RTDSP FINAL PROJECT

Abstract

Noise removal and speech enhancement are important in real-time applications. Using spectral subtraction, noise components can be removed from a spectrum to obtain the desired parts of a signal. Attempting this on a variety of different noise conditions, we analysed potential optimizations while considering the compromises in performance to find effective methods of removing noise.

Keyan Sadeghi Namaghi – CID 00861078

Mert Kaya – CID 00970695

Table of Contents

Introduction 2

Theory 2

Initial Implementation 3

FFT 3

Estimating the Noise Spectrum 3

Spectral Subtraction 4

Output Processing 5

Critical Analysis 5

Enhancements 5

Enhancement 1: Low-Pass Filter in Magnitude Domain 6

Enhancement 2: Low-Pass Filter in Power Domain 6

Enhancement 3: Low-Pass Filter Noise Estimate 7

Enhancement 4: Computation of $g\omega$ in Magnitude Domain 7

Enhancement 5: Computation of $g\omega$ in Power Domain 8

Enhancement 6: Overestimation of Noise Level 9

Enhancement 7: Adjusting Frame Lengths 9

Enhancement 8: Residual Noise Reduction 10

Enhancement 9: Minimum Spectrum Over a Shorter Period 10

Fully Optimised Performance 11

Conclusion 14

Introduction

Speech enhancements are a vital application of real-time processing aimed at reducing background noise to allow speech to be isolated even in noisy environments. This project is focus on implementing an algorithm known as spectral subtraction in C, highlighting potential enhancement methods by analysing both their advantages and disadvantages. This will ultimately create a single flexible algorithm that is able to deal with a multitude of different speech signals without the need for retuning.

Theory

Spectral subtraction takes place in the frequency domain and utilises the assumption that the spectrum of the input is simply the sum of the speech and noise spectra. It is important to note that although the noise is assumed to additive it is not necessarily white noise. The noise is estimated and then subtracted from the spectrum before being converted back into the time domain and outputted. The block diagram that illustrates this process is shown in figure 1.

It is important to note that while the magnitude spectrum of the signal is enhanced, the phase spectrum is left unchanged.

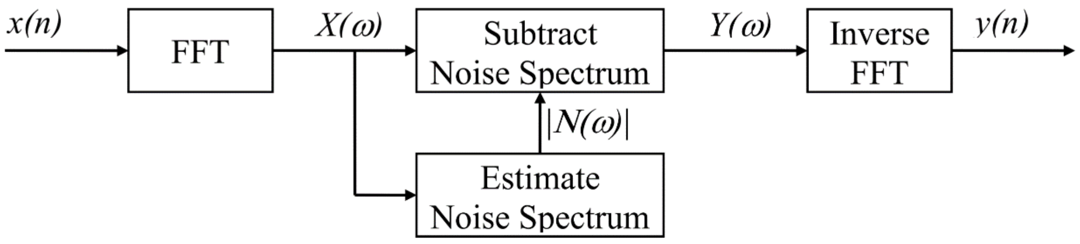


Figure 1: Block Diagram of Spectral Subtraction Process

An input is processed by the algorithm by separating it into frames so that it can be transformed into the frequency domain. From there, a series of square root Hamming windows are applied one quarter of a frame after the last, using the function W(k) = $\sqrt{1-0.85185\cos\left(\frac{(2k+1)\pi}{N}\right)}$. This is done because not using a Hamming window would result in discontinuities and simply using non-overlapping windows would result in attenuation at the edges of the frames as shown in figure 2. The length of the window is a compromise between accuracy and processing time, therefore a frame length of 256 was selected.

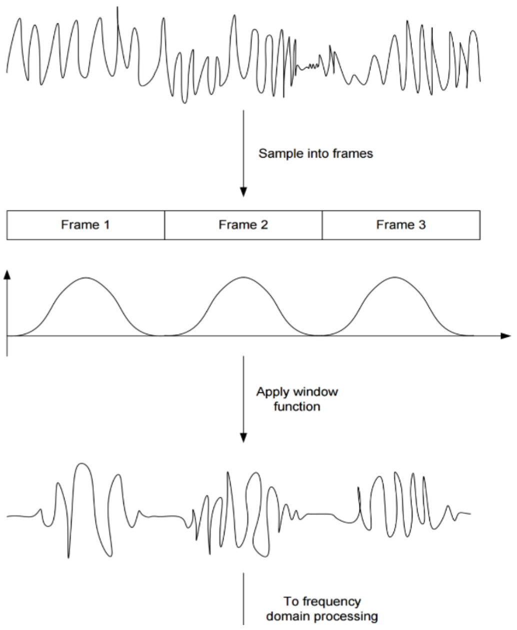


Figure 2: non overlapping Hamming windows attenuation

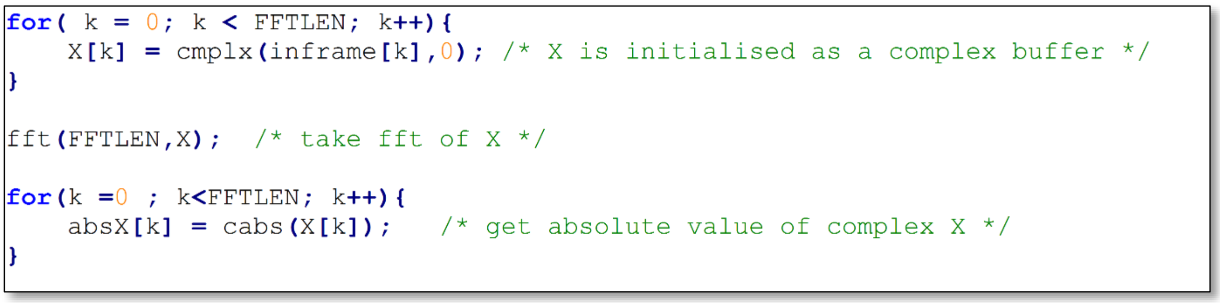
Noise estimation is done over a 10 second rolling window as it is presumed a speaker will pause within this period. The power during the section the speaker is paused will be at a minimum, therefore it stands to reason that the noise can be estimated by this minimum magnitude spectrum. A factor α is used to control how aggressive the approximation is at estimating the noise. Usually, the estimated noise value is far too low and needs to be scaled by α in the range of [2, 20].

Initial Implementation

A skeleton code was supplied which performed the input and output buffering as well as the overlapping addition algorithm required for the output. This code was then built upon to be able to implement the spectral subtraction.

