Imperial College London

Department of Electrical and Electronic Engineering

Real-time digital signal processing

Report on "*Project: Speech enhancement*", march 2015

23 pages total

Report written by:

Quentin McGaw,	CID 00746622	Username QDM12
Alexandra Rouhana,	CID 00736752	Username AR4412

1. Project objective

The aim of the project is to design a program for the DSK 6713 board to remove the unknown background noise from an input speech signal. This operation is also called a speech enhancer. The algorithm used in this project is the *spectral subtraction.* The first step is conversion of the input signal to the frequency domain, using a fast Fourier transform (FFT). The processing and filtering of the signal is then realised in the frequency domain. The results are then written back in the time domain with the inverse fast Fourier transform (IFFT) in order to be outputted. The algorithm assumes the input signal spectrum $X(\omega)$ is the sum of the speech spectrum $Y(\omega)$ and the noise spectrum $N(\omega)$.

(Equation 1) $$X(\omega) = Y(\omega) + N(\omega)$$

The following diagram illustrates the procedure adopted by the algorithm.

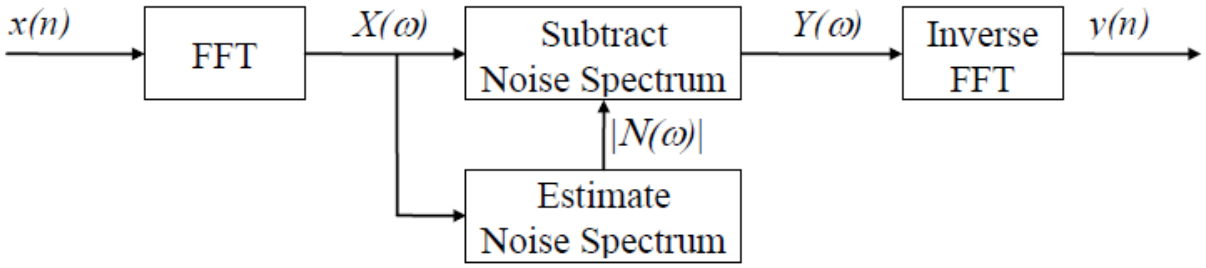


Figure 1: Diagram of the algorithm's procedure for speech enhancement

2. Noise estimation

In order to measure and estimate the noise, we assume the speaker will pause at least once during a period of 10 seconds. This allows the program to measure the noise only at least once each 10 seconds. The estimation of the noise is found by measuring the minimum spectra of the input signal during the 10 seconds.

To reduce the computation needed, the 10 seconds period is split into four 2.5 seconds frames. There are 4 buffers denoted by *M1*, *M2, M3* and *M4* of length 256 (FFTLEN). Each of them corresponds to one of the 2.5 seconds frame in a 10 seconds period. *M1* represents the most recent minimum spectra and is updated continuously. *M2* corresponds to the past value of *M1* (2.5 seconds delay), as does *M3* for *M2* and *M4* for *M3*. Now, *M1* is updated for each frequency bins according to the following equation:

(Equation 2) $$M1(k) = \min(M1(k), X(k))$$

This basically means that for a certain frequency bins, M1 is changed only if X is lower than its previous value. The noise estimation, `magnitude.N[k]` in the code, is then found by

multiplying the minimum of the 4 buffers with the subtraction factor α, `param.alpha` in the code. These requirements are implemented as described below.

To implement the shifting of these 4 buffers efficiently, they are created with pointers and allocated with *calloc()*.

At a global scope:

```
float *M1, *M2, *M3, *M4, *Mx;
/* 4 pointers to noise estimation buffers (initiated to 0s in the main).
Mx is a temporary pointer used to shift the 4 buffers (see later).*/
```

In the main():

```
M1 = (float *) calloc(FFTLEN, sizeof(float));
M2 = (float *) calloc(FFTLEN, sizeof(float));
M3 = (float *) calloc(FFTLEN, sizeof(float));
M4 = (float *) calloc(FFTLEN, sizeof(float));
```

Equation 2 is implemented with the following code in the *for loop* of *process_frame().*

```
//changes M1[k] to the measured magnitude X[k] if and only if it is smaller.
M1[k] = min2F(M1[k],magnitude.X[k]);
//updates the noise magnitude N.
magnitude.N[k] = param.alpha * min4F(M1[k],M2[k],M3[k],M4[k]);
```

The second line of code shows how the noise estimate is calculated as described before. Note that the function *min2F* returns the smallest float variable amongst its 2 arguments. Similarly, *min4F* does the same thing but with 4 float arguments (see full code).

To implement the shifting of the noise estimation buffers, the following code is used, *after the for loop of the process_frame() function.*

```
if(++frame_n > LIM) //When we reach 2.5 seconds (frame_n=78)
{
    frame_n = 0; //reinitialises frame_n
    //shift buffers
    Mx = M4;
    M4 = M3;
    M3 = M2;
    M2 = M1;
    M1 = Mx;
    for(k = 0; k < FFTLEN/2; k++)
    {
        M1[k] = magnitude.X[k];
        //updates M1 with the values of the input signal spectrum
    }
}
```

The variable *frame_n* corresponds to 256 (FFTLEN) samples processed (*for loop*). In time, this corresponds to 256 * $\frac{1}{8000\ Hz}$ = 0.032 second. 2.5 seconds would then be attained when frame_n = $\frac{2.5\ seconds}{0.032\ second} = 78$ so the limit *LIM* is set with the following pre-processor statement:

```
#define LIM (2.5*(FSAMP/FFTLEN))
/*Used to shift the buffers Mi each 2.5 seconds.
```

```
We have this relationship LIM * (1/FSAMP) * FFTLEN = 2.5
so LIM = 2.5 * FSAMP/FFTLEN = 2.5*8000/256 = 78.125 = 78 */
```

When varying the parameters alpha (noise overestimation factor) and lambda (floor parameter), the following results can be observed. As we increase alpha, the underestimation of the noise is corrected, most of the noise is removed. However if alpha is too large, the gain will often be equal to the floor parameter lambda. The speech will sound distant and distorted. The background noise will therefore be increased instead of being removed. If the value of lambda increases, the musical noise decreases. However, if lambda is too large, the white noise will increase. A good compromise was initially found by setting alpha=4.00 and lambda=0.02 (even for final configuration).

