IMPERIAL COLLEGE LONDON

ELECTRICAL AND ELECTRONIC ENGINEERING
DEPARTMENT

Speech Enhancement Project

Authors
Alejandro Gilson Campillo CID: 01112712
Sara Mainiero CID: 01119882

January 23, 2020

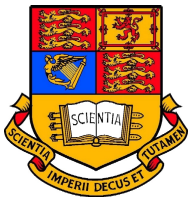


Table of Contents

1 Introduction

The final project of the RTDSP course concerned the implementation of a code for Speech Enhancement. Real time speech enhancement is the removal of the noise from a speech signal in real time. It is of essential importance, especially in devices such as mobile phones, as the quality of the speech has to be good and the message transmitted should be received clearly, being affected as little as possible by corruption. The technique used during the project was the *spectral subtraction* method.

1.1 Spectral subtraction method

The spectral subtraction method consists in removing the spectrum of the noise from the one of the original input signal. The noise is estimated based on the magnitude of the input and removed accordingly. The block diagram in figure 1 shows the process. It occurs in the frequency domain, hence the FFT and IFFT need to be taken. The frame processing technique used to ensure the program runs in real time is the *overlap and add technique*. According to this technique, the speech signal is divided in frames, that are processed separately. To ensure that the process happens fast enough, the various frames are overlapped on each other, and the result of their overlaps summed to each other. The amount by which they are overlapped is called the *oversampling ratio*. In the case of this specific project, the oversampling ratio is **4**. Hence, each frame starts at $\frac{1}{4}$ of the previous one. The drawback of using overlapping frames is the possibilities of discontinuities that can alter the signal. To minimize these discontinuities, a suitable solution is to multiply the input and output signals by a suitable windowing function. The chosen one is the **square root of the Hamming window**:

$$w(k) = \sqrt{1 - 0.85185 cos(\frac{(2k+1)\pi}{N}} \tag{1}$$

for k=0, 1, ..., N-1

This function is particularly suitable for the oversampling ratio chosen, as it has the property of always adding up to 1 for 4x frame oversampling. When the envelope of the overlapping windows adds up to 1, no further manipulation is required on the spectrum to perform processing without losing power in the signal.

1.1.1 Noise estimation method

The estimation of the noise is done using the *minimum noise method*. This method is based on the assumption that the user will pause, at least once, while speaking: at that particular point, any remaining magnitude is associated to noise. The speaker is assumed to stop talking at least once every 10 seconds. The most efficient method uses four different buffers, each storing 2.5 seconds

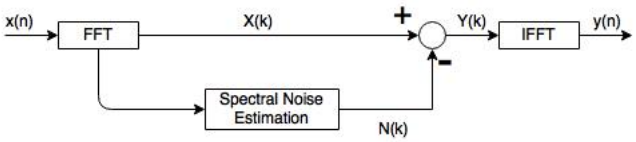


Figure 1: Spectral subtraction diagram

worth of signal. For each frequency component, the information stored in the four buffers is compared: the minimum value out of the four is saved in a separate buffer, hence the desired **minimum noise buffer**. To implement this in real time, the buffers are rotated. The "latest" buffer, M1, is compared with the minimum noise buffer: in case the values of N are smaller than the ones of M1, the latter buffer is updated with the values of the first one. This ensures that the estimation of the noise is correct, even in case of a change of situation and noise.

2 Basic Implementation

A skeleton code, performing buffer rotation and the overlap-add of frames, was already provided. To us student was left to implement the basic noise cancelling algorithm, by obtaining the minimum noise buffer and subtracting it from the original spectrum. This is done in the function *process-frame*.
Since the processing is done in the frequency domain, the **Fourier Transform** of the input signal needs to be taken. However, the *fft* function works with complex valued inputs. Therefore, the values in the input array, which are real values, need to be stored in another array, made of complex values. This is done by assigning their value to the real part of the complex-valued array:

```
//Making the input complex
for(k=0; k < FFTLEN; k++){
    cpx[k].r = inframe[k];
    cpx[k].i = 0;
}
```

The minimum noise buffer method works concerning the minimum **magnitude** of the noise. Therefore,the magnitude of the FFT values is taken, using the function *cabs*, which is included in the complex values library (used in this program):

```
//Calculating magnitude of input
for(k=0; k < FFTLEN; k++){
    mag[k]=cabs(cpx[k]);
}
```

