I, Alexandre Abou Chahine, pledge that this assignment is completely my own work, and that I did not take, borrow or steal work from any other person, and that I did not allow any other person to use, have, borrow or steal portions of my work. I understand that if I violate this honesty pledge, I am subject to disciplinary action pursuant to the appropriate sections of Imperial College London.

I, Theo Franquet, pledge that this assignment is completely my own work, and that I did not take, borrow or steal work from any other person, and that I did not allow any other person to use, have, borrow or steal portions of my work. I understand that if I violate this honesty pledge, I am subject to disciplinary action pursuant to the appropriate sections of Imperial College London.

RTDSP Project – Real-time Speech Enhancement

Alexandre Abou Chahine 00979047

Theo Franquet 00967383

Contents

I) Background and Purpose

The objective of this Project is to implement an algorithm able to process audio data in real time, to suppress the background noise present in a speech recording. Background noise is unknown and can be of any nature, with any intensity. The program is written in C (using the given `enhance.c`) and is run on the TI DSK board provided. Challenges of the project include: suppressing background noise as much as possible while keeping the speech as clear as possible, keeping the program as fast as possible to operate in real time and implementing, testing and combining enhancements to obtain an ideal program able to operate on diverse audio recordings.

This report will highlight and explain the various formulae and algorithms used in the program. It will also include and explain the noise canceling algorithm as well as discussions on the performance of various enhancements.

II) Program Operation and Basic Implementation

i) Introduction to algorithm structure and parameters

The most intuitive way of achieving our goal would be to detect and extract the speech of the recording using spectral subtraction. An effective technique for noise detection, called Voice Activity Detector (VAD), is out of the scope of this course. Hence it is recommended to use a more straightforward approach which will be described further into the report. The aim is to estimate the noise spectrum of a raw signal and subtract it from the original signal in the frequency domain. Initially the raw signal $X(\omega)$ is made up of both the speech component $Y(\omega)$ and a noise component $N(\omega)$ such that $X(\omega) = Y(\omega) + N(\omega)$. The system we aim at implementing is illustrated below.

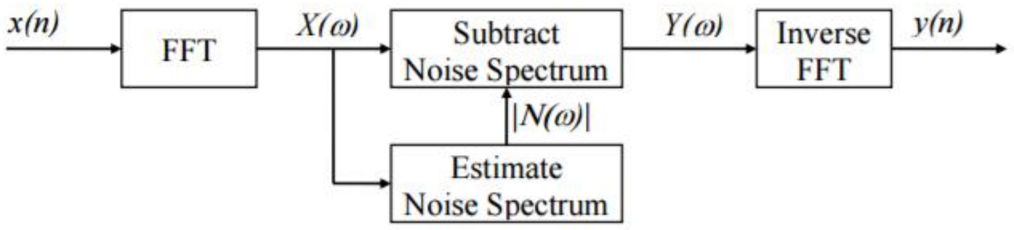


Figure 1: Noise subtraction circuit

All data processing is done in the frequency domain. For frequency domain processing, the continuous time signal is split into overlapping chunks called frames. Regarding the Discrete Fourier transform, both FFT and IFFT algorithms, given in the header file `fft_functions.h`, are optimized to be much faster than the usual DFT and IDFT ones. They considerably reduce the computational time and can be used in real time operations. These functions are even more efficient when working with buffer sizes of power of 2. That is why the Frames are formed by 256 ($= 2^8$) samples arriving at a rate of 8 kHz, it takes 32ms to fill a frame. Each frame is multiplied by the square root of the Hamming window before and after the frequency domain processing to avoid sharp amplitude changes in the time domain at the extremities of each frame. Then, one after the other, the frames are transformed in the frequency domain, processed, and inverse-transformed in the time domain. Frames are then added together in series (with overlapping, more on this later) to re-obtain a continuous time signal.

ii) Oversampling

If the signal is simply chunked into consecutive frames the discontinuities following the windowing will cause distortion in the FFT. By splitting the signal into overlapping frames, it repeats processing for same samples and allows the FFT to be evaluated 4 times per frequency bin, increasing frequency resolution thus avoiding spectral artefacts and power loss. Each frame starts $\frac{frame\ length}{oversampling\ ratio}$ after the previous one. If the oversampling ratio is too low, it might lead to distortions depending on the gain difference from one frame to another after processing. Note that $oversampling\ ratio = 4$ is chosen for this project.

iii) Estimating the noise spectrum

An important part of the speech enhancement procedure described previously is to estimate the noise spectrum present in the recording. Noise is estimated over a 10 second window using a method limiting the amount of audio data being stored.

Assuming the speaker does not speak for 10 seconds straight, it is possible to estimate the background noise when there is a break in his speech. We can determine the minimum magnitude in each frequency bin in the Fourier transform that has been present in any frame over the past ten seconds. We use four rotating buffers to keep the minimum magnitude in each frequency bin over 4x2.5 seconds.

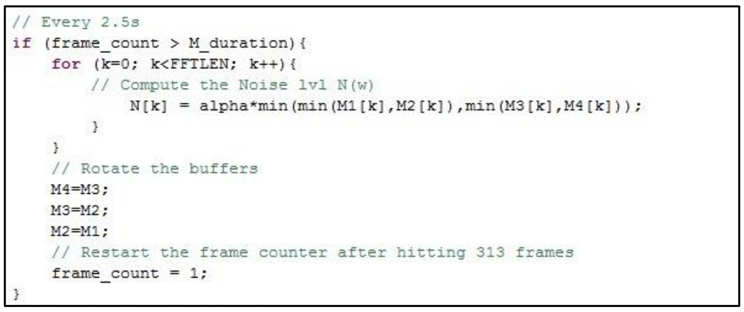
We evaluate the minimum amplitude obtained at each frequency bin in a frame, representing the underestimated amplitude of the background noise. This value is then compared to the minimums found for the other frames of the same 2.5 second window, giving a new array of minimums $M_1(\omega)$. Each element of $M_1(\omega)$ is then compared to the minimums found for the 3 previous 2.5 second windows and multiplied by an α factor to compensate underestimation ($\alpha \geq 1$):

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

The number of frame processed in 2.5seconds of speech is:

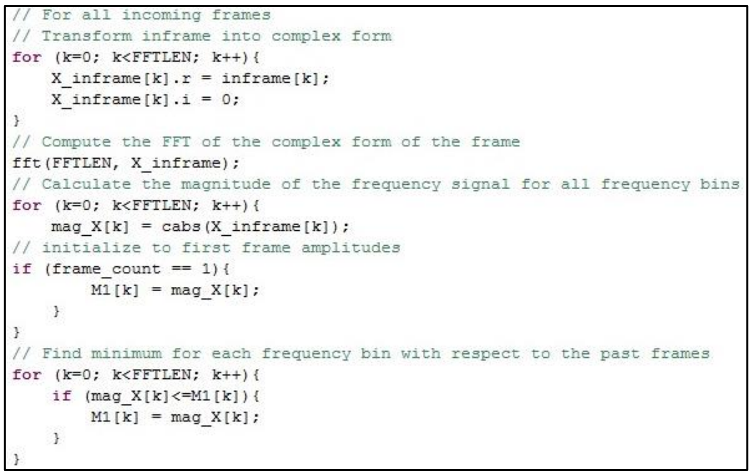
$$M_{duration} = \frac{time * over\_sampling\_ratio}{frame\_rate} = \frac{2.5 * 4}{0.032} = 312.5$$

Thus after 312 frames (equivalent of 2.496 seconds of speech) the buffers rotate switching pointer values ($M_i(\omega)$ transfers to $M_{i+1}(\omega)$) and $M_1(\omega)$ is calculated again. The following code implements the initial processing of an input frame and the noise estimation:



```
// Every 2.5s
if (frame_count > M_duration){
    for (k=0; k<FFTLEN; k++){
        // Compute the Noise lvl N(w)
            N[k] = alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
        }
    }
    // Rotate the buffers
    M4=M3;
    M3=M2;
    M2=M1;
    // Restart the frame counter after hitting 313 frames
    frame_count = 1;
}
```

