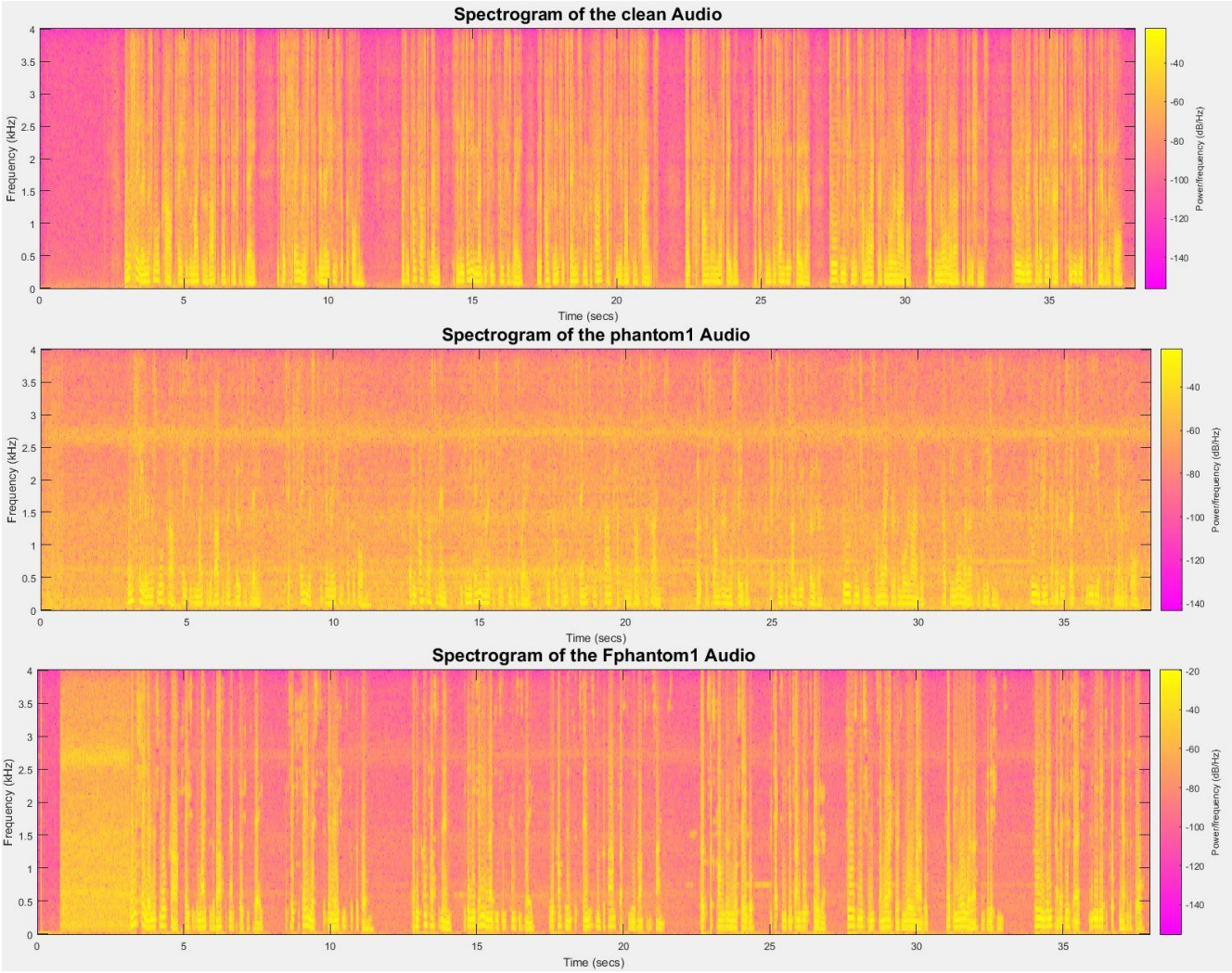

Figure 4 : Spectrogram of clean, phantom1 and filtered phantom1 respectively. We can see that phantom one unfiltered has alot of musical noise as you can be seen by their frequency intesitys in the spectrogram. Phantom filtered has alot musical noise removed and is a lot cleaner and resembles closely to that of the clean version. We can also see that our filter begins filtering in around 2.5 seconds. Further filtered versions in APPENDIX.