Please refer to the full code for the optimal configuration used which is set by default.

3. Noise subtraction

From the equation 1, we obtain the following transfer function $g(\omega)$.

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

However, the phase of the noise signal is unknown so the following expression is used:

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

Finally, as $|N(\omega)|$ may be larger than $|X(\omega)|$ sometimes, this can make $g(\omega)$ negative. To avoid this, the value of $g(\omega)$ is given a floor parameter with the equation 4 below.

(Equation 4) $$g(\omega) = \max\left(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right),$$

where λ, called `param.lambda` in the code, is the positive floor parameter of $g(\omega)$, generally between 0.01 and 0.10. Its influence will be discussed later on.

The calculation of the real gain $g(\omega)$ and the calculation of the output complex signal $Y(\omega)$ are done in the following line of code:

```
signal.Y[k] = rmul(max2F(g_var1,g_var2), signal.X[k]); //calculates signal.Y (complex)
```

The function *rmul(float, complex)* performs a real multiplication on a complex number and the function *max2F(float, float)* returns the largest of its two arguments. The global float

variables *g_var1* and *g_var2* are, in the base case, assigned with the following code:

```
g_var1 = param.lambda;
g_calc = magnitude.N[k] / magnitude.X[k];
g_var2 = 1 - g_calc;
```

These global variables allow to easily change the two expressions in the calculation of the gain for the various enhancements which will be discussed later on.

At the end of the process_frame() function, the symmetry of the FFT is used to save some computation as shown below. The signal is then IFFTed and the result is written to the output frame as described below.

```
    for(k = FFTLEN/2 + 1; k < FFTLEN; k++) //Symmetry of FFT with conjugates
    {
        signal.Y[k] = conjg(signal.Y[FFTLEN-k]); //conjugate
    }
    ifft(FFTLEN, signal.Y); //performs the IFFT on the signal Y (complex)

    for(k = 0; k < FFTLEN; k++)
    {
        outframe[k] = real(signal.Y [k]); //writes the results to output frame

    }
}
```

4. Enhancements

4.1 Logical switches for runtime

The program has been designed with several *if*, *else if*, and *else* blocks in order to be able to change the desired enhancement(s) at runtime. Each of the following enhancements are activated with 1 or deactivated with 0 using Booleans. The configuration describing which enhancements are on or off is in `g_enhancement[MAX_ENHANCEMENTS]`. This global array of Booleans is defined with the following:

```
bool g_enhancement[MAX_ENHANCEMENTS] = { //set to best initially
    0,  //g_enhancement[0]: Low pass filter of magnitude.X
    1,  //g_enhancement[1]: Low pass filter of magnitude.X in the Power domain
    0,  //g_enhancement[2]: Low pass filter of magnitude.N
    0,  //g_enhancement[3]: Dynamic gain & floor parameter (3)
    0,  //g_enhancement[4]: Dynamic gain & floor parameter (4)
    1,  //g_enhancement[5]: Dynamic gain & floor parameter (5)
    0,  //g_enhancement[6]: Dynamic gain & floor parameter (6)
    0,  //g_enhancement[7]: Gain calculated in the power domain
    1,  //g_enhancement[8]: Overestimate noise at low frequencies/bandpass
    1,  //g_enhancement[9]: Dynamic noise overestimation with SNR
    0}; //g_enhancement[10]: Residual noise reduction
```

There is also another array similar to this one, called `g_alt_enhancement[MAX_ENHANCEMENTS]`, which is basically an alternative configuration which can be applied by changing the value of the global Boolean defined by:

```
bool g_switchIt = 0;
//Allow to switch between a configuration of enhancements and another
```

The g_switchIt variable is checked each time the program runs the process_frame() function. If it has been changed to 1 through the Watch window of CCS studio, it will be restored to 0 and the function `switch_configuration(void)` is executed. This function basically swaps around the two enhancements configuration (see full code in appendix 1), hence allowing instant comparisons between them.

4.2 Pre-processor statements for compilation

In addition to the logical switches for runtime, the parameter OPT_BEST_CONFIG defined below allows if set to 1 to remove several non-optimal parts of the code at compilation in order to obtain the best performance possible.

```
#define OPT_BEST_CONFIG 0
/*If 1, parts are skipped at compilation to only stick to the best configuration of
g_enhancements. On the other hand, 0 will allow to choose any configuration as well as
live changing them in the Watch window (during runtime). */
```

Please refer to the full code to see the impact of the pre-processor statements on the performance of the code.

4.3 First enhancement: Low pass filter of $|X(\omega)|$

This first enhancement consists in low pass filtering (or moving average) the magnitude of $X(\omega)$. This low pass filter is implemented in the following function.

```
float lpf(float present, float past, float pole)
{
    return present + pole*(past - present);
} //low pass filter (to smooth transition also)
```

The argument *present* is the present value, *past* is the previous value and *pole* is the z-plane pole. In this enhancement, the pole is defined by

```
#define POLE exp(-(float)TFRAME / ((float)MS/(float)1000.0))
//POLE is used for moving averages for magnitudes X and N.
```

Where the parameters are defined as follow:

```
#define TFRAME FRAMEINC/FSAMP       /* time between calculation of each frame = 8 ms*/
#define MS 40 //40 milliseconds (between 20 and 80 milliseconds)
```

A global variable *K* is then defined with

```
//K is set to the macro POLE. It can be changed during runtime in the Watch window.
float K = POLE;
```

in order to change K during runtime with the Watch window.