The value of mag[k] is allocated to m1[k], so that the values just processed

are contained in the last section. To make sure that these values stored in m1[k] are the most recent ones, the four pointers are rotated:

```
//Buffer rotation
    p = m4;
   m4 = m3;
   m3 = m2;
   m2 = m1;
   m1 = p;
```

A frame counter, called *mag-index*, is incremented. When the index reaches 78 (hence, 2.5 seconds), the comparison between the four buffers is performed. The minimum value at each frequency bin is saved in a separate buffer called *min-mag*:

```
// Calculate which is smaller: M1, M2, M3 or M4?
while(k < FFTLEN){
   min_mag[k] = m1[k];
   if(min_mag[k] > m2[k]){
       min_mag[k] = m2[k];
   }
   else if(min_mag[k] > m3[k]){
      min_mag[k] = m3[k];
   }
   if(min_mag[k] > m4[k]){
      min_mag[k] = m4[k];
   }
   k++;
}
```

The buffer min-mag[k] represents the minimum noise buffer. For overestimation purposes, in order to achieve more significant results, it is multiplied by a value α. For the basic algorithm, α is around 20.

After obtaining the minimum noise buffer, the subtraction is performed. Direct spectral subtraction is not feasible, as the exact values of noise aren't known. They are approximated, and don't take into account the phase. Therefore, the best approach to obtain a filtered output is to reduce the magnitude of the input based on the magnitude of the noise. This is done by the following equation:

$$Y(k) = X(k)G(k) \tag{2}$$

$$\text{where } G(k) = max[\lambda, (1 - \frac{|N(k)}{|X(k)|})]$$

The comparison with the value λ is required to minimize the background musical noise. This kind of noise is worse when $\lambda = 0$, but improves when λ is a small value (slightly above zero). The multiplication is performed in the function *subtract-noise*: the value of G(k) is changed according to the comparison between the ratio and λ.

3 Enhancements

The basic algorithm described above is able to subtract, to some extent, the noise in the input signal. However, it is not perfect and, for this reason, further enhancements are carried out. In order to select which enhancements to use in the final implementation certain variables are defined. These act as switches and allow the user to turn them on or off at real time. Some enhancements are mutually exclusive and this is taken into account in the program. In order to increase the flexibility of the program, each enhancement is implemented in a separate function. Many times the function is called even when the enhancement is not used. This is because in some cases the function implementing the enhancement calculates a value that is used in another enhancement. This improves the efficiency of the overall program.

3.1 Enhancement 1: Low-pass filtered version of the input

The human ear is very sensitive to high pitch tones. It, therefore, makes sense to subtract high frequency noise. The information from the speech voice is already carried from lower frequencies, therefore, this trade-off is worth making use of. The low pass filter is accomplished using equation 3.

$$P_t(\omega) = (1-k) \times |X(\omega)| + k \times P_{t-1}(\omega), \tag{3}$$

$$\text{where } k = exp(-T/\tau)\ , \tau = [0.032, 0.08] \text{ and } T = TIMEFRAME$$

The implementation of this enhancement is shown in the code snippet bellow. After the new value of *mag* is computed. The program will run through other non-exclusive enhancements and will then be subtracted by the noise estimate.

```
void enhancement1(){
   float ex;
   int n;
   ex = pow(2.71828, (-0.25/taus)); //Calculate exponential
   p1[0] = (1 - ex) * mag[0];       // First iteration of the equation
   for(n = 1; n < FFTLEN; n++){
      p1[n] = (1 - ex) * mag[n] + ex * p1[n-1]; //Difference equation
   }
   if(enhancement_1 && !enhancement_2 ){ // Enhancement 1 must be on
      for(n = 0; n < FFTLEN; n++){
         // New value of magnitude is the low pass filtered version
         mag[n] = p1[n];
      }
   }
}
```