Figure 2 Buffers rotation and noise computation



```
// For all incoming frames
// Transform inframe into complex form
for (k=0; k<FFTLEN; k++){
    X_inframe[k].r = inframe[k];
    X_inframe[k].i = 0;
}
// Compute the FFT of the complex form of the frame
fft(FFTLEN, X_inframe);
// Calculate the magnitude of the frequency signal for all frequency bins
for (k=0; k<FFTLEN; k++){
    mag_X[k] = cabs(X_inframe[k]);
// initialize to first frame amplitudes
if (frame_count == 1){
        M1[k] = mag_X[k];
    }
}
// Find minimum for each frequency bin with respect to the past frames
for (k=0; k<FFTLEN; k++){
    if (mag_X[k]<=M1[k]){
        M1[k] = mag_X[k];
    }
}
```

Figure 3 Input frame pre-processing

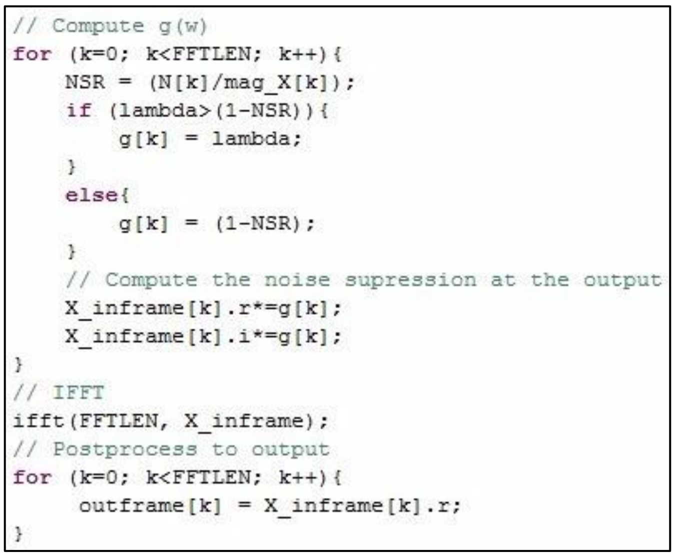
iv) Noise spectrum subtraction

We can only subtract the signal magnitudes in the frequency domain, as no information is found on the phase of $X(\omega)$, the goal is to obtain the following spectrum $|Y(\omega)|$:

$$|Y(\omega)| = |X(\omega)| - |N(\omega)| = |X(\omega)|\left(\frac{|X(\omega)| - |N(\omega)|}{|X(\omega)|}\right) = |X(\omega)|\left(1 - \frac{|N(\omega)|}{|X(\omega)|}\right) = |X(\omega)| \times g(\omega)$$

Note that $\left(1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$ might become negative (recall that $N(\omega)$ is only an estimation) which would lead to unwanted results (negative amplitude in the frequency domain). We modify the expression of $g(\omega)$ to overcome this: $g(\omega) = \max\left(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$ using a value close to zero for λ. Note that $g(\omega)$ is real: each output bin is simply a scaled version of its respective input bin.

The following code implements noise subtraction and the output frame:



```
// Compute g(w)
for (k=0; k<FFTLEN; k++){
    NSR = (N[k]/mag_X[k]);
    if (lambda>(1-NSR)){
        g[k] = lambda;
    }
    else{
        g[k] = (1-NSR);
    }
    // Compute the noise supression at the output
    X_inframe[k].r*=g[k];
    X_inframe[k].i*=g[k];
}
// IFFT
ifft(FFTLEN, X_inframe);
// Postprocess to output
for (k=0; k<FFTLEN; k++){
     outframe[k] = X_inframe[k].r;
}
```

Figure 4 Noise subtraction and output calculation

After noise subtraction and IFFT, the result is already noticeable: background noise volume is reduced and speech is more perceivable, however, some issues remain: abrupt discontinuities at each Mi buffer rotation, noticeable speech distortion, remaining musical noise, performance dependent on a single α value.

III) Algorithm enhancements

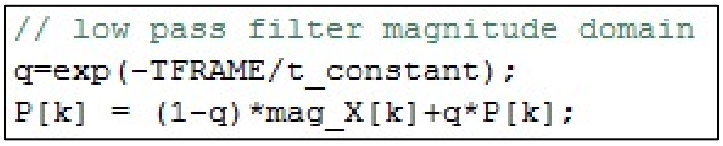
The following enhancements are implemented at the scale of a single frame and their performance evaluated to try and improve the quality of the audio output. All the enhancements are tested separately (different cases in the code) while keeping track (best case) of the most improving ones highlighted in red:

i) Enhancement A, low pass $|X(\omega)|$ for noise estimation

Low pass filtering the `inframe` amplitude spectrum for noise estimation aims at removing musical noise by suppressing abrupt changes in amplitude for a given frequency bin from one frame to another. This filter operates on the magnitude spectrum of the current frame, using information from adjacent frames, not on the time domain version of the frame. It acts like a first order IIR filter and allows to smooth out undesirable discontinuities in the spectrum for adjacent frames. Concretely it muffles high frequency (fast) changes in the spectrum for each frequency bin by using past input frames: $P_t(\omega) = (1-k)|X(\omega)| + kP_{t-1}(\omega)$ where the smoothing factor is equal to: $k = \exp\left(-\frac{T}{\tau}\right)$. The time constant is inversely proportional to k, and determines the cutoff frequency of the filter. For this project, it should be in the range of $20ms$ to $80ms$.

A suitable value (enough smoothing without affecting the changes in speech) for the time constant is $20ms$ giving a value of k equal to: 0.201.

Filtering is simply implemented following the expression for $P_t(\omega)$. The value for $|N(\omega)|$ is then computed using $P_t(\omega)$ instead of $|X(\omega)|$.



```
// low pass filter magnitude domain
q=exp(-TFRAME/t_constant);
P[k] = (1-q)*mag_X[k]+q*P[k];
```

Figure 5 Low pass filter input

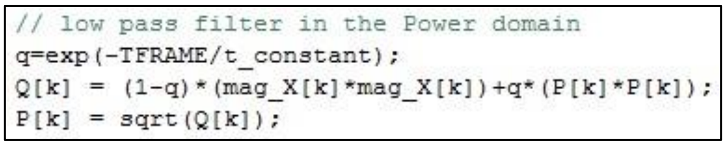
ii) Enhancement B, low pass filtering in the power domain

Here, we are looking to change from the magnitude to the power domain in which the input FFT signal is filtered. Instead of $|X(\omega)|$ we filter $|X(\omega)|^2$ which results in using the following formula:

$$Q_t(\omega) = (1-k)|X(\omega)|^2 + kQ_{t-1}(\omega)$$

$$P_t(\omega) = \sqrt{Q_t(\omega)}$$

Filtering in the power domain improves the overall perceived performance: rapidly changing frequency amplitudes are better identified and smoothed out. The new version of the code is given below:



```
// low pass filter in the Power domain
q=exp(-TFRAME/t_constant);
Q[k] = (1-q)*(mag_X[k]*mag_X[k])+q*(P[k]*P[k]);
P[k] = sqrt(Q[k]);
```

Figure 6 Low pass filter input in the power domain

Musical noise suppression is improved overall, similarly to enhancement A.

iii) Enhancement C, low pass filtering of the noise

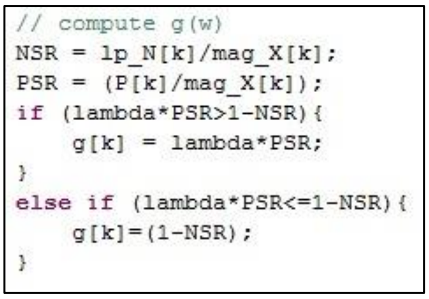
Low pass filtering the noise estimate can be done either in the magnitude or power domain using the same formulas of enhancement A and B. It gives us a smooth changes in noise magnitude estimation, avoiding strong discontinuities. We get a better attenuation of the musical noise. Once again filtering in the power domain appears to be more beneficial in the overall perceived performance.