Now the low pass filter of X is realised in the process_frame() function with

```
        else if(g_enhancement[0]) //updates the magnitude X and stores it in Xp

        {
            magnitude.X[k] = lpf(magnitude.X[k], magnitude.Xp[k], K);
            magnitude.Xp[k] = magnitude.X[k];
        }
```

Here magnitude.Xp is the previous value of magnitude.X.

Now with this enhancement, the noise signal was estimated more precisely and was hence reduced. This allowed the *alpha* coefficient (*param.alpha*) to be reduced from 20 to 4.

4.4 Second enhancement: Low pass filter of | $X(\omega)$ | in the power domain

This second enhancement does exactly the same thing as with the first enhancement, except that the filtering is realised in the power domain. This idea makes sense as humans hear sound in the power domain. In addition, noise estimations will be more reactive to amplitude variations, hence giving a better noise estimation.

This enhancement is implemented with the following block of code.

```
if(g_enhancement[1]) //updates the magnitude X and stores it in Xp

{
    magnitude.X[k] = sqrt(lpf(sq(magnitude.X[k]), sq(magnitude.Xp[k]), K));
    magnitude.Xp[k] = magnitude.X[k];
}
```

This basically takes the square root of the low pass filtered version of the squared magnitude X. Again, Xp is the previous value of X. This enhancement gave a slightly better output quality but the parameters alpha and lambda were not changed.

4.5 Third enhancement: Low pass filter of | $N(\omega)$ |

This enhancement performs again a moving average operation but this time on the noise estimation. After that the noise estimate `magnitude.N[k]` has been calculated, the following code is executed:

```
#if !OPT_BEST_CONFIG
        if(g_enhancement[2])
        {
            //low pass filter on noise magnitude (using N and Np)
            magnitude.N[k] = lpf(magnitude.N[k], magnitude.Np[k], K);
            //Np is the previous value of the noise magnitude N.
            magnitude.Np[k] = magnitude.N[k];
        }
#endif
```

This removes most of the discontinuities caused by random noise, hence lowering the possibility of a noise spike disrupting the speech signal. However, because each of the test signals contains more or less a constant and similar background noise, this enhancement did not make any audible difference.

4.6 Fourth enhancement: Dynamic gain and floor parameter

This enhancement aims to adjust dynamically the floor parameter λ in the equation 4. As this constant floor parameter is cutting out any signal when the noise estimate is close (or larger) than the input signal magnitude, one solution is to make it dynamic. This would

make the program loose less information in some cases. The following code was used to implement this first dynamic floor parameter.

```
#if !OPT_BEST_CONFIG
        else if(g_enhancement[3])
        {
            g_var1 *= g_calc; // lambda * mag.N[k] / mag.X[k]
        }
```

Where, as described before, the variables are:

```
        g_var1 = param.lambda;
        g_calc = magnitude.N[k] / magnitude.X[k];
```

So we have the floor parameter λ replaced by $\lambda \frac{|N(\omega)|}{|X(\omega)|}$. This means that the floor parameter will be lowered for small noise estimates, and larger if the noise estimate becomes more important.

Three other designs of dynamic floor parameter and gain were implemented in the code, which are associated with the enhancements 4, 5 and 6. The following code block shows the implementation of all the four dynamic floor parameter/gain designs.

```
        //g_var1, g_var2 & g_calc set by default for easier calculations

        g_var1 = param.lambda;
        g_calc = magnitude.N[k] / magnitude.X[k];
        g_var2 = 1 - g_calc;
        if(g_enhancement[5])
        {
            g_calc = magnitude.N[k] / magnitude.Xp[k];
            g_var1 *= g_calc; //lambda * mag.N[k] / mag.Xp[k]
            g_var2 = 1 - g_calc; // 1 - mag.N[k] / mag.Xp[k]
        }
#if !OPT_BEST_CONFIG
        else if(g_enhancement[3])
        {
            g_var1 *= g_calc; // lambda * mag.N[k] / mag.X[k]
        }
        else if(g_enhancement[4])
        {
            // lambda * mag.Xp[k] / mag.X[k]
            g_var1 *= (magnitude.Xp[k]/magnitude.X[k]);
        }
        else if(g_enhancement[6])
        {
            // 1 - mag.N[k] / mag.Xp[k]
            g_var2 = 1 - magnitude.N[k]/magnitude.Xp[k];
        }
#endif
        //Y(w) = max(g_var1, g_var2) * X(w)
        signal.Y[k] = rmul(max2F(g_var1,g_var2), signal.X[k]);
```

The enhancements 4, 5 and 6 used the moving average signals Xp and Np in order to see whether a smoother input signal would generate a better floor parameter than the one produced by a rapidly changing input signal. The result is that the enhancement 5 was found to bring the best audible result. The floor parameter was low when the speech signal was present whilst higher when the speaker was taking a break.

4.7 Fifth enhancement: Gain calculated in the power domain

This enhancement, associated with the enhancement 7 in the code, calculates the gain (before the floor parameter comparison) in the power domain. It basically assigns the gain to $\sqrt{1 - \left(\frac{|N(\omega)|}{|X(\omega)|}\right)^2}$ which is implemented as follow.

```
else if(g_enhancement[7])
{
    // sqrt(1 - (mag.N[k]*mag.N[k]) / (mag.X[k]*mag.X[k]))
    g_var2 = sqrt(1 - (sq(g_calc)));
}
```

This enhancement gave a large speech echo and was thus not retained further.