3.2 Enhancement 2: Low-pass filtered version of the input in power domain

This enhancement builds on the previous one. Here instead of taking the magnitude of the signal, the power of it is employed. The power is simply the magnitude squared. When a range of values is squared, those that are small become smaller and those that are big become larger. Since the noise generally has smaller magnitude than the speech this will increase the amount of noise subtracted by the low pass filter. The implementation is shown bellow. The steps following this code are as explained above in enhancement 1.

```
void enhancement2(){
   float ex;
   int n;
   ex = pow(2.71828, (-0.25/taus)); // Calculate the exponential
   p2[0] = sqrt((1 - ex) * mag[0]*mag[0]); // First iteration of the
       equation
   for(n = 1; n < FFTLEN; n++){
      p2[n] = sqrt((1 - ex) * mag[n]*mag[n] + ex * p2[n-1]*px[n-1]);
   }
   if(enhancement_2){ // Implement the enhancement if desired
      for(n = 0; n < FFTLEN; n++){
         mag[n] = p2[n];
```

3.3 Enhancement 3: Low-pass filtered version of the noise

Every time a new noise estimator is calculated, the frequency domain spectrum of what the listener hears changes. This is perceived in the form of abrupt discontinuities. In order to reduce this effect, the low pass filtered version of the estimated noise can be calculated and then used in the noise subtraction algorithm. This algorithm works because discontinuities correspond to fast changes. Fast changes are translated into high frequency components in the frequency domain. A low pass filter will help avoid them. The implementation is shown in the code snippet bellow:

```
 void enhancement3(){
   float ex;
   int n;
   ex = pow(2.71828, (-0.25/taun)); // Compute the exponential
   p3[0] = (1 - ex) * min_mag[0];  //First iteration of difference
       function
   for(n = 1; n < FFTLEN; n++){ // Calculate the difference equation
      p3[n] = (1 - ex) * min_mag[n] + ex * p3[n-1];
   }
   if(enhancement_3){ // Implement the enhancement if desired
      for(n = 0; n < FFTLEN; n++){
         min_mag[n] = p3[n]; //Assign noise estimator a new value
```

The students found that this enhancement reduced the amount of noise subtracted from high frequency components (the most sensitive to the human ear) and decided not to make use of it.

3.4 Enhancement 4: Variation of G(ω)

When subtracting the noise, $G(\omega)$ chooses the maximum between a chosen parameter λ and a function dependent on the magnitude of the signal and the noise (or a modification from an enhancement) as shown in 4. The human ear is not pleased by abrupt changes of amplitude in the sound. Moreover, low levels of noise are pleasant. This is the reason for the use of λ. In this series of enhancements several variations of $G(\omega)$ will be carried out. Each of them play with the magnitude and the power of the signal in different configurations. The terms that make use of the power become larger and, therefore, change the amount of noise to be subtracted form the signal.

$$G(\omega) = max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}) \tag{4}$$

The different variations of G(ω) are listed bellow:

$$G(\omega) = max(\lambda \frac{|N(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|}) \tag{5}$$

$$G(\omega) = max(\lambda \frac{|P(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|}) \tag{6}$$

$$G(\omega) = max(\lambda \frac{|N(\omega)|}{|P(\omega)|}, 1 - \frac{|N(\omega)|}{|P(\omega)|}) \tag{7}$$

$$G(\omega) = max(\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|}) \tag{8}$$

The implementation of the first enhancement of this type is displayed bellow. As it can be seen, only one enhancement can be activated. If two switches are on corresponding to any of these enhancements then the first one in augmenting order will be the one carried out. The subtraction of the noise is carried out in the function itself.

```
void enhancement4(){
  int k;
  if (enhancement_4_1){ //Implement the enhancement if desired
    for(k = 0; k < FFTLEN; k++){
     // Multiply noise estimator by scaling factor
      min_mag[k] = alpha_array[k]*min_mag[k];
       // Second term of g[k] function
      g[k] = (1 - (min_mag[k]/mag[k]));
      if((lambda*(min_mag[k]/mag[k])) > g[k]){ //Find maximum
      //First term of g[k] function
```

```
            g[k] = lambda*(min_mag[k]/mag[k]);
          }
          cpx[k].r = cpx[k].r * g[k]; //Multiply signal by gain factor
          cpx[k].i = cpx[k].i * g[k];
       }
    }
    // Implement enhancement (only one can be computed)
    else if (enhancement_4_2){
    //Function continues
}
```