FFT

The input data is first converted into a complex number with all the imaginary values set to zero as they are not within the scope of this course. The Fast Fourier Transform is then carried out using the *fft* function before the magnitude spectrum is extracted using the *cabs* function as shown in figure 3.



```
for( k = 0; k < FFTLEN; k++){
    X[k] = cmplx(inframe[k],0); /* X is initialised as a complex buffer */
}

fft(FFTLEN,X);  /* take fft of X */

for(k =0 ; k<FFTLEN; k++){
    absX[k] = cabs(X[k]);   /* get absolute value of complex X */
}
```

Figure 3: Code for FFT of input signal

Estimating the Noise Spectrum

The next stage is estimating the noise over the previously discussed 10 second window. The 10 second window is divided up into four 2.5 second windows to reduce latency. This works because the information is updated at four times the rate while still retaining the past 10 seconds of data. To better manage memory usage, not all the frames are stored, as only storing the minimum amplitude in each window still gives the same information.

Buffers that hold the minimum magnitude spectrum are initialised to FLT_MAX which is the maximum value for a float. This is to prevent the system from taking the full 10 seconds to identify the noise and begin the filtering. When all 4 buffers are filled up, they are rotated every 2.5 seconds such that the oldest data is reset to FLT_MAX. The *count* variable is used to ensure that the buffer is rotated at exactly the right time, and is reset with every rotation. The code is shown in figure 4.

```
/*Check if the buffers require rotating*/
if(count >= time_counter){
    count = 0;  /*Reset count*/

    /*Use pointer addresses to rotate buffers*/
    temp = M4;
    M4  = M3;
    M3  = M2;
    M2  = M1;
    M1  = temp;

    for ( k = 0 ; k < FFTLEN ; k++){
        M1[k] = FLT_MAX;    /*Set buffer to maximum*/
    }
}
count++;    /*Increment counter*/
```

Figure 4: Buffer rotation

Each of these 2.5-second buffers are then compared to find the minimum using a simple *min* function. This is then scaled using *alpha* so that the noise is increased to compensate for its underestimation. Figure 5 shows the implementation in C code.

```
for( k = 0; k < FFTLEN; k++){
    M1[k] = min(M1[k],absX[k]); /*Update buffer*/

    /*Find minimum noise over previous 10s*/
    noise[k] = alpha*(min(min(M1[k],M2[k]),min(M3[k],M4[k])));
}
```

Figure 5: Finding minimum then scaling by alpha

Spectral Subtraction

After the minimum noise has successfully been calculated, it then needs to be subtracted from the current frame. This is achieved by scaling the magnitude spectrum by the frequency-dependent gain factor, $g(\omega)$, using the following equation:

$$Y(\omega) = X(\omega) * g(\omega)$$

This gain factor can be calculated in several different ways as demonstrated in the enhancements section later on in this report, but the basic form to use the equation is as follows:

$$g(\omega) = max\left(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$$

The constant λ is added to ensure that $g(\omega)$ is always positive at any point. Without this constant and the maximum function, it would be possible for the gain factor to become negative. This is done in the code by using the *max* function and then using *rmul* function to perform the multiplication of a real and complex value. All of this is shown in figure 6.

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

    /*Calculate frequency dependent gain factor*/
    /*lambda ensuring g[k] is always positive*/
    g[k] = max(lambda, 1-(noise[k]/absX[k]));

    /*Scale the output*/
    Y[k] = rmul(g[k],X[k]); /*rmul is real*complex function*/
}
```

Figure 6: Frequency dependent gain factor

Output Processing

After the noise has been subtracted, the inverse Fourier transform is calculated using the *ifft* function. Figure 7 shown this code in C. Theoretically, Y should be real but in practice a complex number is returned. This is due to the finite precision of floating-point numbers so the code has to ensure to only take the real component.

```
/*Take inverse Fourier transform*/
ifft(FFTLEN,Y);

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

    outframe[k] = Y[k].r;   /*Output real part of Y*/
}
```

Figure 7: Output process

Critical Analysis

This basic spectral subtraction system does clearly filter some of the noise from the input. However, it is very clear that a significant amount of noise still remains and, in some of the samples, the processed speech is still difficult to understand.

One aspect of this code is that it is highly dependent on the value of alpha that is selected. This is an incredibly undesirable characteristic as it makes the system very inflexible. The algorithm always drastically underestimates the amount of noise so a large value of alpha has to be selected for the time being to correct this.

By itself, this system does not operate well enough to be particularly useful and see real-time application. As a result, this basic structure was then built upon with several enhancements to help remove the remaining noise.

Enhancements

The basic spectral subtraction method can theoretically be improved upon by the following optimisations. Some enhancements may actually have an adverse impact upon the performance when implemented while running the sample audio files which were provided; this will be explained later on in the report. In light of this, the final code has software switches which allowed the testing of every combination and the enhancements which, if ultimately unused, are simply switched off. These switches were implemented by *if* statements controlled by a variable initiated at the start.

It is important to note the meaning of musical noise. It is the sound caused by the isolated spikes in the spectrum after the spectral subtraction and other enhancements that sounds like musical sounds to the human ear. We will attempt to tackle this issue as well.

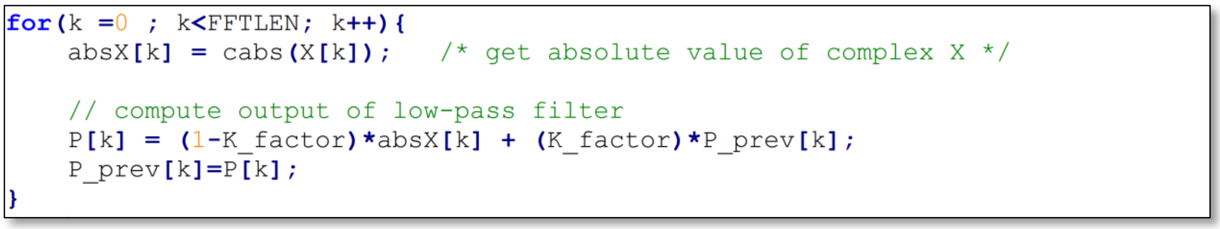
Enhancement 1: Low-Pass Filter in Magnitude Domain

The initial enhancement suggested by the literature is a low-pass filter of the input signal in the magnitude domain. Therefore, the Fourier transform of the input signal is still required, as this process cannot be carried out in the time domain.

Although over a prolonged time averages tend to have a relatively regular magnitude spectrum, in short samples, the magnitude behaves rather erratically. This filter removes these large changes in the magnitude spectrum, helping to reduce the amount of musical noise. The equation below demonstrates the mathematics behind how this filter was implemented:

$$P_t(\omega) = \left(1 - e^{\frac{-T}{\tau}}\right) * |X(\omega)| + e^{\frac{-T}{\tau}} * P_{t-1}(\omega)$$

The value T is given in the skeletal code as TFRAME and is the frame rate of the system. The time constant, τ, is likely to work well between 20ms and 80ms, so a value of 50ms was chosen as to avoid either extreme and seemed to perform very well experimentally. As a result of implementing this filter, the estimate of the noise by the system performed so much better when alpha was reduced from 25 to 2.5. This method is so effective because the human ear is incredibly sensitive to changes in magnitude. Figure 8 shows the implementation in code.