iv) Enhancement D, choosing the best function $g(\omega)$

Choosing the right function $g(\omega)$ is also important: it influences how noise is subtracted from the original signal, and what factors are involved during this process. As of now, the value of λ is arbitrarily set in a range of 0.001 to 0.1. Using a λ dependent on other signal factors at a given frequency bin might improve the performance of the crucial noise subtraction step. Several versions of $g(\omega)$ are given, each of them is implemented and tested. The one with the best performance is found to be:

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

This confirms the hypothesis of improving performance using a variable λ. In this case, λ depends on the ratio of magnitudes of the filtered $X(\omega)$ (namely $P(\omega)$) and $X(\omega)$ itself. The new version of $g(\omega)$ is changed in the code:

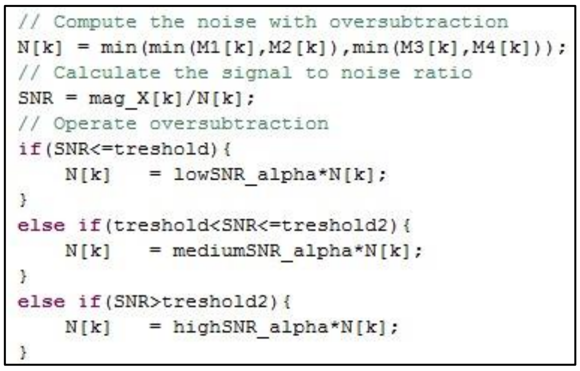


```
// compute g(w)
NSR = lp_N[k]/mag_X[k];
PSR = (P[k]/mag_X[k]);
if (lambda*PSR>1-NSR){
    g[k] = lambda*PSR;
}
else if (lambda*PSR<=1-NSR){
    g[k]=(1-NSR);
}
```

Figure 7 Optimal g function

v) Enhancement E, oversubstraction

Oversubstraction is the process of increasing α when estimating $|N(\omega)|$ for frequency bins having low SNR. This will lead to attenuating these unwanted frequency bins during the incoming noise subtraction. This technique gives very good results as we select how much each individual frequency bin must be suppressed. To implement oversubstraction, we first evaluate the SNR for each sample of `X_inframe`: $SNR = \frac{|X(\omega)|}{|N(\omega)|}$ and compare it to two thresholds $Thr1$ and $Thr2$. For SNRs lower than $Thr1$, we set α to 8 and for SNRs between $Thr1$ and $Thr2$, we set α to 4. If the SNR performance is good, we set it to unity. This is implemented as follows:



```
// Compute the noise with oversubtraction
N[k] = min(min(M1[k],M2[k]),min(M3[k],M4[k]));
// Calculate the signal to noise ratio
SNR = mag_X[k]/N[k];
// Operate oversubtraction
if(SNR<=treshold){
    N[k]    = lowSNR_alpha*N[k];
}
else if(treshold<SNR<=treshold2){
    N[k]    = mediumSNR_alpha*N[k];
}
else if(SNR>treshold2){
    N[k]    = highSNR_alpha*N[k];
}
```

Figure 8 Computation of noise with oversubtraction

Using two thresholds increase the accuracy of the algorithm as three values of α are available. Using a single threshold value would decrease the accuracy of the system in choosing a suitable value of α. Using two threshold values is a good compromise between accuracy and speed of the program (limited number of comparisons). Specific values of $Thr1$ and $Thr2$ were fine-tuned by listening to the output.

vi) Enhancement F, changing frame length

Changing the frame length corresponds to increasing/decreasing the time length of each frame and the resolution of the FFT. However, if the number of samples is too low, the musical noise is increased as half the samples are common from a frame to the next (vs. 75% for frame of 256 samples), leading to discontinuities in the frequency domain. If it is too high, the computing time increases and frame rate decreases. The initial low pass filter introduced make the smoothing far too strong when the frame rate drops and the output becomes completely indistinguishable.

The performance was tested for twice as much and twice less samples (512 & 128 elements respectively). As expected the short frame result in speech distortion and longer frames sound slurred. The frame length is consequently kept at the optimal value of 256 samples using the same time constant as before.

vii) Enhancement G, residual noise reduction

A proposed enhancement is to select the output that has a high SNR for a certain value of ω amongst 3 values of $Y(\omega)$: $Y_{t-1}(\omega), Y_t(\omega), Y_{t+1}(\omega)$ with t the frame number. This can help poor quality outputs without having any impact on the speech itself as each frame is separated by just $\frac{2.5\ seconds}{312\ frames} \approx 8ms$.

Implementing this enhancement involves a lot of supplementary processing (computing and storing the SNR for each frequency bin and often comparing three values per frequency bin processing). Running our version of this enhancement led to slow frame processing and corrupted output audio.

With other enhancements giving reasonable results in terms of noise suppression, this residual noise suppression technique is left unused.

v) Enhancement H, estimating noise over shorter period

This enhancement consists in taking the minimum amplitude of $|X(\omega)|$ for each value of ω for noise estimation in windows shorter than 2.5s. This aims at making the system more responsive to any significant change in noise level or nature.

This is implemented by reducing the frame count used for the buffers' rotation. Previously the number of frame used to calculate $M_1(\omega)$ was 312; by reducing this number to 125 each buffer will store the minimum over 1 second of speech. Testing the system led to rather disappointing results for our testing recordings. In fact, noise level and nature remains constant in amplitude and nature overall during the recording, having an increased noise estimation rate does not help much in our case. Moreover, each time buffers rotate, slight discontinuities are heard through speech distortions. Decreasing the time between each rotation increases overall distortion, degrading the performance of the system.

For this project, this enhancement is also left unused.

IV) Program Summary

i) Program skeleton

Each main step of the signal processing is illustrated in the flowchart below:

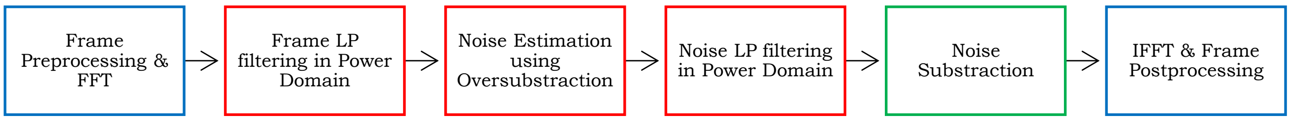

Frame Preprocessing & FFT
Frame LP filtering in Power Domain
Noise Estimation using Oversubstraction
Noise LP filtering in Power Domain
Noise Substraction
IFFT & Frame Postprocessing

Figure 9 Program flowchart

ii) Parameters Choice

Parameters have been chosen in order to find a good balance between voice clarity and noise suppression. The parameters chosen are summarized in the following table:

Parameter	**Value**
Alpha	2
Lambda	0.01
LowSNR Alpha	8
MidSNR Alpha	4
HighSNR Alpha	1
Output gain	3
$Thr1$	1.5
$Thr2$	7
Time constant	0.02

iii) Performance evaluation

The performance of the system is evaluated testing on 5 different recordings. Each one with different noise levels and topology. The quality of the output can be evaluated in three different ways: hearing the output of the system, looking at the time domain output and finally at the frequency output. For all the following tests the best version of our algorithm, using enhancements described previously, is being used:

a) Listening at the output

CAR1	Good noise suppression and speech clarity. Very small crackles in the background.
FACTORY2	Good background noise suppression and speech clarity. Perseverance of high pitch, abrupt background noises.
LYNX2	Background noise amplitude is very attenuated but still perceivable. Good speech clarity.
PHANTOM2	Similar to Factory2 with an addition of small musical noise
PHANTOM4	Background noise is still present with significant speech distortion. Speech remains understandable and is distinguishable from the noise, unlike in the raw recording.

b) Time Domain (CCS)

For time domain performance evaluation, we plot the input/output of the system for the same frame buffers for two recordings: Factory2 and Phantom2. The input frame signals are less smooth and have a lot more abrupt variations that corresponds to high frequency noise components. The output signals have attenuated amplitude before introducing the output gain (×3). Note that given output signal plots have been amplified by this $\alpha = 3$ factor.