4.8 Sixth enhancement: Overestimation of noise for small frequency bins

To overestimate the noise for small frequency bins, 256 coefficients are used to attenuate the noise estimate in its calculation. The calculation of the noise estimate then becomes in the C code:

```
magnitude.N[k] = param.alpha * alpha_coef[k] * min4F(M1[k],M2[k],M3[k],M4[k]);
```

The following Matlab script is used to generate the attenuation coefficients for small frequency bins.

```
%% MATLAB SCRIPT TO GENERATE FFTLEN ATTENUATION

%% Attenuates small frequency bins
%% COEFFICIENTS FOR ALPHA (USED FOR NOISE ESTIM.)
%% AUTHORS: Quentin McGaw and Alexandra Rouhana
%% DATE: MARCH 2015
clc;
% ######## PARAMETERS ########
FFTLEN = 256; %number of coefficients
att_LF = 1.5; %Low frequency attenuation
freq_LP = 10; %Low pass frequency bin
% ######## CALCULATIONS ########
coef = ones(FFTLEN,1); %all coef to 1
% ### Low pass ###
for i=0:(freq_LP-1)
    coef(i+1) = (freq_LP*att_LF - (att_LF-1)*i)/freq_LP;
    coef(FFTLEN-i) = coef(i+1); %symmetry
end
% ######## RESULTS ########
% ### Plot ###
plot(1:FFTLEN,coef)
axis tight
title('Alpha exaggeration coefficient')
xlabel('Frequency bin')
ylabel('Amplification factor')
% ### Writing ###
f=fopen('attenuation1.txt','w'); %open file
fprintf(f,'float alpha_coef[] = {');
fprintf(f,'%1.16f,',coef(1:FFTLEN-1));
%16f to avoid rounding consequences
fprintf(f,'%1.16f};\n',coef(FFTLEN));
```

```
fclose(f); %close the text file
```

The text file *attenuation1.txt* is then included in the C code with

```
#include "attenuation1.txt" //Includes noise attenuation coefficients
```

This enhancement removed slightly more noise from the signal.

4.8 Seventh enhancement: Decreasing the FFT length for the overlap add algorithm

Decreasing the length of the FFT to 128 lowers the amplitude of the speech signal and more musical noise is introduced. This is because one frequency bin represents a range of $\frac{FSAMP}{FFTLEN}$. So for a FFT length of 256, each frequency bin represents a range of $\frac{8000}{256} = 31.25\ Hertz$. Now for FFTLEN=128, each frequency bin represents a range of 62.5 Hertz, hence reducing the frequency resolution logarithmically. Increasing the FFT length to 512 produced a lot of echo and added musical noise. The FFTLEN was thus kept to 256 which appear to give the best compromise available.

4.9 Eighth enhancement: Residual noise reduction

After removing the estimation of the noise from the input signal, the noise remaining can be further reduced by replacing the output by the minimum value of the previous three frames. Indeed, since the frequency of the noise is higher than the frequency of the signal, we consider that the variation of the output is mainly due to the noise signal which is additive. The adjacent frame with minimum output must therefore be the one with the lower noise. This algorithm is applied if noise subtraction is not efficient enough and the noise magnitude to signal magnitude ratio is superior to a threshold called 'residue'.

In order to determine whether or not the output should take the result of the adjacent frame, a delay of one frame is added to the for loop of the algorithm. This design is implemented with the following code.

```
#if !OPT_BEST_CONFIG
        //Calculates signal.Y[k] in function of the values of signal.X at k, k-1 & k-2
        if(g_enhancement[10]) //Residual noise reduction
        {
            //updates oldstate for the if condition below.
            oldstate = state;
            //updates the state for next iteration.
            state = ((magnitude.N[k]/magnitude.X[k]) > param.residue);
            /*If noise magnitude was previously >= half the input signal magnitude,
              the output signal is set to the minimum of the three adjacent frames.*/
            if(oldstate)
            {
                signal.Y[k] = min3C(signal.Y[k],signal.Y[k-1],signal.Y[k-2]);
            }
            else
            {
                signal.Y[k] = signal.Y[k-1];
            }
        }
#endif
```

5. Additional enhancements

5.1 Ninth enhancement: Dynamic noise overestimation with SNR (Using reference 1)

Another enhancement was to make the noise overestimation factor alpha dynamic. In order to do, the signal to noise ratio (SNR), which is SNR.value in the code, is calculated in each frame. The overestimation factor, param.alpha, is then changed according to the value of the SNR (in decibels). The following code implements this design.