```
for(k =0 ; k<FFTLEN; k++){
    absX[k] = cabs(X[k]);    /* get absolute value of complex X */

    // compute output of low-pass filter
    P[k] = (1-K_factor)*absX[k] + (K_factor)*P_prev[k];
    P_prev[k]=P[k];
}
```

Figure 8: Enhancement 1

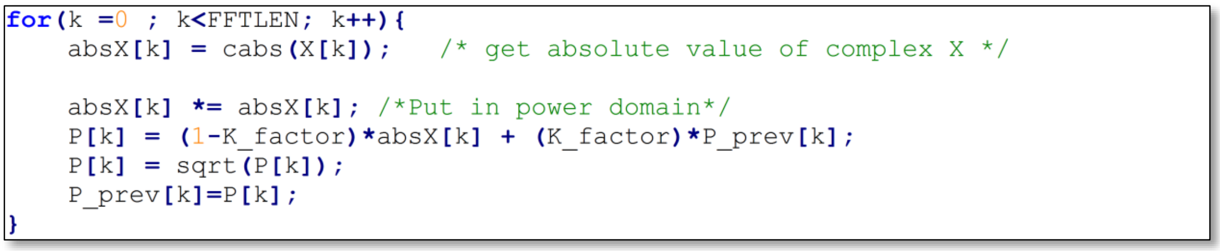
Enhancement 2: Low-Pass Filter in Power Domain

As human hearing is so sensitive to changes in magnitude, it is intuitive that operating the low-pass filter in the power domain should yield similar positive result as the ones from the first enhancement. The mathematics behind this enhancement is almost identical to the previous one, with slight adjustments to convert it into the power domain, as shown below:

$$P_t(\omega) = \sqrt{\left(1 - e^{\frac{-T}{\tau}}\right) * |X(\omega)|^2 + e^{\frac{-T}{\tau}} * P_{t-1}(\omega)}$$

Despite originally thinking this would result in better filtering than enhancement 1, in practice it turned out to not be quite as effective as it was found to be not as robust. This is disappointing because filtering in the power domain would have meant a non-linear processes which should have meant the system would have greater resolution in the relevant sections.

The conversion to the power domain is directly implemented in C code as shown in figure 9:



```
for(k =0 ; k<FFTLEN; k++){
    absX[k] = cabs(X[k]);    /* get absolute value of complex X */

    absX[k] *= absX[k]; /*Put in power domain*/
    P[k] = (1-K_factor)*absX[k] + (K_factor)*P_prev[k];
    P[k] = sqrt(P[k]);
    P_prev[k]=P[k];
}
```

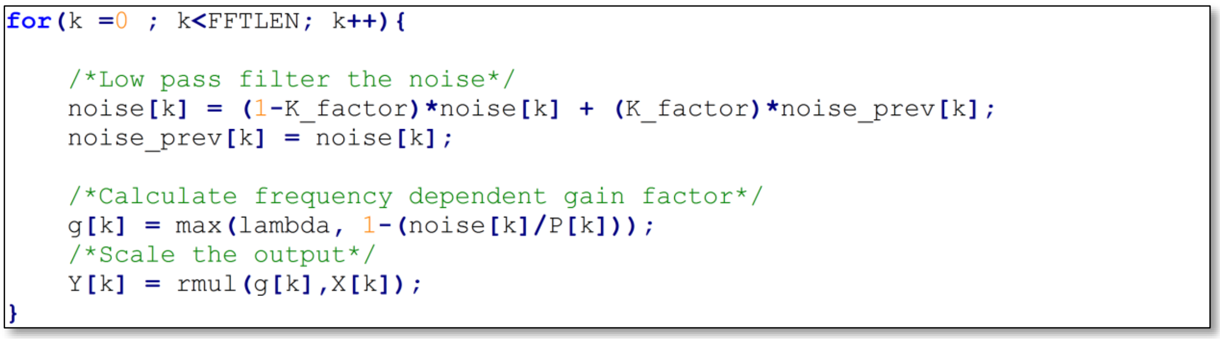
Figure 9: Enhancement 2

Enhancement 3: Low-Pass Filter Noise Estimate

If the noise levels are greatly fluctuating, very noticeable abrupt discontinuities will occur when the buffers are rotated. A solution to this problem is posited by performing a low-pass filter on the noise estimate which has been calculated. This is calculated using a modified version of the equation used in enhancement 1 shown below:

$$M_t(\omega) = \left(1 - e^{\frac{-T}{\tau}}\right) * |N(\omega)| + e^{\frac{-T}{\tau}} * M_{t-1}(\omega)$$

The low-pass filter and scaling are shown in figure 10:



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

    /*Low pass filter the noise*/
    noise[k] = (1-K_factor)*noise[k] + (K_factor)*noise_prev[k];
    noise_prev[k] = noise[k];

    /*Calculate frequency dependent gain factor*/
    g[k] = max(lambda, 1-(noise[k]/P[k]));
    /*Scale the output*/
    Y[k] = rmul(g[k],X[k]);
}
```