3.5 Enhancement 5: Variation of G(ω) in the power domain

This improvement builds on enhancement 4. It is simply the square root of the right hand side term of $G(\omega)$, where the noise and signal magnitudes are squared. This increments the amount of signal to be subtracted. The students have found it to be particularly effective on audio with low SNR. A basic example is shown in 9 and its implementation is displayed bellow:

$$G(\omega) = max(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}}) \tag{9}$$

```
void enhancement5(){
   int k;
   for(k = 0; k < FFTLEN; k++){
      min_mag[k] = alpha_array[k]*min_mag[k]; //Use the scaling factor
          alpha
      g[k] = sqrt(1 - ((min_mag[k]*min_mag[k])/(mag[k]*mag[k])));

      if(lambda > g[k]){
         g[k] = lambda;
      }
      //Subtract noise from the signal
      cpx[k].r = cpx[k].r * g[k];
      cpx[k].i = cpx[k].i * g[k];
   }
}
```

3.6 Enhancement 6: Over-subtraction

The noise spectrum is not the same at all frequencies. It would be more effective, therefore, to subtract different amounts of signal at different frequencies. This will increase the SNR. Moreover, it will also reduce the musical noise that remains when the audio is processed. This is because those artifacts come from the fact that the noise has been modulated and not completely eliminated. In

order to carry out this improvement the SNR at each bin is calculated and for those that have a poor ratio, the scaling factor α of their noise estimation is increased. This will increase the subtraction at those bins and will, therefore, improve the output SNR. The implementation is shown bellow:

```
void enhancement6() {
   int k;
   for (k = 0; k < FFTLEN; k++) {
      // Initialize alpha_array
      alpha_array[k] = alpha;
      snr = mag_original[k] / min_mag[k]; //Calculate the SNR
      if (snr < threshold_snr) { //If SNR is too bad
         alpha_array[k] = new_alpha; //Assign new alpha to that bin
      }
   }
}
```

The use of this enhancement created the need for an array storing different values of α. This array was called *alpha-array*. This change was introduced into all the enhancements that made use of this constant.

3.7 Enhancement 7: Modification of frame lengths

At the beginning, the frame lengths were set to a size of 256 samples. It is important to keep this number to a power of two in order to be able to compute the FFT and increase the speed of the transform. Modifying this value has an effect on the overall performance of the denoiser. When smaller values are used, fewer amplitude bumps are observed. However, there is a higher level of the "musical noise" artifact that comes from not completely eliminating noisy components. On the other hand, when a larger value is used (such as 512) the sound does not improve significantly. Moreover, other issues can add up such as an excessive use of memory since the frames are now longer.

3.8 Enhancement 8: Variation of noise-estimation period

The initial noise estimation period was set to 78 frames. This corresponds to 2.5 seconds over which the minimum magnitude bins were calculated. It was suggested to reduce the this period and observe its effect. The students found the audio to be too distorted since it was changing the noise estimator more frequently. However, the students chose to do just the opposite: increase the period of estimation to 156 frames. This minimized the amount of amplitude bumps since the noise estimation changed at a lower frequency. The estimator was still very accurate making this change a valuable trade off.

3.9 Further enhancements

More improvements were studied in order to improve the overall performance of the program. These will be explained in detail.

3.9.1 Compressor

One issue found when the noise estimation changed was the change in amplitude that it caused. One possible solution to this problem is a compressor, this reduces the dynamic range of the signals. Whenever the amplitude of the signal goes beyond a certain threshold it is reduced according to a ratio. If the threshold is set to 8dB and the ratio to 4dB, then when the signal reaches 12dB it will be reduced to 9dB. If the ratio is infinite then it will be set to the value of the threshold. Other parameters can be used for a compressor such as attack and release but, for simplicity reasons, it was kept to the two previous parameters. The implementation is shown bellow:

```
float compressor(float sample){
    // Calculate magnitude in dB of the sample
    db = 20*log10(abs(sample));
    if(db > c_threshold){ // If it exceeds the threshold
        // Reduce by a ratio
        db = (db - c_threshold)/c_ratio + c_threshold;
        if(sample < 0){ //We lost sign information, put it back
            sample = -pow(10,(db/20));
        }
        else{
            sample = pow(10,(db/20));
        }
    }
   else{
        return sample;
    }
    return sample;
}
```