Input frame (time domain) – Factory2:

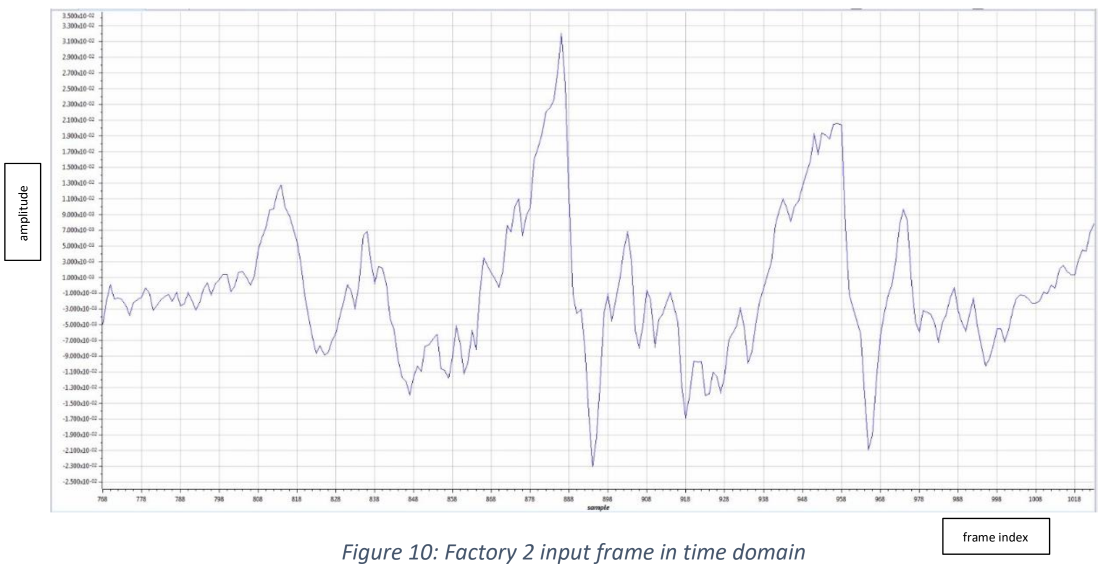


Figure 10: Factory 2 input frame in time domain

Output frame (time domain) – Factory2:

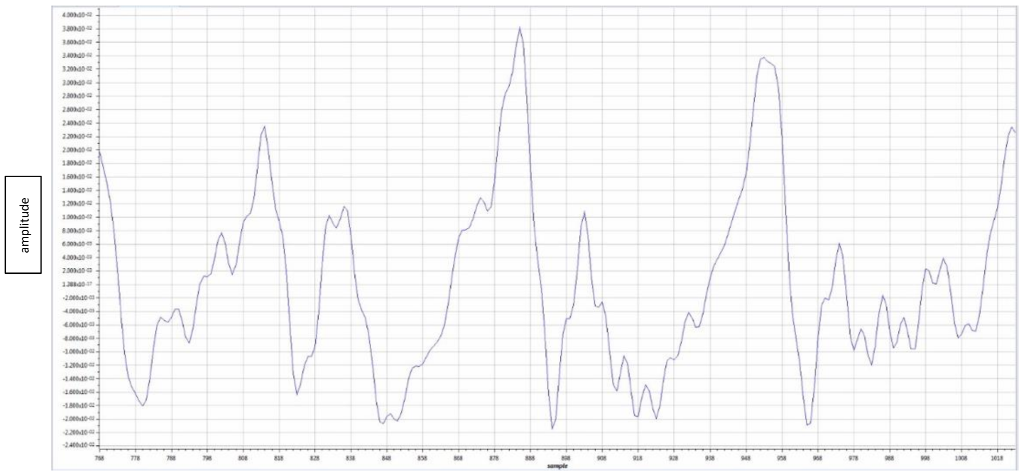


Figure 11: Factory 2 output frame in time domain

Input frame (time domain) – Phantom2:

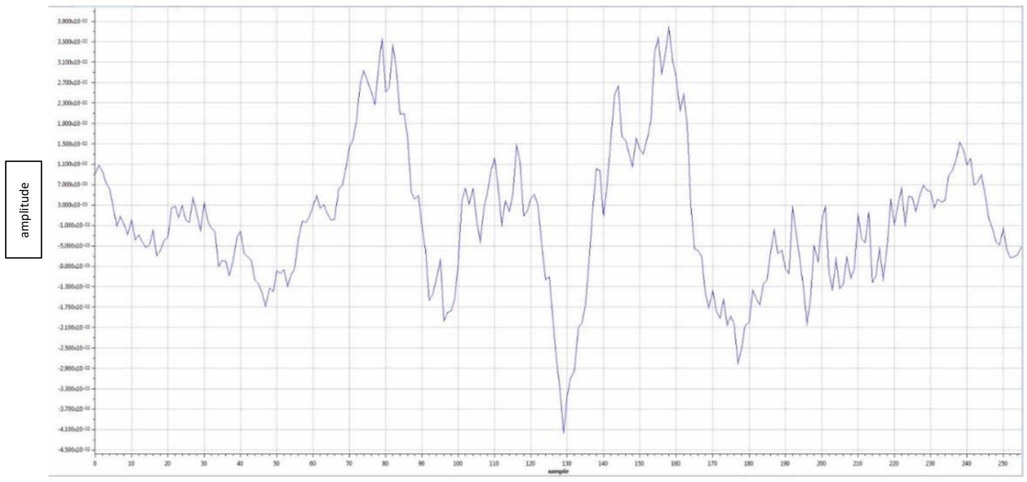


Figure 12: Phantom 2 input frame in time domain

Output frame (time domain) – Phantom2:

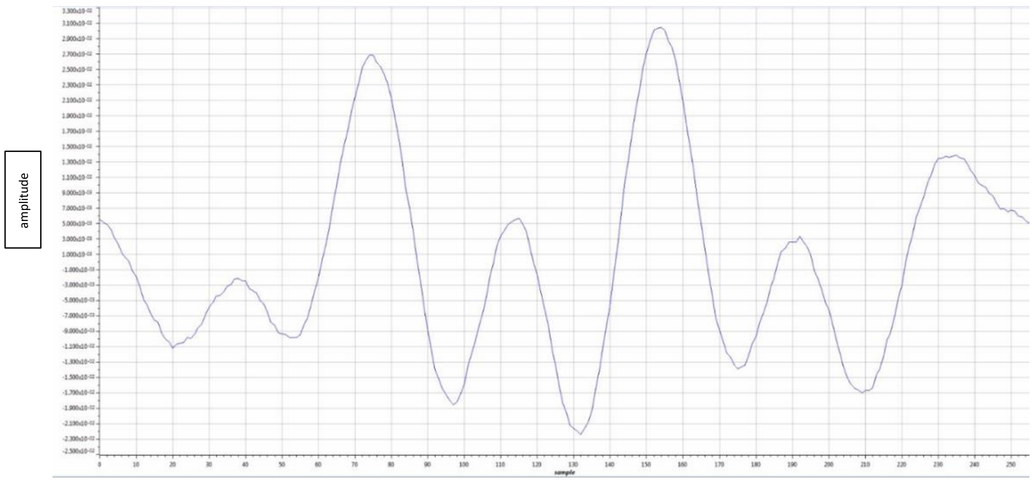


Figure 13: Phantom 2 output frame in time domain

c) Frequency Domain (CCS)

For frequency domain performance evaluation, we plot the input/output of the system for frame buffers for two recordings: a single 1kHz wave and Lynx2.