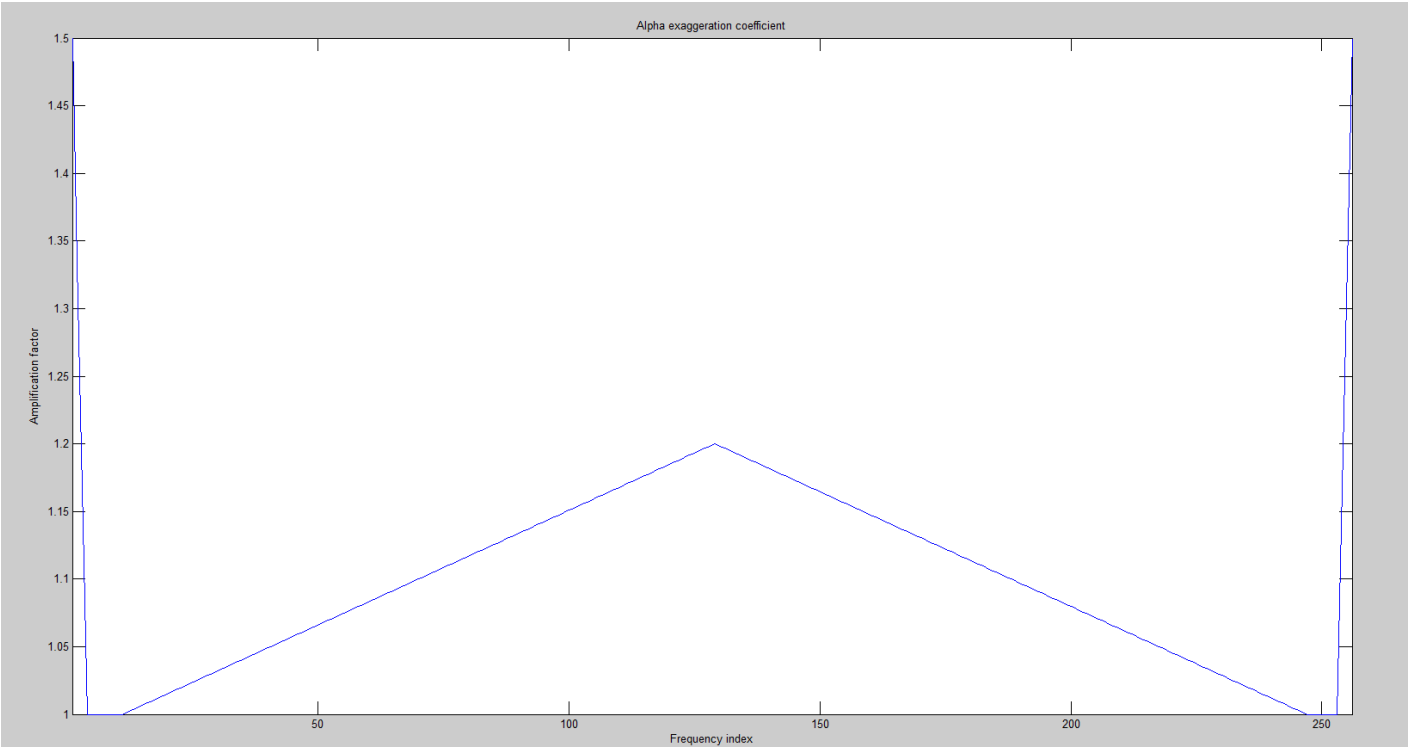
```
if(g_enhancement[9]) //SNR-dependent alpha value
{
    SNR.value = 10 * log10(SNR.SPower/SNR.NPower); // in decibels
    if(SNR.value < SNR.lower) //SNR < -5
    {
        param.alpha = SNR.alpha_high; //alpha = 4.5
    }
    else if(SNR.value > SNR.upper) //SNR > 20
    {
        param.alpha = SNR.alpha_low; //alpha = 1
    }
    else //line equation when -5 < SNR < 20
    {
        //Straight line equation
        float b = (SNR.alpha_low - SNR.alpha_high)/(SNR.upper - SNR.lower);
        float a = SNR.alpha_high - b * SNR.lower;
        param.alpha = a + b*SNR.value;
    }
}
```

If the SNR is below its lower threshold (SNR.lower), then alpha is assigned a value of 4.5 (determined by SNR.alpha_high). This is similar when the SNR exceeds its upper threshold. Finally, if the SNR is between these two thresholds, alpha is assigned a value obtained from the straight line equation with the two thresholds on it.

This means that when the noise power increases, alpha will dynamically increases. And when the SNR increase, alpha will then decrease as it is not needed in essence.

5.2 Enhancement: Overestimation of noise out of human frequency bins

As the human can only hear sound within the frequency range 80 Hz – 260 Hz, a last enhancement was to modify the attenuation coefficients (alpha_coef). This involved attenuating frequency bins not only for low ones but also for high ones. This was done with MatLab script in Appendix 2. The resulting coefficients are shown in the figure below.



This improved again slightly the quality of the filtered output.

6. Conclusion

This project showed us a few possibilities and limitations of background noise removal techniques for an unknown signal. Several enhancements were tested and the best ones were having dynamic behaviours. This project also taught us how frequency operations such as FFTs or filters were useful in background noise removal.

REFERENCES

[1] Berouti,M. Schwartz,R. & Makhoul,J.,"*Enhancement of Speech Corrupted by Acoustic Noise*", Proc ICASSP, pp208-211, 1979.

APPENDICES

Appendix 1: Full *enhance.c* code

enhance.c

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

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

                                PROJECT: Frame Processing

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

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

 *************************************************************************
                              By Danny Harvey: 21 July 2006
                              Updated for use on CCS v4 Sept 2010
 *************************************************************************

                     Modified by Alexandra Rouhana and Quentin McGaw

 ***************************** Pre-processor statements ***************************/
//  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 = 129*/
#define OVERSAMP 4                  /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP)  /* Frame increment = 64*/
#define CIRCBUF (FFTLEN+FRAMEINC)   /* length of I/O buffers = 320*/

#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 = 8 ms*/

/* ###################### Code added : Preprocessor statements ###################### */
#include "attenuation1.txt" //Includes noise attenuation coefficients
#define MS 40 //40 milliseconds (between 20 and 80 milliseconds)
#define POLE exp(-(float)TFRAME / ((float)MS/(float)1000.0))
//POLE is used for moving averages for magnitudes X and N.
#define LIM (2.5*(FSAMP/FFTLEN))
/*Used to shift the buffers Mi each 2.5 seconds.
We have this relationship LIM * (1/FSAMP) * FFTLEN = 2.5
so LIM = 2.5 * FSAMP/FFTLEN = 2.5*8000/256 = 78.125 = 78  */
#define MAX_ENHANCEMENTS 11
//Maximum number of g_enhancements defined in the code.
#define OPT_BEST_CONFIG 1
/*If 1, parts are skipped at compilation to only stick to the best configuration
of g_enhancements. On the other hand, 0 will allow to choose any configuration
as well as live changing them in the Watch window (during runtime). */
typedef short bool; //Defines the boolean type for clearer code.
/* ############################## END OF ADDED CODE ############################## */