Figure 10: Enhancement 3

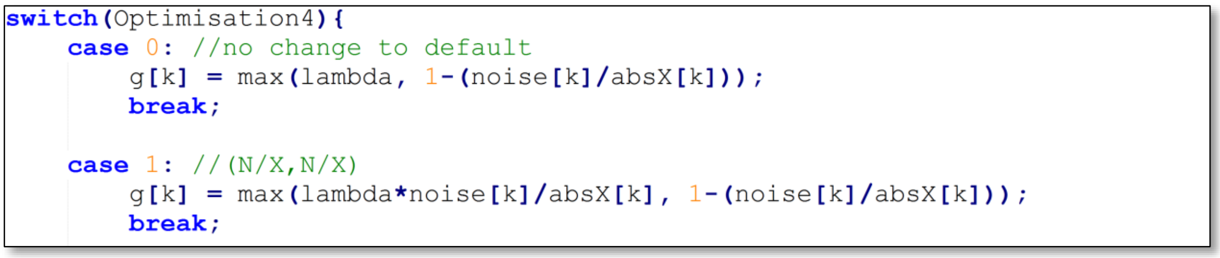
Enhancement 4: Computation of $g(\omega)$ in Magnitude Domain

Enhancement 4 revolves around testing four different methods of calculating the frequency dependent gain factor. The suggested equations to investigate are listed below:

i) $g(\omega) = max\left(\lambda\frac{|N(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$ ii) $g(\omega) = max\left(\lambda\frac{|P(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$

iii) $g(\omega) = max\left(\lambda\frac{|N(\omega)|}{|P(\omega)|}, 1 - \frac{|N(\omega)|}{|P(\omega)|}\right)$ iv) $g(\omega) = max\left(\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|}\right)$

The results of this were that equations iii) and iv) performed the best, with iv) eventually being chosen as the best due to its robust and flexible filtering. It was implemented into the code using case statements (with break) with a small section of the code shown below in figure 11 to illustrate its implementation.



```
switch(Optimisation4){
    case 0: //no change to default
        g[k] = max(lambda, 1-(noise[k]/absX[k]));
        break;

    case 1: //(N/X,N/X)
        g[k] = max(lambda*noise[k]/absX[k], 1-(noise[k]/absX[k]));
        break;
```

Figure 11: Enhancement 4

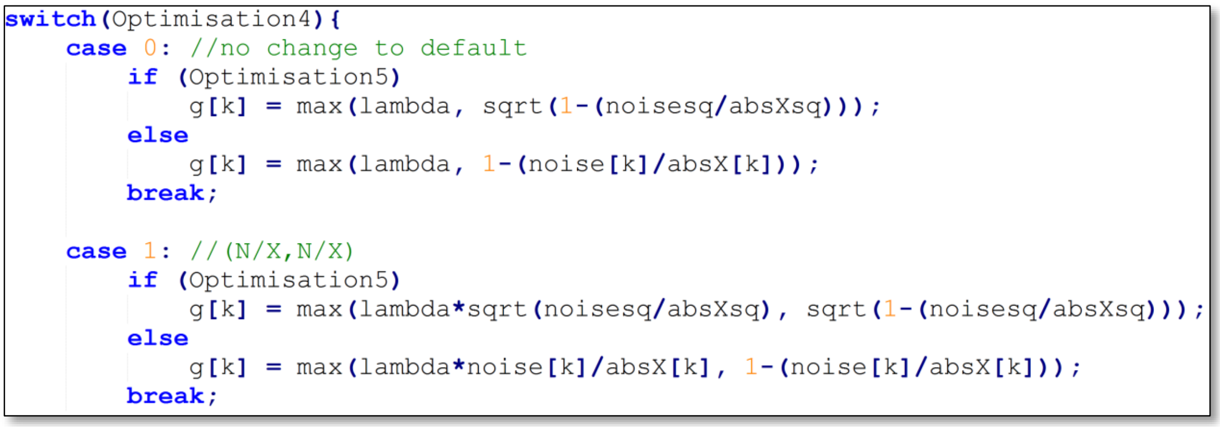
Enhancement 5: Computation of $g(\omega)$ in Power Domain

It was then postulated that the same principle as enhancement 2 could be implemented on enhancement 4 and instead calculate the frequency dependent gain factor in the power domain. This enhancement did not audibly improve the sound quality and actually caused the sound to become very gentle, or attenuated, in places and as a result was discarded.

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

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

The code was adapted to have optimisation 5 within optimisation 4's switch statement, by using if else statements within each case to specify which domain to perform the calculation in. The code is shown in figure 12.



```
switch(Optimisation4){
    case 0: //no change to default
        if (Optimisation5)
            g[k] = max(lambda, sqrt(1-(noisesq/absXsq)));
        else
            g[k] = max(lambda, 1-(noise[k]/absX[k]));
        break;

    case 1: //(N/X,N/X)
        if (Optimisation5)
            g[k] = max(lambda*sqrt(noisesq/absXsq), sqrt(1-(noisesq/absXsq));
        else
            g[k] = max(lambda*noise[k]/absX[k], 1-(noise[k]/absX[k]));
        break;
```

Figure 12: Enhancement 5

Enhancement 6: Overestimation of Noise Level

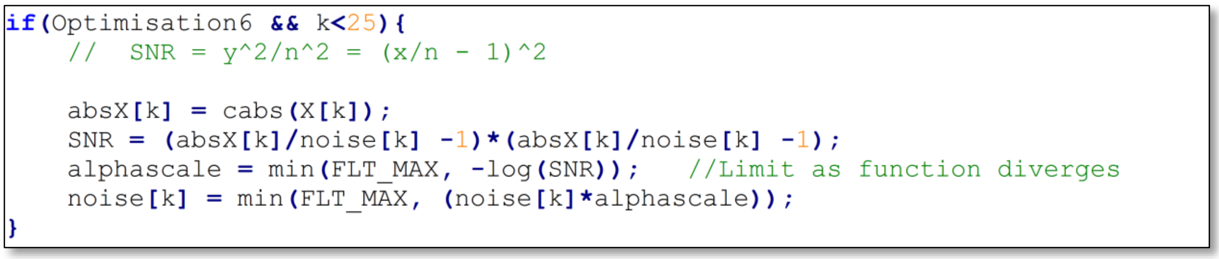
Deliberate overestimation of the noise level is another method of attempting to remove musical noise. The basic principle is to over-subtract the noise estimate to remove spectral peaks by selectively scaling alpha. If alpha was arbitrarily increased it would achieve significant reduction of musical noise but at the expense of attenuating the speech signal. To avoid this, an estimate of the SNR is used to scale alpha. The relation between the two values is inversely proportional as the larger the SNR, the smaller the alpha would be made. The SNR is found using the below:

$$X(\omega) = Y(\omega) + N(\omega)$$

$$\frac{X(\omega)}{N(\omega)} = \frac{Y(\omega)}{N(\omega)} + 1$$

$$\frac{|Y(\omega)|^2}{|N(\omega)|^2} = \left(\frac{X(\omega)}{N(\omega)} - 1\right)^2 = SNR$$