3.9.2 IIR filter

The human voice is characterized by having its frequency components in the range of 300 Hz and 3000 Hz. This knowledge can tell the designer a lot of information of where pure noise is located. For this reason a 4th order elliptic IIR filter was created to remove these unwanted frequency components. In order to improve the efficiency of the filter a direct form 2 transposed filter architecture is used. The use of FIR filters introduces a computation delay that can reduce the effectiveness of the denoiser. The filter was tested for both input and output signals. The frequency response and the implementation are shown

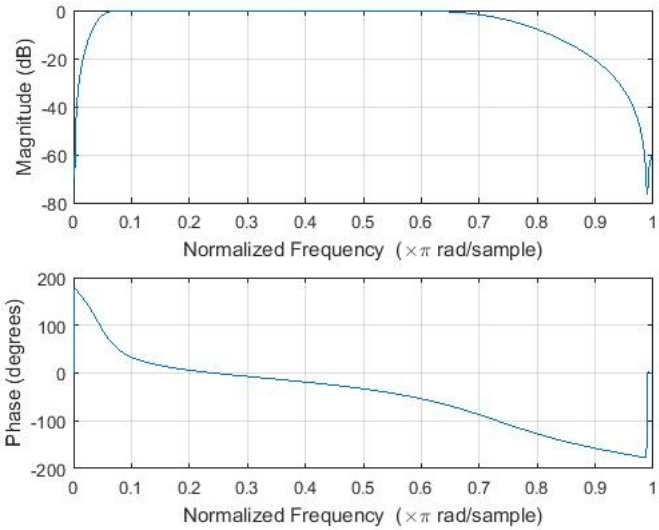


Figure 2: Elliptic filter for noise removal

below. The full MATLAB code to obtain the coefficient is shown in Appendix 6.2.

```
float filter(float sample) {
   int i;
   y = 0;
    // First iteration of difference equation
   y = x[0]+b[0]*sample;
   for(i=0; i < FILTER_ORDER - 1; i++){
       // Calculate difference equation of direct form 2 transposed
      x[i] = x[i+1] + b[i+1]*sample - a[i+1]*y;
   }
   // Last iteration
   x[FILTER_ORDER-1] = b[FILTER_ORDER]*sample - a[FILTER_ORDER]*y;
   return y;
}
```

3.9.3 Frequency domain filter

Another method of implementing a filter is by setting the magnitude bins to zero if they lie outside the speech frequency spectrum. It was observed that this method was more efficient, faster and it provided better results. The bins that fell inside the speech frequency spectrum could also be amplified by a gain constant. This filter was used both at the input and the output. At the input it eliminated low frequency noise inherent from the audio samples and at the

Parameter	Value
α	0.9
λ	0.05
$\tau_{magnitude}$	0.12
τ_{noise}	0.032
output-gain	1.8
lower-bins	140
upper-bins	252

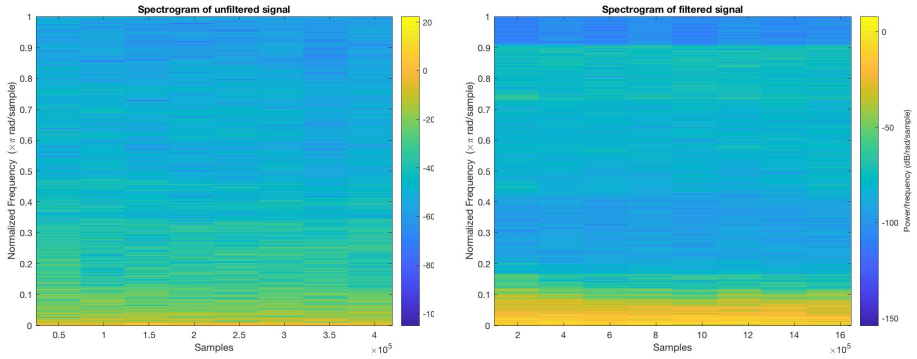
Table 1: Parameters used for the denoiser

output it eliminated some of the noise generated by the program itself. The use of the filter allowed the designer to reduce the value of the subtraction constant α and therefore reduce the musical noise artifact and increase the sharpness of the speech. The parameters used in the filter were lower-bins and upper-bins. The implementation is shown bellow:

```
void frame_filter(){
   int n;
   for(n = 0; n < FFTLEN; n++){
      cpx[n].r = cpx[n].r;
      if(n < lower_bins){ // High pass filter
         cpx[n].r = 0;
         cpx[n].i = 0;
      }
      if(n > upper_bins){ //Low pass filter
         cpx[n].r = 0;
         cpx[n].i = 0;
      }
      if(n < upper_bins && n > lower_bins){ // Amplifier
         cpx[n].r *= output_gain;
      }
   }
}
```