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


/* ###################### Code added : Global variables ###################### */
// ========= Enhancements configurations =========
bool g_enhancement[MAX_ENHANCEMENTS] = { //set to best initially
    0,  //g_enhancement[0]: Low pass filter of magnitude.X
```

```
    1,  //g_enhancement[1]: Low pass filter of magnitude.X in the Power domain
    0,  //g_enhancement[2]: Low pass filter of magnitude.N
    0,  //g_enhancement[3]: Dynamic gain & floor parameter (3)
    0,  //g_enhancement[4]: Dynamic gain & floor parameter (4)
    1,  //g_enhancement[5]: Dynamic gain & floor parameter (5)
    0,  //g_enhancement[6]: Dynamic gain & floor parameter (6)
    0,  //g_enhancement[7]: Gain calculated in the power domain
    1,  //g_enhancement[8]: Overestimate noise at low frequencies/bandpass
    1,  //g_enhancement[9]: Dynamic noise overestimation with SNR
    0}; //g_enhancement[10]: Residual noise reduction
#if !OPT_BEST_CONFIG
bool g_switchIt = 0;
//Allow to switch between a configuration of enhancements and another
bool g_alt_enhancement[MAX_ENHANCEMENTS] = {
    0,  //g_alt_enhancement[0]: Low pass filter of magnitude.X
    1,  //g_alt_enhancement[1]: Low pass filter of magnitude.X in the Power domain
    0,  //g_alt_enhancement[2]: Low pass filter of magnitude.N
    0,  //g_alt_enhancement[3]: Dynamic gain & floor parameter (3)
    0,  //g_alt_enhancement[4]: Dynamic gain & floor parameter (4)
    1,  //g_alt_enhancement[5]: Dynamic gain & floor parameter (5)
    0,  //g_alt_enhancement[6]: Dynamic gain & floor parameter (6)
    0,  //g_alt_enhancement[7]: Gain calculated in the power domain
    0,  //g_alt_enhancement[8]: Overestimate noise at low frequencies
    0,  //g_alt_enhancement[9]: Dynamic noise overestimation with SNR
    0}; //g_alt_enhancement[10]: Residual noise reduction
#endif

// ========= Structure declarations & initialisation =========
struct ComplexSignalType {
    complex X[FFTLEN]; //complex value of X(w).
    complex Y[FFTLEN]; //complex value of output Y(w)
} signal;

struct MagnitudeType {
    float X[FFTLEN]; //MODIF: Magnitude of X(w)
    float Xp[FFTLEN]; //MODIF: Past value of X (magnitude)
    float N[FFTLEN]; //MODIF: Magnitude of noise estimation
    float Np[FFTLEN]; //for enhancement[2]
} magnitude = {{0}, {0}, {0}, {0}};

struct ParametersType {
    float alpha; //Initialised to 4, for noise overestimation.
    float lambda; //Initialised to 0.02, for noise overestimation
    float residue; //Initialised to 0.5, for VAD g_enhancement (9).
} param = {4, 0.02, 0.5};


//This structure (SNR variable) corresponds to g_enhancement[9] with SNR dependent
alpha.
struct SNRType {
    float SPower; //Power of the signal, initialised to 0.
    float NPower; //Power of the noise, initialised to 0.
    float value; //Value of the SNR in decibels, initialised to 1.
    float lower; //Lower SNR bound, initiated to -5 (db)
    float upper; //Upper SNR bound, initiated to 20 (db)
    /* Lower value of alpha, corresponding  to when SNR > upper bound (20 db)
    Initiated to 1.0 */
    float alpha_low;
    /* Higher value of alpha, corresponding  to when SNR < lower bound (-5 db)
    Initiated to 4.5 */
    float alpha_high;
} SNR = {0, 0, 1, -5, 20, 1.0, 4.5};

// ========= Other global variables =========
//K is set to the macro POLE. It can be changed during runtime in the Watch window.
float K;
/* 4 pointers to noise estimation buffers (initiated to 0s in the main).
```

```
   Mx is a temporary pointer used to shift the 4 buffers later on.*/
float *M1, *M2, *M3, *M4, *Mx;
//Float variables used in several cases for calculations (no specific use).
float g_var1, g_var2, g_calc;
#if !OPT_BEST_CONFIG
/*For enhancement 10 only. Both states are intiated to false.
oldstate contains the previous value of state and state*/
bool state = 0, oldstate = 0;
#endif
/*Frame count, which represents FFTLEN=256 samples.
In time, this represents 256 / 8000 = 32 milliseconds.
frame_n is compared to LIM (78) in order to shift the noise estimation
buffers Mi every 2.5 seconds. */
int frame_n = 0;
/* ############################## END OF ADDED CODE ############################## */


 /****************************** 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 */
/* ################ Code added : Function prototype declarations ################ */
float sq(float x); //Returns the square of the float argument x.
float mag(complex z); //Returns the magnitude of the complex argument z.
//The mag function is 40 to 50 instruction cycles faster than cabs().
float lpf(float present, float past, float pole); //performs the LP filtering (moving
avge)
#if !OPT_BEST_CONFIG //This function is useless for optimal configuration.
void switch_configuration(void); //Switches between g_enhancement and
g_alt_enhancement.
#endif
//The function max2F returns the maximum for 2 float arguments
float max2F(float x, float y);
/* The functions min2F, min4F and min3C return the minimum respectively for
   2 float arguments, 4 float arguments and 3 complex arguments (magnitude). */
float min2F(float x, float y);
float min4F(float a, float b, float c, float d);
complex min3C(complex x, complex y, complex z); //Complex with the smallest magnitude.
/* ############################## END OF ADDED CODE ############################## */