To only address the low frequency bins, the condition is only run for values of k less than 25. It is also important to note that the value of *alphascale* would diverge to infinity if it is not passed through a minimum function with the maximum float value to prevent any overflow errors. Figure 13 shows the code implementation.



```
if(Optimisation6 && k<25){
    //  SNR = y^2/n^2 = (x/n - 1)^2

    absX[k] = cabs(X[k]);
    SNR = (absX[k]/noise[k] -1)*(absX[k]/noise[k] -1);
    alphascale = min(FLT_MAX, -log(SNR));    //Limit as function diverges
    noise[k] = min(FLT_MAX, (noise[k]*alphascale));
}
```

Figure 13: Enhancement 6

Enhancement 7: Adjusting Frame Lengths

One potential enhancement would be to adjust the frame length by changing the variable FFTLEN. Changing the frame length controls the number of frequencies at which the Fourier transform is calculated. Therefore, as having many data points will improve resolution, it could be speculated that this would improve the output.

With a shorter frame size of 128 samples, which is half of the original 256, the resulting output was very poor and choppy. This is due to having to spread the samples further apart from each other, causing an abundance of discontinuities to appear in the magnitude spectrum.

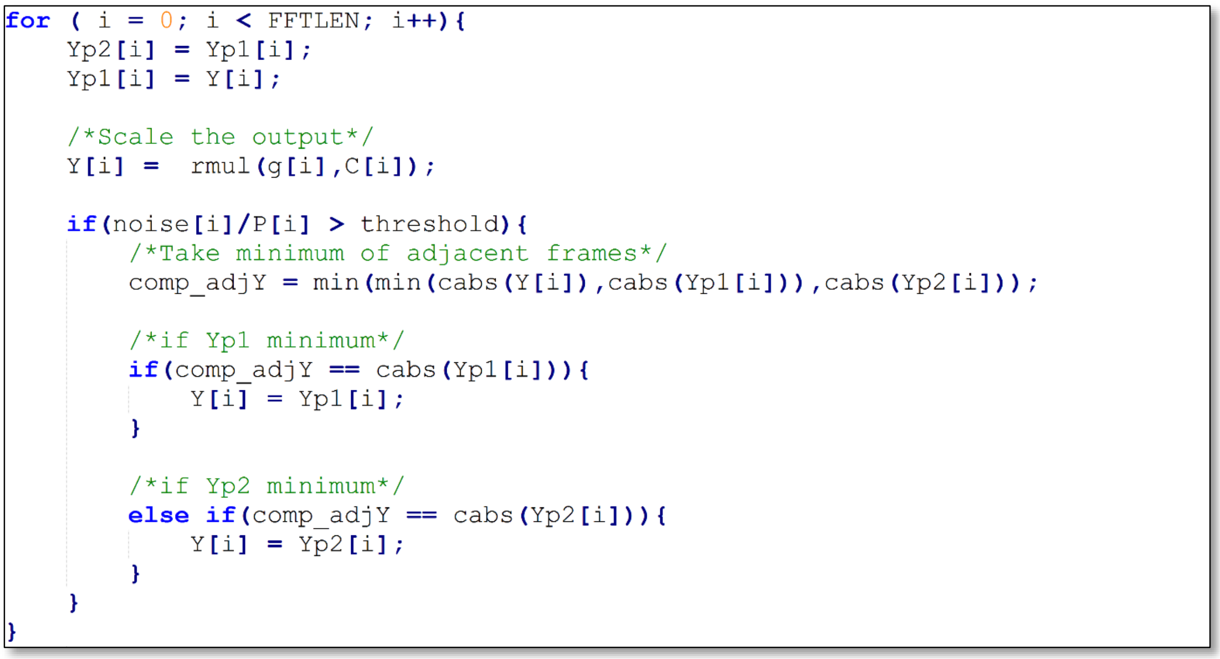
When the frame size was doubled to be 512 samples in length the output was once again suboptimal in comparison to the original frame length. The output was not choppy but the voice of the speaker was slowed down and had become slurred. This unsatisfactory result meant that the frame length was ultimately left unaltered as it was already the best compromise available.

Enhancement 8: Residual Noise Reduction

Musical noise, as mentioned before, is caused by isolated short spectral peaks appearing seemingly at random so it should be possible to suppress these artefacts by taking the minimum of adjacent frames. This minimum should only be taken for spectral components which are not assumed to be part of the speech signal. This is done with the use of a variable named *threshold* which acts as the minimum magnitude to be considered part of the speech signal.

Although theoretically effective at removing musical noise, in practice the speech signal was not found to have been improved and as such, this enhancement was left unused.

The code is shown in figure 14. After the output is scaled, the minimum values of the frames is found, and depending on which one is selected, the values are shifted.



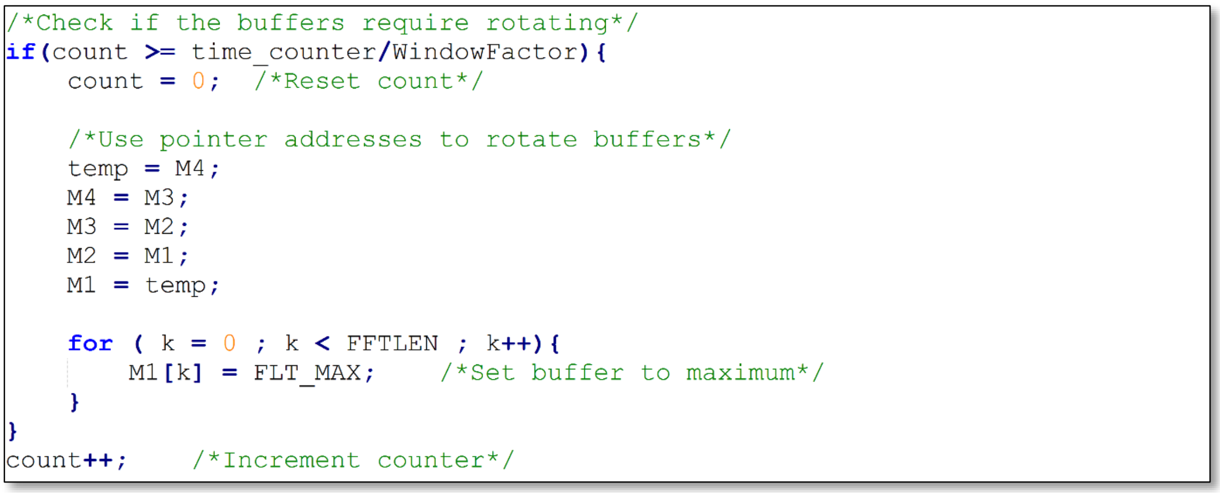
```
for ( i = 0; i < FFTLEN; i++){
    Yp2[i] = Yp1[i];
    Yp1[i] = Y[i];

    /*Scale the output*/
    Y[i] =  rmul(g[i],C[i]);

    if(noise[i]/P[i] > threshold){
        /*Take minimum of adjacent frames*/
        comp_adjY = min(min(cabs(Y[i]),cabs(Yp1[i])),cabs(Yp2[i]));

        /*if Yp1 minimum*/
        if(comp_adjY == cabs(Yp1[i])){
            Y[i] = Yp1[i];
        }

        /*if Yp2 minimum*/
        else if(comp_adjY == cabs(Yp2[i])){
            Y[i] = Yp2[i];
        }
    }
}
```