For a single 1kHz wave, we are expecting to achieve complete signal suppression: the program's noise estimator perceives the wave as noise leading to its attenuation:

Input frame (frequency domain):

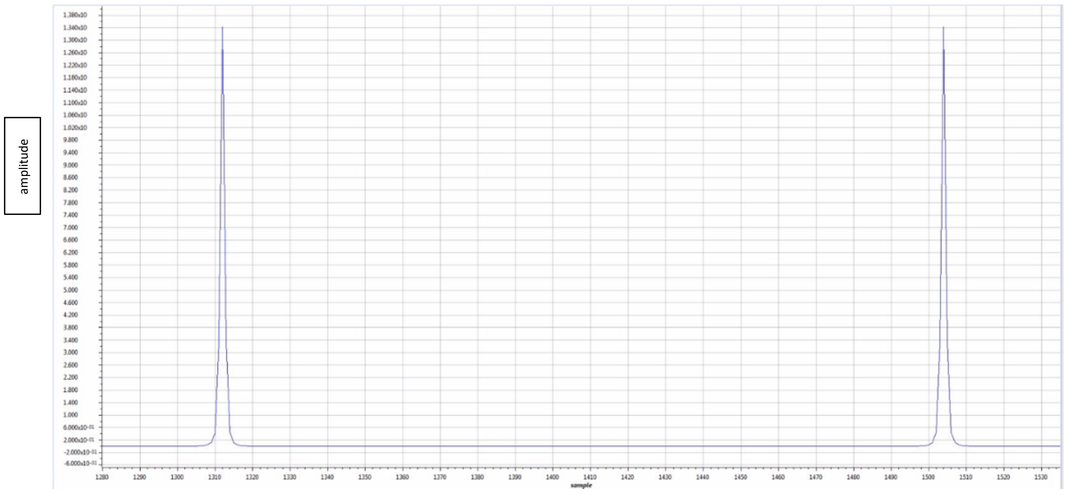


Figure 14: 1kHz sine wave input frame in frequency domain

Output frame (frequency domain):

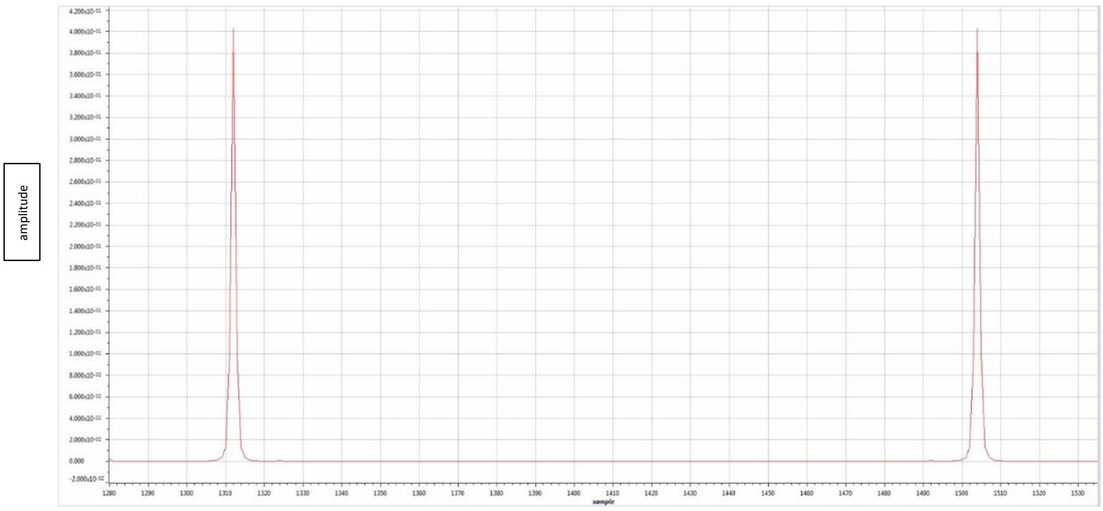


Figure 15: 1kHz sine wave output frame in frequency domain

We can clearly see that the frame processing algorithm attenuates the 1kHz signal by a factor of around 32 as expected. Input spectrum has a max magnitude of 13 and output spectrum a max magnitude of 0.4.

For the Lynx2 recording, we can also illustrate the performance of the noise estimator and subtraction below:

Input frame (frequency domain):

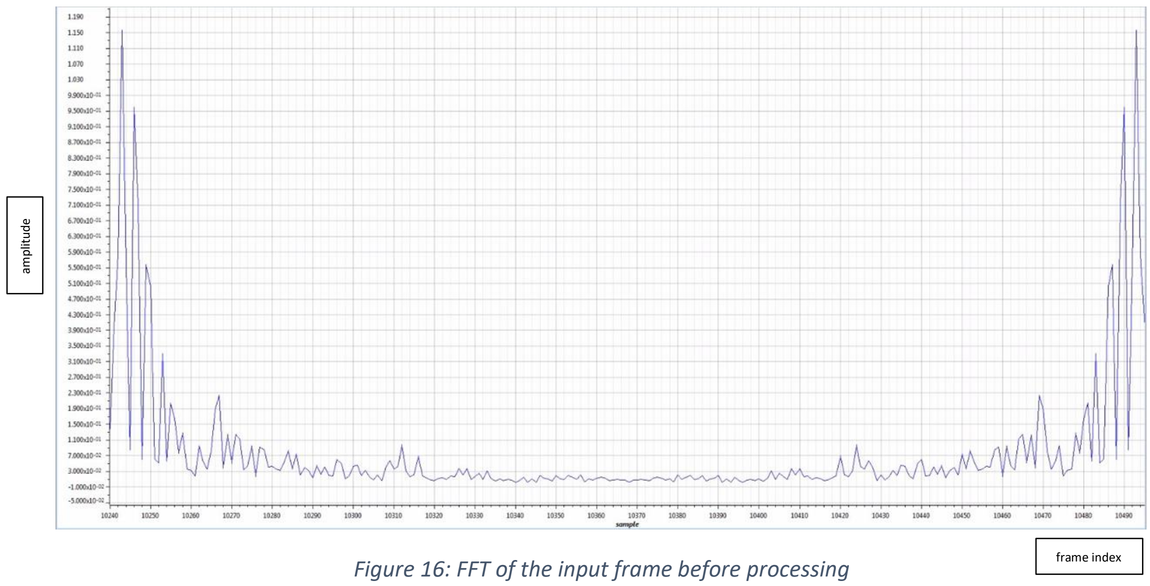


Figure 16: FFT of the input frame before processing

Output frame (frequency domain):