/******************************** 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 */
/** ################### Code added : Variables allocation ################### */
    M1          = (float *) calloc(FFTLEN, sizeof(float));
    M2          = (float *) calloc(FFTLEN, sizeof(float));
    M3          = (float *) calloc(FFTLEN, sizeof(float));
    M4          = (float *) calloc(FFTLEN, sizeof(float));
    K = POLE; //Set K to the POLE constant (defined at beginning of code).
    //K is a global variable so that we can modify at runtime with the Watch window.
/** @@@@@@@@@@@@@@@@@@@@@@@@@@@@ END OF ADDED CODE @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */

    /* 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];
/** ################### Code added : Variables initialisation ################## */
        /* Clears all the arrays (M1, M2, M3, M4, magnitude.Xp, magnitude.Np. */
        M1[k] = 0;
        M2[k] = 0;
        M3[k] = 0;
        M4[k] = 0;
        magnitude.Xp[k] = 0; //For enhancements 0, 1, 4, 5, 6.
        //Used for low pass filtering using past value of the magnitude of X(w)
#if !OPT_BEST_CONFIG
        magnitude.Np[k] = 0; //For enhancement 2 only.
        if(!g_enhancement[8]) //For overestimation of noise at low frequencies.
        { //overwrite alpha_coef with 1s if enhancement 5 is off.
            alpha_coef[k] = 1;
        }
#endif
/** @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ END OF ADDED CODE @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
    }
    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 */
    }

/** ################### Code added : Processing of frame ################### */
#if !OPT_BEST_CONFIG
    /* Check to switch enhancements configurations around */
    if(g_switchIt) //Checks if g_switchIt was changed (from Watch window only).
    {
        g_switchIt = 0; //restores g_switchIt to 0.
        switch_configuration(); //If it did, changes the enhancements configuration.
    }
#endif
    /* For VAD enhancement 9 only, reinitialises signal and noise powers */
    if(g_enhancement[9])
    {
        SNR.SPower = 0; //restores the signal power to 0.
        SNR.NPower = 0; //restores the noise power to 0.
    }

    /* Creates a complex array signal.X from the input frame inframe */
    for (k = 0; k < FFTLEN; k++)
    {
        signal.X[k] = cmplx(inframe[k],0);
    }
    /* FFT of signal.X */
    fft(FFTLEN, signal.X);



    for(k = 0; k < FFTLEN; k++)
    {
        //calculates the magnitude X from the signal X (complex).
        magnitude.X[k] = mag(signal.X[k]);
        if(g_enhancement[9])
        {
            SNR.SPower += sq(magnitude.X[k]); //calculates the signal power.
        }
        if(g_enhancement[1]) //updates the magnitude X and stores it in Xp
```

```
        {
            magnitude.X[k] = sqrt(lpf(sq(magnitude.X[k]), sq(magnitude.Xp[k]), K));
            magnitude.Xp[k] = magnitude.X[k];
        }
#if !OPT_BEST_CONFIG
        else if(g_enhancement[0]) //updates the magnitude X and stores it in Xp
        {
            magnitude.X[k] = lpf(magnitude.X[k], magnitude.Xp[k], K);
            magnitude.Xp[k] = magnitude.X[k];
        }
#endif
        //changes M1[k] to the measured magnitude X[k] if and only if it is smaller.
        M1[k] = min2F(M1[k],magnitude.X[k]);
        //updates the noise magnitude N.
        magnitude.N[k] = param.alpha * alpha_coef[k] * min4F(M1[k],M2[k],M3[k],M4[k]);
#if !OPT_BEST_CONFIG
        if(g_enhancement[2])
        {
            //low pass filter on noise magnitude (using N and Np)
            magnitude.N[k] = lpf(magnitude.N[k], magnitude.Np[k], K);
            //Np is the previous value of the noise magnitude N.
            magnitude.Np[k] = magnitude.N[k];
        }
#endif
        if(g_enhancement[9])
        {
            //Noise power calculation
            SNR.NPower += sq(magnitude.N[k]);// / param.alpha;
            //=min(M1,M2,M3,M4)*alpha_coef for optimal configuration.
        }
        //g_var1, g_var2 & g_calc set by default for easier calculations
        g_var1 = param.lambda;
        g_calc = magnitude.N[k] / magnitude.X[k];
        g_var2 = 1 - g_calc;
        if(g_enhancement[5])
        {
            g_calc = magnitude.N[k] / magnitude.Xp[k];
            g_var1 *= g_calc; //lambda * mag.N[k] / mag.Xp[k]
            g_var2 = 1 - g_calc; // 1 - mag.N[k] / mag.Xp[k]
        }
#if !OPT_BEST_CONFIG
        else if(g_enhancement[3])
        {
            g_var1 *= g_calc; // lambda * mag.N[k] / mag.X[k]
        }
        else if(g_enhancement[4])
        {
            // lambda * mag.Xp[k] / mag.X[k]
            g_var1 *= (magnitude.Xp[k]/magnitude.X[k]);
        }
        else if(g_enhancement[6])
        {
            // 1 - mag.N[k] / mag.Xp[k]
            g_var2 = 1 - magnitude.N[k]/magnitude.Xp[k];
        }
        else if(g_enhancement[7])
        {
            // sqrt(1 - (mag.N[k]*mag.N[k]) / (mag.X[k]*mag.X[k]))
            g_var2 = sqrt(1 - (sq(g_calc)));
        }
#endif
        //Y(w) = max(g_var1, g_var2) * X(w)
        signal.Y[k] = rmul(max2F(g_var1,g_var2), signal.X[k]);
#if !OPT_BEST_CONFIG
        //Calculates signal.Y[k] in function of the values of signal.X at k, k-1 & k-2
        if(g_enhancement[10]) //Residual noise reduction
        {
```