4 Final choices

Not all enhancements could be used. Not only because some of them excluded others but because they introduced a delay that could grow bigger than the time between frames. If this happened the signal could be corrupted. After much testing the enhancements used where number 2 and 3, a change to the noise estimation period (156 frames) and two frequency domain filters at the input and output. The values of the constants used are shown in the table bellow:



(a) Spectrogram of original signal (b) Spectrogram of filtered signal

Figure 3: Spectrogram of original and filtered signals for one audio file

α was reduced in order to increase the crispness of the speech that had been reduced in the denoising process. This could be done because much of the noise had been eliminated by the input filter. λ was chosen to be 0.1 in order to smooth the output result. $\tau_{magnitude}$ was set to a high value so that the musical noise could be reduced. The filter reduces the amplitude of the signal and for this reason bins that laid in the speech frequency spectrum were multiplied by an amplifying constant "output-gain" whose value was chosen to be 1.8. The frequency filter had an impressive performance, it was able to set the first 100 bins to zero (lower-bins) without affecting the speech. This was not the case for the low-pass filter side of the filter which was not able of reducing the high frequency components without altering the speech.

5 Conclusion

The final enhancement choice has been tested with various audio files, with different levels of noise. It has been proven that it is very challenging to produce a code that cancels noise in an effective way for all the various levels. However, the enhancements and parameters chosen have proven to be the best fit possible. It has been tested mostly empirically, through listening tests, but it can be proven also analytically by looking at the spectrograms of filtered and unfiltered signals, that are shown in figure 3. The project can be considered, hence, successful.