Conclusion

In conclusion it is clear from the figure 4 that the final algorithm implemented work quite effectively as it removed a large amount of background noise. Comparing the clean signal and filtered signal one could tell the filtered version closely resembles the clean signal as oppose to before when all the speech signal was corrupted by the background noise.The final values of alpha and lambda used was 4 and 0.01 respectively for optimal performance. However, despite all the enhancements used it is impossible to remove all the noise but it is without a doubt an effective means for satisfactory noise reduction for mobile technology.

References

1. D. Mitcheson, P.P. (2014) *EE3-19 Real Time Digital Signal Processing*. Available at: https://bb.imperial.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_885311_1&course_id=_9509_1 (Accessed: 20 March 2017).

2. W. H. House, A. (2004) *The Convolution Sum for Discrete-Time LTI Systems*. Available at: http://www.eecg.toronto.edu/~ahouse/mirror/engi7824/course_notes_7824_part6.pdf (Accessed: 20 March 2017).

3. Berouti,M. Schwartz,R. & Makhoul,J.,"Enhancement of Speech Corrupted by Acoustic Noise", Proc ICASSP, pp208-211, 1979.(Accessed: 20th March 2017).

Appendix

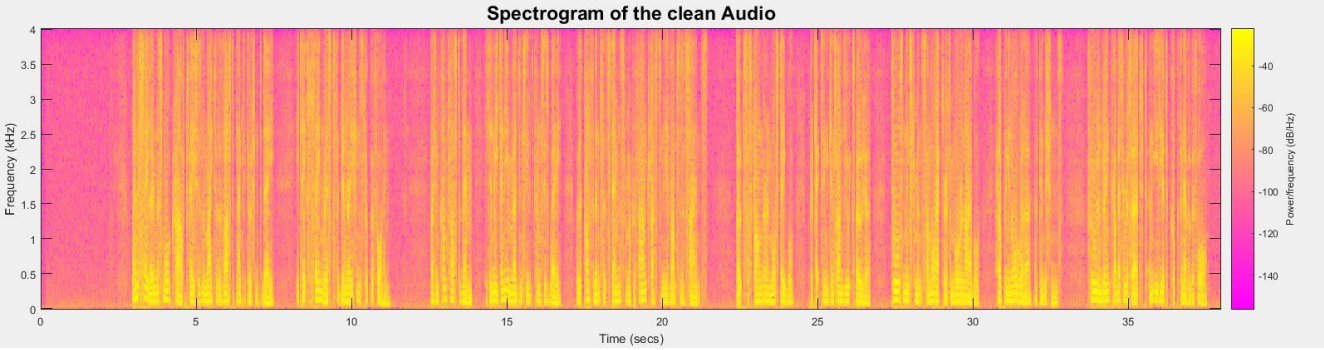

Figure 5: Figure shows that the frequency spectrogram of the clear audio file

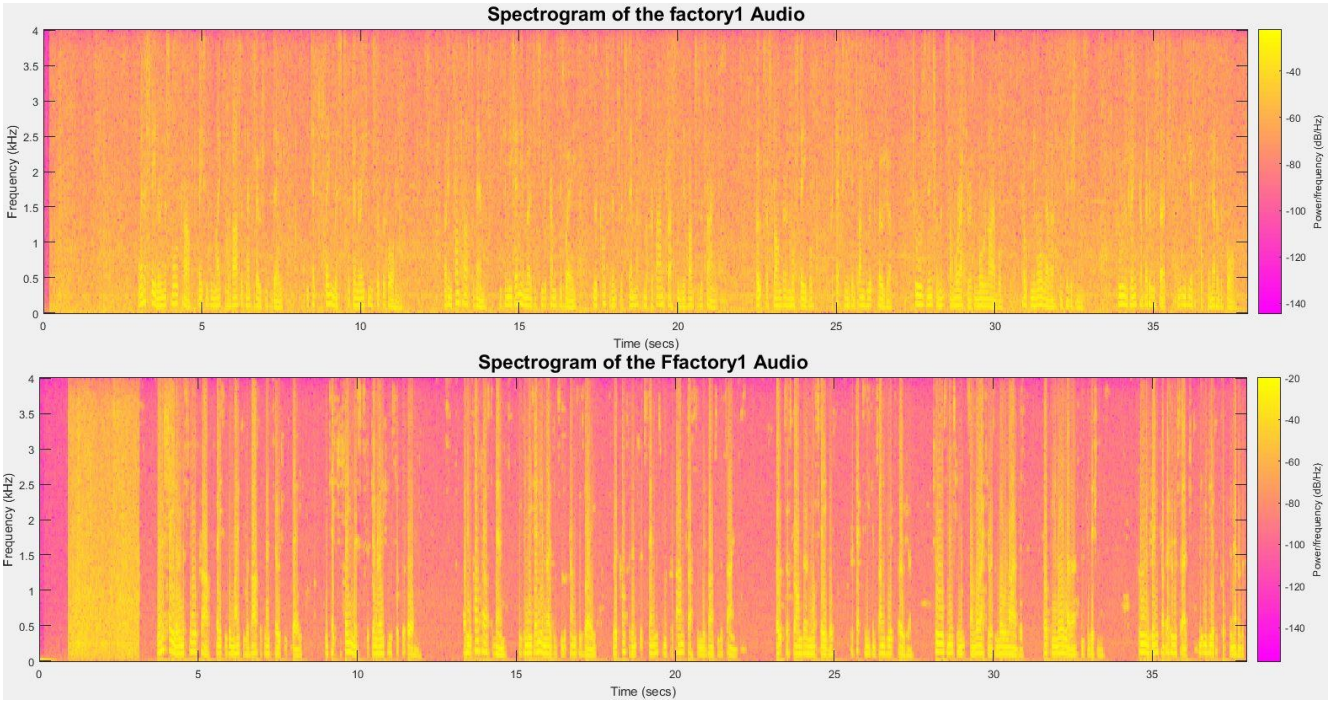

Figure 6: Figure shows Factory one unfiltered, wee can see it contain noise, and below is it filtered version off factory.