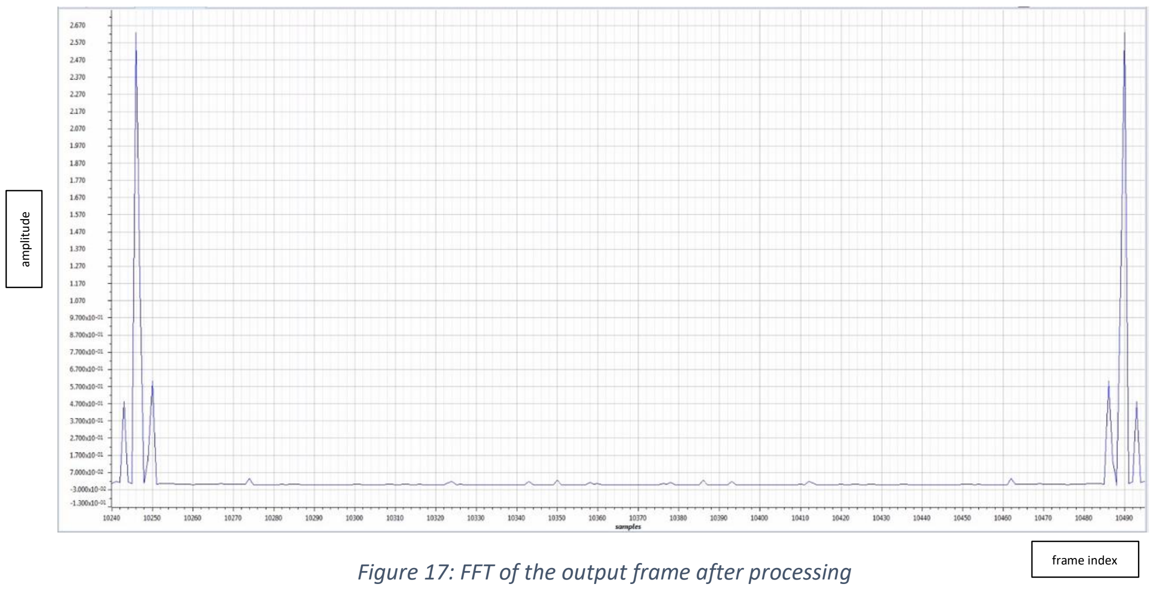


Figure 17: FFT of the output frame after processing

The output frame has been cleared of any noise. Only speech spectrum components remain. The noise estimator evaluates values of noise amplitudes and suppresses them to leave only frequencies corresponding to the speech itself as seen above.

d) Frequency Domain (Matlab Spectrogram)

Another way of evaluating the performance of our noise cancelling algorithm is to use a function in matlab called 'spectrogram'. This function takes as input a recording and outputs the relative magnitude (in dB/Hz) of each frequency component as a function of time. We show here the results for Lynx 2 and Phantom 4. It is clear that an important part of the noise has been removed. Human voice is usually between 85-180Hz, filtered outputs show these high amplitude frequencies at times when the speaker is active.

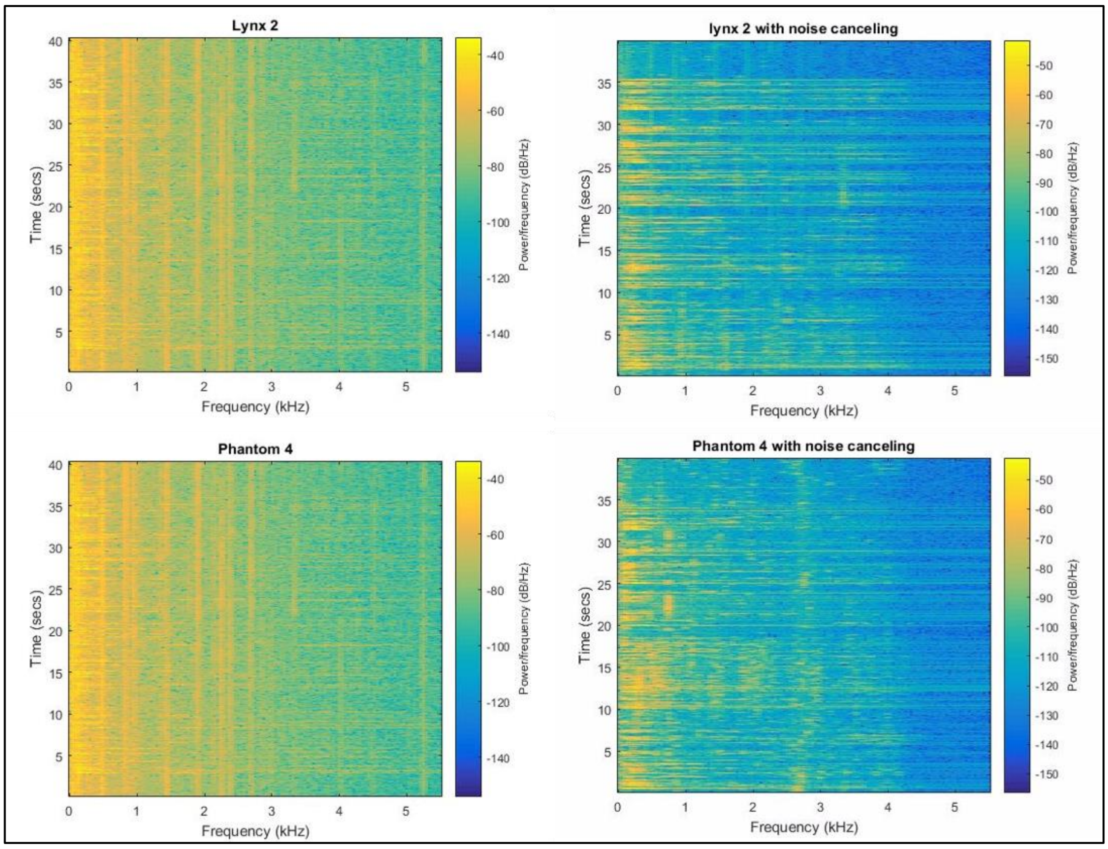


Figure 18: Spectrogram of Lynx 2 and phantom 4

Conclusion

The simple spectrum subtraction algorithm successfully attenuates background noise, keeping speech understandable. However, new issues of musical noise and speech distortions are introduced and supplementary enhancements are used to improve the output. To find a good compromise between both speech clarity and noise cancelation, several improvements have been implemented and tested and choices had to be made in order to obtain the best combination possible.

APPENDIX:

Best processing algorithm is towards the end: Case 10.

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

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

                                     PROJECT: Frame Processing

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

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

       *********************************************************************************
                                     By Danny Harvey: 21 July 2006
                                   Updated for use on CCS v4 Sept 2010
       ********************************************************************************/
                                                /*
          *      You should modify the code so that a speech enhancement project is built
                                    *  on top of this template.
                                                */
       /**************************** Pre-processor statements ******************************/

//  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  */
#define PI 3.141592653589793                /*PI defined here for use in your code */
#define TFRAME FRAMEINC/FSAMP               /* time between calculation of each frame */


/******************************* 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                    */\
                    /************************************************************************/
};

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

// Inout/Output variables
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 */
int io_ptr0;

// Indexes
int k;
int m;
int frame_count;                      /* Number of frame counted*/

// Pointers declarations
complex X_inframe[FFTLEN];            /* Complex equivalent of the new frame*/
complex Y_outframe[FFTLEN];
float *mag_X;                         /* Compute the Magnitude of X_inframe*/
float *mag_Y;                         /* Compute the Magnitude of X_inframe*/
float *P;                             /* Low pass magnitude signal in frequency domain */
float *Q;                             /* Low pass magnitude signal in power domain */
float *M1;                            /* Minimum spectrum 1 */
float *M2;                            /* Minimum spectrum 2 */
float *M3;                            /* Minimum spectrum 3 */
float *M4;                            /* Minimum spectrum 4 */
float *g;                             /* Frequency dependant gainb factor*/
float *N;                             /* Magnitude Noise in power domain */
float *lp_N;                          /* Low pass magnitude Noise in  frquency domain */
float *lp_NQ;                         /* Low pass magnitude Noise in  power domain */

// Sound parameters
float lambda = 0.01;                  /* lambda */
float t_constant = 0.02;
float t_constant_N = 0.02;            /* time constant */
float alpha = 2;                      /* alpha */
float treshold=1.5;                   /* SNR threshold */
float treshold2=7;                    /* SNR threshold */
float lowSNR_alpha=8;                 /* alpha when the SNR is below the treshold1 */
float highSNR_alpha=1;                /* alpha when the SNR is above the treshold2 */
float mediumSNR_alpha=4;              /* alpha when the SNR is between tresholds*/
float vol_gain = 3;                   /* Output volume gain */

// Computed parameters
float M_duration = OVERSAMP*2.5/(FFTLEN/FSAMP);      /* Number of frames for 2.5 secondes*/
float q;                                             /* k in the LPF formula */
float qN;
float NSR;                                           /* Noise to signal ratio*/
float SNR;                                           /* Signal to noise ratio*/
float NPR;                                           /* Noise to low pass signal ratio*/
float PSR;                                           /* Low pass signal to signal ratio*/