Figure 14: Enhancement 8

Enhancement 9: Minimum Spectrum Over a Shorter Period

Initially a 10 second window was chosen as it was assumed a speaker would certainly pause during this time frame. In reality, a regular speaker would in fact pause far more frequently than this worst-case scenario. Decreasing the window size results in far more efficient memory usage as well as making the system more robust to sudden changes in noise level, both of which are beneficial to a real-time system.

The best result was found when the *WindowFactor* variable was initialised at 4, meaning the system used windows of 2.5 seconds. However, it should be noted that all the audio samples used in this investigation used the same speech signal, so this parameter has not fully been analysed as speaking patterns vary greatly. As a result, if this system was to actually be implemented, the window size should be increased to 5 seconds to offer some of the sound benefits whilst still retaining a sufficient level of robustness. All of this is shown in figure 15.



```
/*Check if the buffers require rotating*/
if(count >= time_counter/WindowFactor){
    count = 0;  /*Reset count*/

    /*Use pointer addresses to rotate buffers*/
    temp = M4;
    M4 = M3;
    M3 = M2;
    M2 = M1;
    M1 = temp;

    for ( k = 0 ; k < FFTLEN ; k++){
        M1[k] = FLT_MAX;    /*Set buffer to maximum*/
    }
}
count++;    /*Increment counter*/
```

Figure 15: Enhancement 9

Fully Optimised Performance

Although all the enhancements are theoretically good at removing noise from a sample, the practical results turned out to be rather subjective. Some enhancements performed well under certain conditions but fell short of the mark when subjected to a different noise to filter. For a real-time noise filter to have any real application, it must be robust and flexible. Only enhancements which were able to perform well under the whole range of noises were selected. Some of the enhancements produced result by themselves but failed to produce high quality audio when paired with other enhancements so the final selection was a carefully considered compromise.

The final code used enhancements 1 (low-pass filter in magnitude domain), 3 (low-pass filter noise estimate), 4 iv) (computation of $g(\omega)$ in magnitude domain), 6 (overestimation of noise level) and 9 (minimum spectrum over a shorter period), with the rest turn off in the software.

The fully optimised system was a drastic improvement on the non-optimised one. The noise was far quieter and in none of the sample audio files was the speech difficult to understand. However, it was not without fault, as phantom4 still possessed some amount of noise and the occasional attenuation of the speech could be heard in some of the files.

The results of the optimisations can be seen in the following figures.

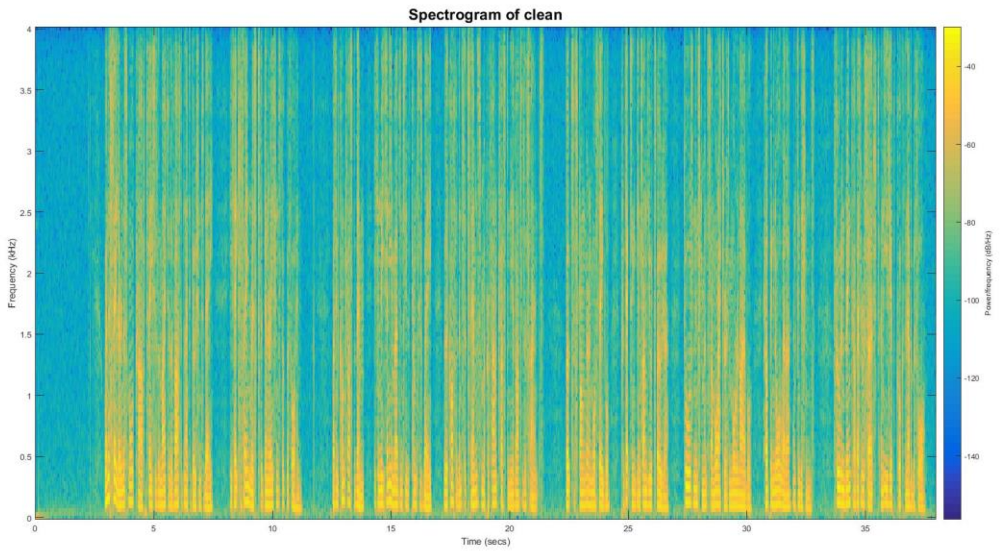


Figure 16: Spectrogram of clean voice

The components of the voice without noise can be seen in figure 16 as the yellow components.