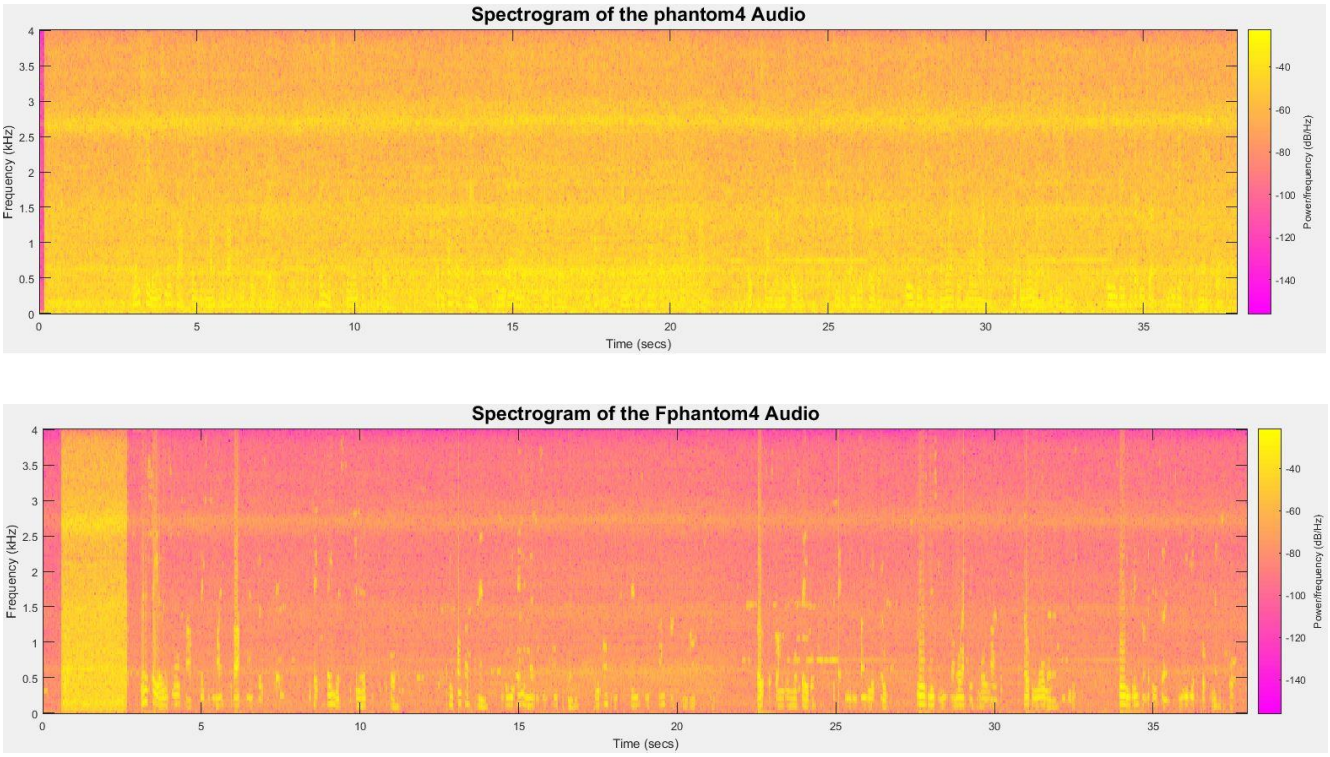

Figure 7: Figure shows Phantom4 one unfiltered, wee can see it contain HEAVY noise, and below is it filtered version off Phantom4, we can see coorect filtering has occurred but due to the heavily load of noise some of the speech has to be attenuated or cut