// Switches
int enhance=1;                                /* Switch between with and without processing */
int type=10;                                  /* Switch between different enhancement types */
int gfunction=0;                              /* Switch between different functions of g(w) */
int oversubtraction=1;                        /* Switch between with/without oversubtraction */

 /******************************** 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);
float max(float a,float b);

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

/*  Initialize and zero fill arrays */
inbuffer      = (float *) calloc(CIRCBUF, sizeof(float));  /* Input array */
outbuffer     = (float *) calloc(CIRCBUF, sizeof(float));  /* Output array */
inframe       = (float *) calloc(FFTLEN, sizeof(float));   /* Array for processing*/
outframe      = (float *) calloc(FFTLEN, sizeof(float));   /* Array for processing*/
inwin         = (float *) calloc(FFTLEN, sizeof(float));   /* Input window */
outwin        = (float *) calloc(FFTLEN, sizeof(float));   /* Output window */
mag_X         = (float *) calloc(FFTLEN, sizeof(float));
mag_Y         = (float *) calloc(FFTLEN, sizeof(float));
P             = (float *) calloc(FFTLEN, sizeof(float));
Q             = (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));
g             = (float *) calloc(FFTLEN, sizeof(float));
N             = (float *) calloc(FFTLEN, sizeof(float));
lp_N          = (float *) calloc(FFTLEN, sizeof(float));
lp_NQ         = (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;

        /* initialize the frame count */
        frame_count = 1;

        /* main loop, wait for interrupt */
        while(1){
        q=exp(-TFRAME/t_constant);
        qN = exp(-TFRAME/t_constant_N);
        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)
{
        /* 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 ***************************/

        // Switch between processing or not of the incoming signal
        switch (enhance){

//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////// Without Processing ////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

                case 0:
                for (k=0;k<FFTLEN;k++) {
                        outframe[k] = inframe[k]; /* copy input straight into output */
                }
                break;

//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////// Processing /////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

                case 1:

                // Preprocessing common to all enhancement

                // Every 2.5s or 312 frames counted
                if (frame_count > M_duration){

                        // compute the Noise lvl N(w) and amplify
                        for (k=0; k<FFTLEN; k++){
                        // switch between oversubtraction or usual noise calculating method
                                switch(oversubtraction){
                                        // Compute the noise with oversubtraction, enhancement 6
                                        case 1:
                                                N[k] = min(min(M1[k],M2[k]),min(M3[k],M4[k]));
                                                // Calculate the signal to noise ratio
                                                SNR = mag_X[k]/N[k];
                                                // Operate the oversubtraction with three alpha levels
                                                if(SNR<=treshold){
                                                        // for low SNR increase attenuation
                                                        N[k]   = lowSNR_alpha*N[k];
                                                }
                                                else if(treshold<SNR<=treshold2){
                                                        N[k]    = mediumSNR_alpha*N[k];
                                                }
                                                else if(SNR>treshold2){
                                                        // for high SNR keep an identical the signal
                                                        N[k]    = highSNR_alpha*N[k];
                                                }
                                        break;

                                        // Compute the noise without oversubtraction
                                        case 0:
                                                N[k] = alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
                                        break;
                                }

                                // Low pass filter the noise in the power domain
                                lp_NQ[k] = (1-qN)*(N[k]*N[k])+qN*(lp_N[k]*lp_N[k]);
                                lp_N[k] = sqrt(lp_NQ[k]);
                        }

                        // rotate the buffers and restart the frame count
                        M4=M3;
                        M3=M2;
                        M2=M1;
                        frame_count = 1;
                }

                // preprocess input frame, inframe, into complex form
                for (k=0; k<FFTLEN; k++){
                        X_inframe[k].r = inframe[k];
                        X_inframe[k].i = 0;
                }

                // Compute the FFT of the complex form of the frame
                // X_inframe is transformed from time domain to frequency
                fft(FFTLEN, X_inframe);

                // Calculate the magnitude of the frequency signal for all frequnecy bins
                for (k=0; k<FFTLEN; k++){
                        mag_X[k] = cabs(X_inframe[k]);

                // initialize to first frame amplitudes when we calculate a new value of M1
                if (frame_count == 1){
                                M1[k] = mag_X[k];
                        }
                }

                // Determine which enhancement we are using
                switch (type){

////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////// CASE 0 ////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

                case 0: // simple processing without any further enhancement

                // find minimum for a single frame
                for (k=0; k<FFTLEN; k++){
                        if (mag_X[k]<=M1[k]){
                                M1[k] = mag_X[k];
                        }
                }

                // compute g(w)
                for (k=0; k<FFTLEN; k++){
                        NSR = (N[k]/mag_X[k]);
                        if (lambda>(1-NSR)){
                                g[k] = lambda;
                        }
                        else{
                                g[k] = (1-NSR);
                        }
                }

                // supress noise and compute output
                for (k=0; k<FFTLEN; k++){
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }

                // IFFT
                // transform the output from the frequency to the time domain
                ifft(FFTLEN, X_inframe);

                // postprocess to output
                for (k=0; k<FFTLEN; k++){
                         outframe[k] = X_inframe[k].r;
                }

                // count up to the number of frames in 2.496s (312 frame)
                frame_count++;

                break;

////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////// CASE 1 ////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

                case 1: // simple processing w/ LowPass filter on the input frame

                // create a low pass P of the inframe
                for (k=1; k<FFTLEN; k++){
                        // low pass filter magnitude domain
                        P[k] = (1-q)*mag_X[k]+q*P[k];
                        if (P[k]<=M1[k]){
                                M1[k] = P[k];
                        }
                }

                // compute g(w)
                for (k=0; k<FFTLEN; k++){
                        NSR = (N[k]/mag_X[k]);
                        if (lambda>(1-NSR)){
                                g[k] = lambda;
                        }
                        else{
                                g[k] = (1-NSR);
                        }
                }

                // supress noise and compute output
                for (k=0; k<FFTLEN; k++){
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }

                // IFFT
                // transform the output from the frequency to the time domain
                ifft(FFTLEN, X_inframe);

                // postprocess to output
                for (k=0; k<FFTLEN; k++){
                         outframe[k] = X_inframe[k].r;
                }

                // count up to the number of frames in 2.496s (312 frame)
                frame_count++;

                break;

////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////// CASE 2 ////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

                case 2: // simple processing w/ LowPass filter in the power domain on the input

                // create a low pass P using the inframe
                for (k=1; k<FFTLEN; k++){
                        Q[k] = (1-q)*(mag_X[k]*mag_X[k])+q*Q[k];
                        P[k] = sqrt(Q[k]);
                        // find minimum for a single frame
                        if (P[k]<=M1[k]){
                                M1[k] = P[k];
                        }
```

```
                }

                // compute g(w)
                for (k=0; k<FFTLEN; k++){
                        NSR = (N[k]/mag_X[k]);
                        if (lambda>(1-NSR)){
                                g[k] = lambda;
                        }
                        else{
                                g[k] = (1-NSR);
                        }
                }

                // supress noise and compute output
                for (k=0; k<FFTLEN; k++){
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }

                // IFFT
                // transform the output from the frequency to the time domain
                ifft(FFTLEN, X_inframe);

                // postprocess to output
                for (k=0; k<FFTLEN; k++){
                         outframe[k] = X_inframe[k].r;
                }

                // count up to the number of frames in 2.496s (312 frame)
                frame_count++;

                break;

////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////// CASE 3 ///////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

                case 3: // simple processing w/ LowPass filtering input and noise in power domain

                // create a low pass P using the inframe
                for (k=1; k<FFTLEN; k++){
                        Q[k] = (1-q)*(mag_X[k]*mag_X[k])+q*Q[k];
                        P[k] = sqrt(Q[k]);
                // find minimum for a single frame
                        if (P[k]<=M1[k]){
                                M1[k] = P[k];
                        }