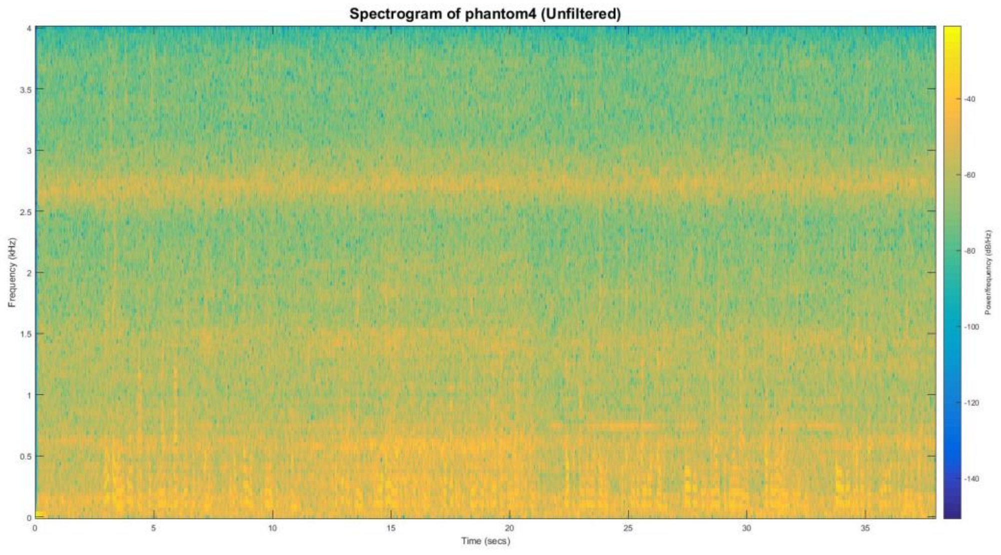


Figure 17: Spectrogram of unfiltered phantom4

Figure 17 shows the spectrogram of phantom4, which has the loudest noise, and the small signal-to-noise (SNR) ratio is evident from the graph.

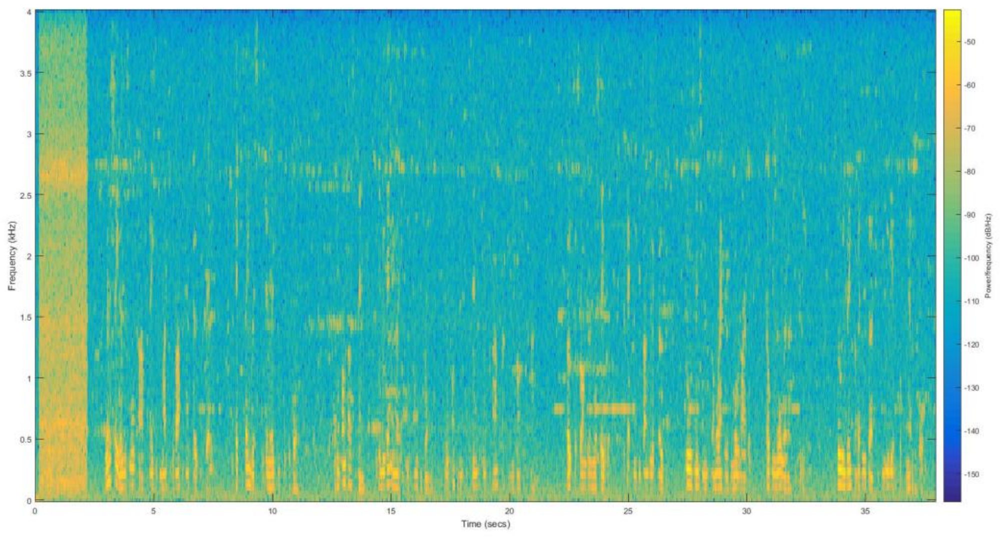


Figure 18: Spectrogram of filtered phantom4

The filtered spectrogram of phantom4 is shown. It retains some of the signal components yet is quite attenuated. Also, it can be seen that in the first 2.5 seconds the signal is not yet filtered.

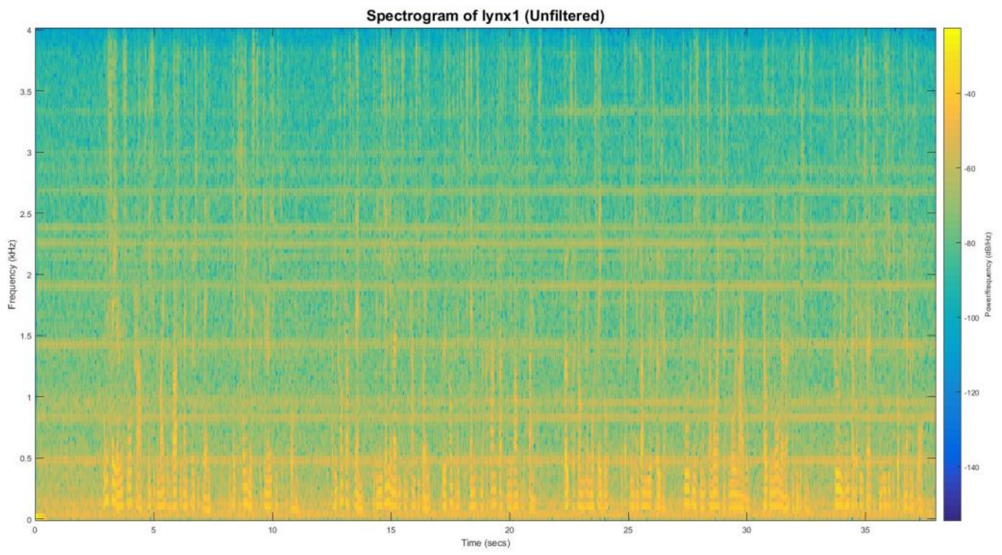


Figure 19: Spectrogram of unfiltered lynx1

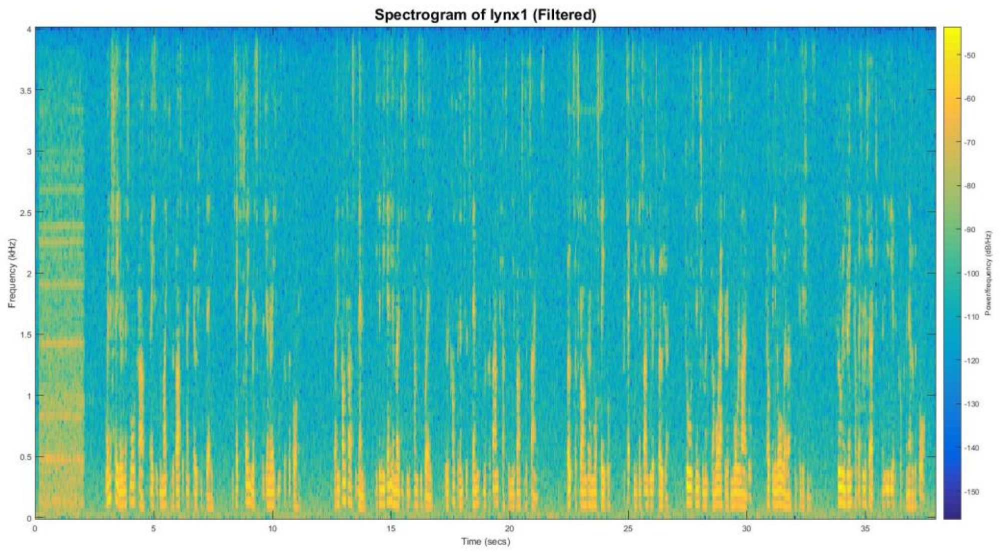


Figure 20: Spectrogram of filtered lynx1

As can be seen from figures 19 and 20, the noise is component is filtered better than in phantom4 due to the greater SNR value of the lynx1 signal.