off. Despiste this it filtered relatively well and removed as much noise as possible whilst maintaing the intrigity of the speech..

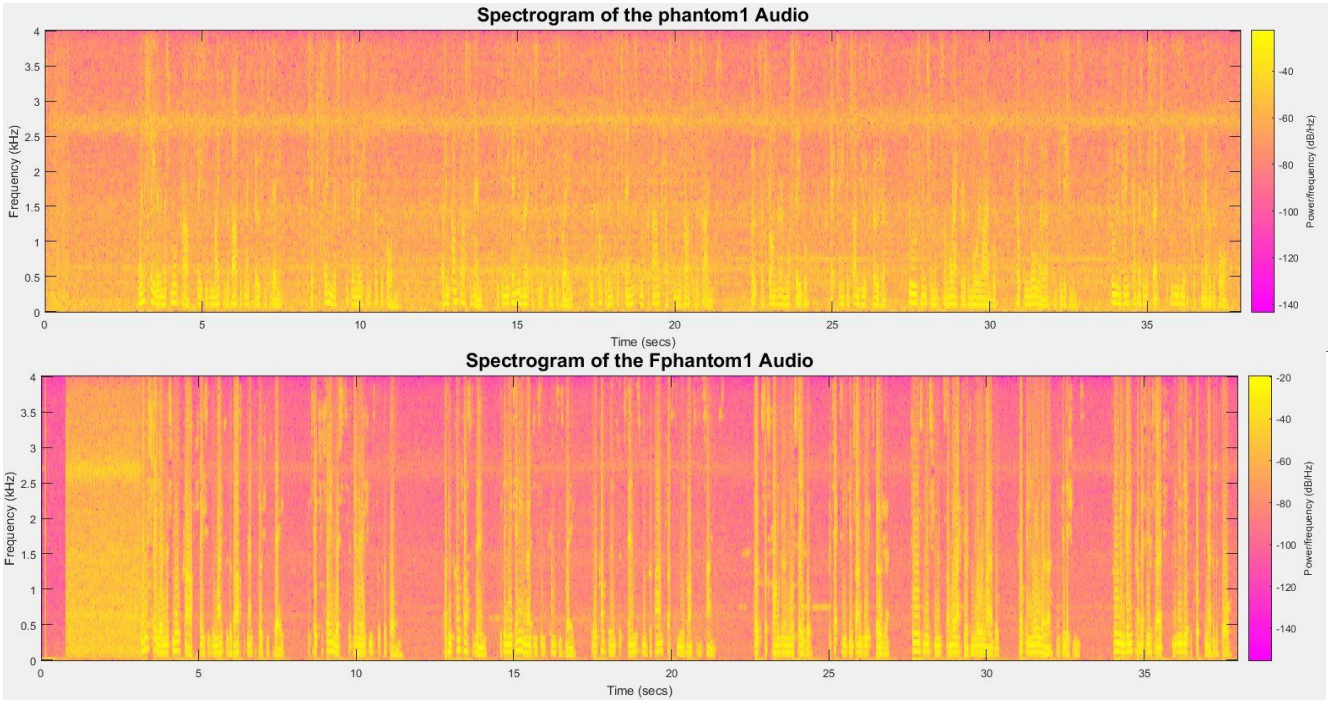

Figure 6: Figure shows phantom1 unfiltered, wee can see it contain noise, and below is it filtered version off phantom.