                // compute g(w)
                for (k=0; k<FFTLEN; k++){
                        NSR = (N[k]/mag_X[k]);
                        if (lambda>(1-NSR)){
                                g[k] = lambda;
                        }
                        else{
                                g[k] = (1-NSR);
                        }
                }

                // supress noise and compute output
                for (k=0; k<FFTLEN; k++){
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }

                // IFFT
                // transform the output from the frequency to the time domain
                ifft(FFTLEN, X_inframe);

                // postprocess to output
                for (k=0; k<FFTLEN; k++){
                         outframe[k] = X_inframe[k].r;
                }

                // count up to the number of frames in 2.496s (312 frame)
                frame_count++;
```

```
		break;

///////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////// CASE 4 /////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////////////////

		case 4: // processing w/ different g functions

		// create a low pass P in the power domain of the complex inframe
		for (k=0; k<FFTLEN; k++){
			Q[k] = (1-q)*(mag_X[k]*mag_X[k])+q*Q[k];
			P[k] = sqrt(Q[k]);
			// find minimum for a single frame
			if (P[k]<=M1[k]){
				M1[k] = P[k];
			}
		}

		switch (gfunction){

		case 1:
		// enhancement 5
			for (k=0; k<FFTLEN; k++){
				// compute g(w)
				NSR = (lp_N[k]/mag_X[k]);
				if (lambda>(1-NSR)){
					g[k] = lambda;
				}
				else{
					g[k] = (1-NSR);
				}
				// supress noise
				X_inframe[k].r*=g[k];
				X_inframe[k].i*=g[k];
			}
		break;

		case 2:
			for (k=0; k<FFTLEN; k++){
				// compute g(w)
				NSR = (lp_N[k]/mag_X[k]);
				if (lambda*NSR>(1-NSR)){
					g[k] = lambda*NSR;
				}
				else{
					g[k] = 1-NSR;
				}
				// supress noise
				X_inframe[k].r*=g[k];
				X_inframe[k].i*=g[k];
			}
		break;

		case 3:
			for (k=0; k<FFTLEN; k++){
				// compute g(w)
				PSR = (P[k]/mag_X[k]);
				if (lambda*PSR>1-(lp_N[k]/mag_X[k])){
					g[k] = lambda*PSR;
				}
				else{
					g[k] = 1-(lp_N[k]/mag_X[k]);
				}
				// supress noise
				X_inframe[k].r*=g[k];
				X_inframe[k].i*=g[k];
			}
		break;

		case 4:
			for (k=0; k<FFTLEN; k++){
				// compute g(w)
				NPR = (lp_N[k]/P[k]);
				if (lambda*NPR>(1-NPR)){
					g[k] = lambda*NPR;
				}
				else{
```

```
                                g[k] = 1-NPR;
                        }
                        // supress noise
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }
        break;

        case 5:
                for (k=0; k<FFTLEN; k++){
                        // compute g(w)
                        NPR = (lp_N[k]/P[k]);
                        if (lambda>(1-NPR)){
                                g[k] = lambda;
                        }
                        else{
                                g[k] = 1-NPR;
                        }
                        // supress noise
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }
        break;
        }

        // IFFT
        // transform the output from the frequency to the time domain
        ifft(FFTLEN, X_inframe);

        // postprocess to output
        for (k=0; k<FFTLEN; k++){
                 outframe[k] = X_inframe[k].r;
        }

        // count up to the number of frames in 2.496s (312 frame)
        frame_count++;

        break;


//////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////// CASE 5 ////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

        case 5: // processing w/ different g functions in the power domain

        // create a low pass P in the power domain of the complex inframe
        for (k=0; k<FFTLEN; k++){
                Q[k] = (1-q)*(mag_X[k]*mag_X[k])+q*Q[k];
                P[k] = sqrt(Q[k]);
                // find minimum for a single frame
                if (P[k]<=M1[k]){
                        M1[k] = P[k];
                }
        }

        switch (gfunction){

        case 1:
        // enhancement 5
                for (k=0; k<FFTLEN; k++){
                        // compute g(w)
                        NSR = (lp_N[k]/mag_X[k]);
                        NSR = NSR*NSR;
                        if (lambda>sqrt(1-NSR)){
                                g[k] = lambda;
                        }
                        else{
                                g[k] = sqrt(1-NSR);
                        }
                        // supress noise
                        X_inframe[k].r*=g[k];
                        X_inframe[k].i*=g[k];
                }
        break;

        case 2:
```

```
        for (k=0; k<FFTLEN; k++){
                // compute g(w)
                NSR = (lp_N[k]/mag_X[k]);
                NSR = NSR*NSR;
                if (lambda*NSR>(1-NSR)){
                        g[k] = lambda*NSR;
                }
                else{
                        g[k] = sqrt(1-NSR);
                }
                // supress noise
                X_inframe[k].r*=g[k];
                X_inframe[k].i*=g[k];
        }
break;

case 3:
        for (k=0; k<FFTLEN; k++){
                // compute g(w)
                PSR = (P[k]/mag_X[k]);
                NSR = (lp_N[k]/mag_X[k]);
                NSR = NSR*NSR;
                if (lambda*PSR>(1-NSR)){
                        g[k] = lambda*PSR;
                }
                else{
                        g[k] = sqrt(1-NSR);
                }
                // supress noise
                X_inframe[k].r*=g[k];
                X_inframe[k].i*=g[k];
        }
break;

case 4:
        for (k=0; k<FFTLEN; k++){
                // compute g(w)
                NPR = (lp_N[k]/P[k]);
                NPR = NPR*NPR;
                if (lambda*NPR>sqrt(1-NPR)){
                        g[k] = lambda*NPR;
                }
                else{
                        g[k] = sqrt(1-NPR);
                }
                // supress noise
                X_inframe[k].r*=g[k];
                X_inframe[k].i*=g[k];
        }
break;

case 5:
        for (k=0; k<FFTLEN; k++){
                // compute g(w)
                NPR = (lp_N[k]/P[k]);
                NPR = NPR*NPR;
                if (lambda>sqrt(1-NPR)){
                        g[k] = lambda;
                }
                else{
                        g[k] = sqrt(1-NPR);
                }
                // supress noise
                X_inframe[k].r*=g[k];
                X_inframe[k].i*=g[k];
        }
break;
}

// IFFT
// transform the output from the frequency to the time domain
ifft(FFTLEN, X_inframe);

// postprocess to output
for (k=0; k<FFTLEN; k++){
         outframe[k] = X_inframe[k].r;
}
```

```
                // count up to the number of frames in 2.496s (312 frame)
                frame_count++;

                break;


////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////// BEST CASE //////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

                case 10:

                // create a low pass P in the power domain of the complex inframe
                for (k=0; k<FFTLEN; k++){
                        // low pass filter in the Power domain
                        Q[k] = (1-q)*(mag_X[k]*mag_X[k])+q*(P[k]*P[k]);
                        P[k] = sqrt(Q[k]);
                        // find minimum for a single frame
                        if (P[k]<=M1[k]){
                                M1[k] = P[k];
                        }
                        // compute g(w) and calculate Y
                        NSR = lp_N[k]/mag_X[k];
                        PSR = (P[k]/mag_X[k]);
                        if (lambda*PSR>1-NSR){
                                g[k] = lambda*PSR;
                        }
                        else if (lambda*PSR<=1-NSR){
                                g[k]=(1-NSR);
                        }

                        X_inframe[k].r*=vol_gain*g[k];
                        X_inframe[k].i*=vol_gain*g[k];
                }

                // IFFT
                ifft(FFTLEN, X_inframe);

                // postprocess to output
                for (k=0; k<FFTLEN; k++){
                         outframe[k] = X_inframe[k].r;
                }

                // count up to the number of frames in 2.5s
                frame_count++;

                break;
                }
        break;
        }
/***************************************************************************************/

    // 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;
}

/************************************* FUNCTIONS USED *****************************************/

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

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