```
            //updates oldstate for the if condition below.
            oldstate = state;
            //updates the state for next iteration.
            state = ((magnitude.N[k]/magnitude.X[k]) > param.residue);
            /*If noise magnitude was previously >= half the input signal magnitude,
              the output signal is set to the minimum of the three adjacent frames.*/
            if(oldstate)
            {
                signal.Y[k] = min3C(signal.Y[k],signal.Y[k-1],signal.Y[k-2]);
            }
            else
            {
                signal.Y[k] = signal.Y[k-1];
            }
        }
#endif
    }

    if(g_enhancement[9]) //SNR-dependent alpha value
    {
        SNR.value = 10 * log10(SNR.SPower/SNR.NPower); // in decibels
        if(SNR.value < SNR.lower) //SNR < -5
        {
            param.alpha = SNR.alpha_high; //alpha = 4.5
        }
        else if(SNR.value > SNR.upper) //SNR > 20
        {
            param.alpha = SNR.alpha_low; //alpha = 1
        }
        else //line equation when -5 < SNR < 20
        {
            //Straight line equation
            float b = (SNR.alpha_low - SNR.alpha_high)/(SNR.upper - SNR.lower);
            float a = SNR.alpha_high - b * SNR.lower;
            param.alpha = a + b*SNR.value;
        }
    }

    /* Circular shift & update of buffers */
    if(++frame_n > LIM) //When we reach 2.5 seconds (frame_n=312)
    {
        frame_n = 0; //reinitialises frame_n
        //shift buffers
        Mx = M4;
        M4 = M3;
        M3 = M2;
        M2 = M1;
        M1 = Mx; //Mx is a temporary pointer
        for(k = 0; k < FFTLEN/2; k++)
        {
            M1[k] = magnitude.X[k];
            //updates M1 with the values of the input signal spectrum
        }
    }
    for(k = FFTLEN/2 + 1; k < FFTLEN; k++) //Symmetry of FFT with conjugates
    {
        signal.Y[k] = conjg(signal.Y[FFTLEN-k]); //conjugate
    }
    ifft(FFTLEN, Y); //performs the IFFT on the signal Y (complex)
    for(k = 0; k < FFTLEN; k++)
    {
        outframe[k] = real(signal.Y[k]);
    }
/** @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ END OF ADDED CODE @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */

    /* 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;
}

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


/** ################### Code added : Function definitions ################### */
float sq(float x)
{
    return x*x;
}

float mag(complex z) //Returns the magnitude of the complex number z.
{
    return sqrt(z.r * z.r + z.i * z.i);
}
/*This implementation is :
40 instruction cycles faster than cabs() (719 vs 759), for small numbers.
up to 50 instruction cycles faster for big numbers
all the numbers have 3 decimals after their point. */

float lpf(float present, float past, float pole)
{
    return present + pole*(past - present);
} //low pass filter (to smooth transition also)


#if !OPT_BEST_CONFIG
void switch_configuration(void) //switches configuration of enhancements.
{
    bool temp;
    int i;
    for(i = 0; i < MAX_ENHANCEMENTS; i++)
    {
        temp = g_enhancement[i];
        g_enhancement[i] = g_alt_enhancement[i];
        g_alt_enhancement[i] = temp;
```

```
    } //this switches around the two enhancements configurations.
}
#endif

float max2F(float x, float y)
{
    if(x > y)
    {
        return x;
    }
    return y;
}

float min2F(float x, float y)
{
    if(x < y)
    {
        return x;
    }
    return y;
}

float min4F(float a, float b, float c, float d)
{
    return min2F(min2F(a,b),min2F(c,d));
}

complex min3C(complex x, complex y, complex z)
{
    float temp = mag(x); //to avoid multiple mag() calculations
    complex result = x;
    if(temp > mag(y)) //x > y
    {
        temp = mag(y);
        result = y;
    }
    if(mag(z) < temp)
    {
        return z;
    }
    return result;
}
/** @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ END OF ADDED CODE @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ */
```

Appendix 2: MatLab script 2

```
%% MATLAB SCRIPT TO GENERATE FFTLEN ATTENUATION

%%
%% COEFFICIENTS FOR ALPHA (USED FOR NOISE ESTIM.)
%% AUTHORS: Quentin McGaw and Alexandra Rouhana
%% DATE: MARCH 2015
clc;
% ######## PARAMETERS ########
FFTLEN = 256; %number of coefficients
att_LF = 1.5; %Low frequency attenuation
freq_LP = 3; %Low pass frequency bin - 90 Hz
att_HF = 1.2; %High frequency attenuation
freq_HP = 10; %High pass frequency bin - 280 Hz
% ######## CALCULATIONS ########
coef = ones(FFTLEN,1); %all coef to 1
% ### Low pass ###
for i=0:(freq_LP-1)
    coef(i+1) = (freq_LP*att_LF - (att_LF-1)*i)/freq_LP;
```

```
    coef(FFTLEN-i) = coef(i+1); %symmetry
end
% ### High pass ###
a_eq = (att_HF-1)/((FFTLEN/2)-freq_HP); %2/9.8 = 0.2
b_eq = 1 - a_eq*freq_HP; %1 - 6.12
for i=freq_HP:(FFTLEN/2)
    coef(i+1) = a_eq*i + b_eq;
    coef(FFTLEN+1-i) = coef(i+1); %symmetry
end
% ######## RESULTS ########
% ### Plot ###
plot(1:FFTLEN,coef)
axis tight
title('Alpha exaggeration coefficient')
xlabel('Frequency index')
ylabel('Amplification factor')
% ### Writing ###
f=fopen('attenuation2.txt','w'); %open file
fprintf(f,'float alpha_coef[] = {');
fprintf(f,'%1.16f,',coef(1:FFTLEN-1));
%16f to avoid rounding consequences
fprintf(f,'%1.16f};\n',coef(FFTLEN));
fclose(f); %close the text file
```