Code:

```
//  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 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 time_count 312
#define alpha 4            /* random number to start with*/
#define lamda 0.01
#define tau 0.02
#define float_max   2147483647
#define K_enhance1 0.8
#define residue 10
/*****************************Enhancement
switches*********************************/
#define enhance_1 0          //Good.
#define enhance_2 1                  //Reduces volume too much.
#define enhance_3 1                  //Good, helps remove large fluctuating noise.
#define enhance_4_1 0          //
#define enhance_4_2 0
#define enhance_4_3 1          //Good.
#define enhance_4_4 0          //Good, near same performace as above.
#define enhance_5 0
#define enhance_6 1
#define enhance_8 0
```

```
/****************************** 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 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 *C;
float *absolute_C;
float *opt1_temp;
float *opt3_temp;
float *noise;
float *min_estimate_M1;
float SNR = 0;
float *min_estimate_M2;
float *min_estimate_M3;
float *min_estimate_M4;
float *P;
float *temp;
```

```
complex *Y;
complex *Y_past1;
complex *Y_past2;
float       *g;
float *temp;
/* static variables */
int i =0;
int k=0;
int SNR_max= 0 ;
int time_counter =0;
int start_up_count = 0;
int flag = 0;
int variable_alpha = alpha;
float compareY;
 /****************************** 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 );  /* Find min value of two numbers*/
float max (float a, float b );  /* Find max value of two numbers*/
/********************************* Main routine
*********************************/
void main(){

      int k; // 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
*/
      C                 = (complex *) calloc(FFTLEN, sizeof(complex));
      absolute_C  = (float *) calloc(FFTLEN, sizeof(float));
      min_estimate_M1   = (float *) calloc(FFTLEN, sizeof(float));
      min_estimate_M2   = (float *) calloc(FFTLEN, sizeof(float));
      min_estimate_M3   = (float *) calloc(FFTLEN, sizeof(float));
      min_estimate_M4   = (float *) calloc(FFTLEN, sizeof(float));
      temp  = (float *) calloc(FFTLEN, sizeof(float));
      noise = (float *) calloc(FFTLEN, sizeof(float));
      Y     = (complex *) calloc(FFTLEN, sizeof(complex));
      g     = (float *) calloc(FFTLEN, sizeof(float));
      P     = (float *) calloc(FFTLEN, sizeof(float));
      opt1_temp = (float *) calloc(FFTLEN, sizeof(float));
      opt3_temp = (float *) calloc(FFTLEN, sizeof(float));

      /* 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;
      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;
    for (k=0;k<FFTLEN;k++)
      {
            inframe[k] = inbuffer[m] * inwin[k];
            if (++m >= CIRCBUF) m=0; /* wrap if required */
      }

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


      for( i=0; i<FFTLEN ; i++)
            {
                  C[i]=cmplx(inframe[i], 0);         /* input data into Buffer 'C'
*/
            }

      fft(FFTLEN, C);
                                          /* perform the fft onto buffer 'C' */


      for( i=0; i<FFTLEN ; i++)
      {
                  absolute_C[i] = cabs(C[i]);      /* Find the absolute values of
'C' and place it into a new array for use in noise reduction*/
      }

/************************************************Enhancement 1
start*****************************/
      #if enhance_1 == 1

      for( i=0; i<FFTLEN ; i++)
            {
                  P[i] = (1-K_enhance1)*absolute_C[i] + K_enhance1*opt1_temp[i];
                  opt1_temp[i]= P[i];
```

```
            }
      #endif

/*************************************************Enhancement 1 End
****************************/
/*****************************************************************************
*****************/
/*************************************************Enhancement 2
start****************************/
      #if enhance_2 == 1

      for( i = 0;  i < FFTLEN ; i++)
      {
            P[i] = (1-K_enhance1)*absolute_C[i]*absolute_C[i] +
K_enhance1*opt1_temp[i];
            P[i] = sqrt(P[i]);
            opt1_temp[i] = P[i];
      }

      #endif

/*************************************************Enhancement 2 End
****************************/
/*****************************************************************************
*****************/


      if (time_counter >= time_count/4)
      {

            time_counter=0;

            temp = min_estimate_M4;
            min_estimate_M4=min_estimate_M3;
            min_estimate_M3=min_estimate_M2;
            min_estimate_M2=min_estimate_M1;
            min_estimate_M1= temp;

            for(i=0; i<FFTLEN; i++)
            {
                  min_estimate_M1[i] = P[i];
            }

            /*************************************************Enhancement 6
start****************************/
                  #if enhance_6 ==1
                  for( i = 0 ; i <FFTLEN; i++)
                  {
                        if(SNR > 20*log(P[i]/noise[i]))
                        {
                        SNR = 20*log(P[i]/noise[i]);
                        }
                  }


                        if (SNR <-10)