Conclusion

The basic spectral subtraction technique does remove an audible amount of noise from the samples but does not produce the high quality audio that the optimised system does. This improvement is a result of several trade-offs to prevent the speech signal becoming distorted in any way. This turned out to be a very subjective problem and similar results can be obtained with several different combinations of optimisations used.

Appendix

Full 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>
#include <float.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
#define time_counter 312



/******************************* 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;
int count = 0;           /* Frame pointer */
complex *X;
float *absX;
float *M1;
float *M2;
float *M3;
float *M4;
float *temp;
float *noise;
float *noise_prev;
float *g;
complex *Y;
float K_factor = 0.77;
float P[FFTLEN];
float P_prev[FFTLEN];
float absXsq;
float noisesq;
float Psq;
float SNR=1;
float alphascale=1;
float alpha= 4;
float lambda= 0.012;


/**********************************Switches*********************************/
int Optimisation1=1;
int Optimisation2=0;
int Optimisation3=1;
int Optimisation4=4;
int Optimisation5=0;
int Optimisation6=1;
//int Optimisation7=0;
//int Optimisation8=0;
//int unfiltered =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);
float min (float a, float b);
float max (float a, float b);     /* Frame processing routine */
float lowpass(float x, float prev[], int k);

/********************************** Main routine ************************************/
void main()
{

  int k=0; // 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 */
X = (complex *) calloc(FFTLEN, sizeof(complex));
absX = (float *) calloc(FFTLEN, sizeof(float));
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));
noise_prev = (float *) calloc(FFTLEN, sizeof(float));
g = (float *) calloc(FFTLEN, sizeof(float));
Y = (complex *) calloc(FFTLEN, sizeof(complex));
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];
P[k] = 0;
P_prev[k] =0;
}
  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 **************************/
/* please add your code, at the moment the code simply copies the input to the
ouptut with no processing */


/***************** Applying fft ****************/
for( k = 0; k < FFTLEN; k++){
        X[k] = cmplx(inframe[k],0);} /* copy input straight into output */


fft(FFTLEN,X);
for(k =0 ; k<FFTLEN; k++){
        absX[k] = cabs(X[k]);

        if(Optimisation1){
                P[k] = (1-K_factor)*absX[k] + (K_factor)*P_prev[k];
                P_prev[k]=P[k];
                }

        if(Optimisation2){
                absX[k] *= absX[k];
                P[k] = (1-K_factor)*absX[k] + (K_factor)*P_prev[k];
                P_prev[k]=P[k];
                P[k] = sqrt(P[k]);
                }
        }
```

```
if(count >= time_counter/4){
        count = 0;
        temp = M4;
        M4 = M3;
        M3 = M2;
        M2 = M1;
        M1 = temp;

        for ( k = 0 ; k < FFTLEN ; k++)
        {
                M1[k] = P[k];
                }
        }
count++;

for( k = 0; k < FFTLEN; k++){
        M1[k] = min(M1[k],P[k]);
        noise[k] = alpha*(min(min(M1[k],M2[k]),min(M3[k],M4[k])));



        if(Optimisation6 && k<20){
                absX[k] = cabs(X[k]);
                SNR = (absX[k]/noise[k] -1)*(absX[k]/noise[k] -1);
                alphascale = min(FLT_MAX, -log(SNR));
                noise[k] = min(FLT_MAX, (noise[k]*alphascale));
        }


        if(Optimisation3){
                noise[k] = (1-K_factor)*noise[k] + (K_factor)*noise_prev[k];

                noise_prev[k] = noise[k];
        }



        /***********************      OPTIMISATION 5 goes here      ************************/
        if(Optimisation5){

                absX[k] = min(cabs(X[k]), FLT_MAX);
                absXsq = min(absX[k]*absX[k], FLT_MAX);
                noisesq = min(noise[k]*noise[k], FLT_MAX);
                Psq = min(P[k]*P[k], FLT_MAX);

        }
        /*********************************************************************************/

        switch(Optimisation4){

                //no change to default
                case 0:
                        if (Optimisation5)
                                g[k] = max(lambda, sqrt(1-(noisesq/absXsq)));
                        else
                                g[k] = max(lambda, 1-(noise[k]/absX[k]));

                        break;

                //(N/X,N/X)
                case 1:
                        if (Optimisation5)
                                g[k] = max(lambda*sqrt(noisesq/absXsq), sqrt(1-
(noisesq/absXsq)));
                        else
                                g[k] = max(lambda*noise[k]/absX[k], 1-(noise[k]/absX[k]));

                        break;

                //(P/X,N/X)
                case 2:
                        if (Optimisation5)
                                g[k] = max(lambda*sqrt(Psq/absXsq), sqrt(1-(noisesq/absXsq)));
                        else
                                g[k] = max(lambda*P[k]/absX[k], 1-(noise[k]/absX[k]));

                        break;
                //(N/P,N/P)
                case 3:
                        if (Optimisation5)
                                g[k] = max(lambda*sqrt(noisesq/Psq), sqrt(1-(noisesq/Psq)));
                        else
                                g[k] = max(lambda*noise[k]/P[k], 1-(noise[k]/P[k]));


                        break;

                //(L,N/P)
                case 4:
                        if (Optimisation5)
                                g[k] = max(lambda, sqrt(1-(noisesq/Psq)));
                        else
                                g[k] = max(lambda, 1-(noise[k]/P[k]));

                        break;
        }

g[k] = max(lambda, 1-(noise[k]/P[k]));

Y[k] = rmul(g[k],X[k]);
                }       //end of loop


ifft(FFTLEN,Y);

for( k = 0; k < FFTLEN; k++){
        outframe[k] = Y[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;
}

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

/*Minimum of two floats*/
float min (float a, float b){
        if      ( a < b ){return a;}
        else    {return b;}}
```

```
/*Minimum of two floats*/
float max (float a, float b){
        if      ( a < b ){return b;}
        else    {return a;}}
```