                        {
                              variable_alpha = 3*alpha/2;
                        }
                        else if(-10 <SNR < -5)
```

```
                        {
                              variable_alpha = alpha;
                        }
                        else if (-5 < SNR < 20)
                        {
                              variable_alpha = alpha/2;
                        }
                        else
                        {
                              variable_alpha = 1;
                        }
                  #endif
/*************************************************Enhancement 6 End
*****************************/

      }
            time_counter++;

            for( i=0; i<FFTLEN ; i++)
            {
                  min_estimate_M1[i] = min(min_estimate_M1[i],P[i]); /*Find
            minimun between first 2.5 secounds and absolute value*/
            }

            for(i=0 ; i<FFTLEN ; i++)
            {
            noise[i] =
variable_alpha*(min(min(min_estimate_M1[i],min_estimate_M2[i]),min(min_estimate_M3
[i],min_estimate_M4[i])));
            }

/*************************************************Enhancement 3
start*****************************/
            #if enhance_3 == 1
                        for ( i=0; i<FFTLEN ; i++)
                        {
                        noise[i] = (1-K_enhance1)*noise[i] +
K_enhance1*opt3_temp[i];
                        opt3_temp[i]= noise[i];
                        }
            #endif
/*************************************************Enhancement 3
start*****************************/

/*************************************************Enhancement 4.1
start*****************************/
            #if enhance_4_1 == 1
            for(i= 0; i<FFTLEN ; i++)
            {
            g[i]=max(lamda*(noise[i]/absolute_C[i]), 1-(noise[i]/absolute_C[i]));
            }
            #endif
/*************************************************Enhancement 4.2
start*****************************/
            #if enhance_4_2 == 1
            for(i= 0; i<FFTLEN ; i++)
            {
            g[i]=max(lamda*(P[i]/absolute_C[i]), 1-(noise[i]/absolute_C[i]));
            }
            #endif
```

```
/*************************************************Enhancement 4.3
start*****************************/
            #if enhance_4_3 == 1
            for(i= 0; i<FFTLEN ; i++)
            {
            g[i]=max(lamda*(noise[i]/P[i]), 1-(noise[i]/P[i]));
            }
            #endif
/*************************************************Enhancement 4.4
start*****************************/
            #if enhance_4_4 == 1
            for(i= 0; i<FFTLEN ; i++)
            {
            g[i]=max(lamda, 1-(noise[i]/P[i]));
            }
            #endif
/*************************************************Enhancement 4
end*******************************/
/*************************************************Enhancement 5
start*******************************/
            #if enhance_5 ==1
            for(i= 0; i<FFTLEN ; i++)
            {
            g[i]=max(lamda, sqrt(1-(noise[i]*noise[i]/P[i]*P[i])));
            }
            #endif
/*************************************************Enhancement 5
end*******************************/
/*************************************************Enhancement 8
start*****************************/
            #if enhance_8 == 1
            for ( i = 0; i < FFTLEN; i++)
            {
                  Y_past2[i] = Y_past1[i];
                  Y_past1[i] = Y[i];
            }
            #endif

            for(i =0; i<FFTLEN; i++)
            {
            Y[i] =      rmul(g[i],C[i]);
            }

            #if enhance_8 == 1
            if( SNR < -15)
            {
                  for ( i = 0; i < FFTLEN; i++)
                  {
                        if(noise[i]/P[i] > residue)
                        {
                              compareY =
min(min(cabs(Y[i]),cabs(Y_past1[i])),cabs(Y_past2[i]));

                              if(compareY == cabs(Y_past1[i]))
                              {
                                    Y[i] = Y_past1[i];
                              }
                              else if (compareY == cabs(Y_past2[i]))
                              {
                                    Y[i] = Y_past2[i];
```

```
                                    }
                                }
                            }
                        }
                        #endif
/*************************************************Enhancement 9
end*******************************/
                        ifft( FFTLEN, Y);

            for (k=0;k<FFTLEN;k++)
                        {
                        outframe[k] = Y[k].r;/* copy input straight into output */
                        }



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

          /* 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;
}

/*************************************************************************
***/
//Custom functions
```

```
float min (float a, float b )
{
	if(a < b)
	{
		return a;
	}
	else
	{
		return b;
	}
}

float max (float a, float b ){
	if(a<b){
		return b;
	}
	else
		return a;
}
```