6 Appendix

6.1 C Code

```
/*************************************************************************************
              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"
```

```
// Filter coefficients
#include "noise_filter_coeff.txt"

// Some functions to help with writing/reading the audio ports when
    using interrupts.
#include <helper_functions_ISR.h>

#define WINCONST 0.85185      /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0  /* sample frequency, ensure this matches Config
    for AIC */
#define FFTLEN 256            /* fft length = frame length 256/8000 = 32
    ms*/
#define NFREQ (1+FFTLEN/2)    /* number of frequency bins from a real
    FFT */
#define OVERSAMP 4            /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP) /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC) /* length of I/O buffers */


#define OUTGAIN 16000.0       /* Output gain for DAC */
#define INGAIN (1.0/16000.0)  /* Input gain for ADC */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP    /* time between calculation of each
    frame */
int FILTER_ORDER = sizeof(a)/sizeof(a[0])-1;   // Find the order of the
    filter

/****************************** Global declarations
    *******************************/

/* Audio port configuration settings: these values set registers in the
    AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
       info. */
DSK6713_AIC23_Config Config = { \
         /*****************************************************************/
         /*   REGISTER           FUNCTION           SETTINGS       */
         /*****************************************************************/\
    0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
                   */\
    0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
                  */\
    0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
                   */\
    0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
                  */\
    0x0011, /* 4 ANAPATH   Analog audio path control    DAC on, Mic boost
        20dB*/\
```

```
    0x0000, /* 5 DIGPATH  Digital audio path control   All Filters off
            */\
    0x0000, /* 6 DPOWERDOWN Power down control         All Hardware on
            */\
    0x0043, /* 7 DIGIF    Digital audio interface format 16 bit
                  */\
    0x008d, /* 8 SAMPLERATE Sample rate control    8 KHZ-ensure matches
        FSAMP */\
    0x0001  /* 9 DIGACT   Digital interface activation On
                       */\
         /*************************************************************/
};

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

float *inbuffer, *outbuffer;     /* Input/output circular buffers */
float *inframe, *outframe;       /* Input and output frames */
float *inwin, *outwin;           /* Input and output windows */
float ingain, outgain;        /* ADC and DAC gains */
float cpufrac;                /* Fraction of CPU time used */
volatile int io_ptr=0;           /* Input/ouput pointer for circular
    buffers */
volatile int frame_ptr=0;        /* Frame pointer */
complex cpx[FFTLEN];       /* Complex variable for operating with the
    frames */

float *mag;           //Variable to store the magnitude of the frequency
    spectrum
float *mag_original;
float *m1;            // Buffer containing the minimum magnitude of the
    first 312 frames
float *m2;            // Buffer containing the minimum magnitude of the
    second 312 frames
float *m3;            // Buffer containing the minimum magnitude of the
    third 312 frames
float *m4;            // Buffer containing the minimum magnitude of the
    fourth 312 frames
float *p;                // Buffer to store M
float *p1;               // Buffer for the implementation of
    enhancement 1
float *p2;               // Buffer for the implementation of
    enhancement 2
float *p3;               // Buffer for the implementation of
    enhancement 3
float *x;
float *x2;
// Buffer for the implementation of enhancement 6 storing all alphas for
    every bin
float *alpha_array;
```

```
// Second value that alpha can take to supress bins with low SNR
float new_alpha;
float y;                  // Variable used to compute the filters
int mag_index;            // Frame counter
float *min_mag;           // Buffer to store the noise estimator
float lambda;             // Noise tolerance parameter
float g[FFTLEN];          // Gain factor
int switch_on;            // Switch basic noise substractor on
float alpha;              // Noise estimation scaling factor
float taus;               // Enhancement 1 and 2 parameter for the low
    pass filter
float taun;               // Enhancement 3 parameter for the low pass
    filter
float threshold_snr;      // SNR threshold used for enhancement 6
float snr;                // Variable to compute the snr in enhancement 6
float input_gain;         // Gain at the output of the input filter
float output_gain;        // Gain at the output of the output filter
float c_threshold;        // Threshold used in the compressor
float c_ratio;            // Ratio by which to attenuate the signal in
    the compressor
float db;             // Variable to store the magnitude in dB of the
    signal in the compressor
int lower_bins;           // Bellow this number all the bins are set to
    zero in the frame filter
int upper_bins;           // Above this number all the bins are set to
    zero in the frame filter

// SWITCHES TO ENHANCEMENTS

int enhancement_1;
int enhancement_2;
int enhancement_3;
int enhancement_4;
int enhancement_4_1;
int enhancement_4_2;
int enhancement_4_3;
int enhancement_4_4;
int enhancement_5;
int enhancement_6;
int filter_input;
int filter_output;
int compressor_on;
int frame_filter_input;
int frame_filter_output;



 /****************************** Function prototypes
     ******************************/
```

```
void init_hardware(void);  /* Initialize codec */
void init_HWI(void);         /* Initialize hardware interrupts */
void ISR_AIC(void);          /* Interrupt service routine for codec */
void process_frame(void);    /* Frame processing routine */
void substract_noise();      // Basic noise substractor

// Enhancement functions
void enhancement1();
void enhancement2();
void enhancement3();
void enhancement4();
void enhancement5();
void enhancement6();
float filter(float sample);
float filter2(float sample);
float compressor(float sample);
void frame_filter();


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

   int k; // used in various for loops
   mag_index = 0;
   lambda = 0.1;
   switch_on = 1;
   alpha = 0.9;
   new_alpha = 15;
   taus = 0.08;
   taun = 0.032;
   threshold_snr = 2;
   c_ratio = 4;
   c_threshold = 100;
   enhancement_1 = 0;
   enhancement_2 = 1;
   enhancement_3 = 0;
   enhancement_4 = 0;
   enhancement_4_1 = 0;
   enhancement_4_2 = 0;
   enhancement_4_3 = 0;
   enhancement_4_4 = 0;
   enhancement_5 = 0;
   enhancement_6 = 0;
   filter_input = 0;
   filter_output = 0;
   compressor_on = 0;
   frame_filter_input = 1;
   frame_filter_output = 1;
```

```
   output_gain = 1.8;
   input_gain = 2;
   lower_bins = 100;
   upper_bins = FFTLEN;


/*  Initialize and zero fill arrays */

   inbuffer = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
    outbuffer = (float *) calloc(CIRCBUF, sizeof(float)); /* Output
        array */
   inframe    = (float *) calloc(FFTLEN, sizeof(float)); /* Array for
       processing*/
    outframe  = (float *) calloc(FFTLEN, sizeof(float)); /* Array for
        processing*/
    inwin     = (float *) calloc(FFTLEN, sizeof(float)); /* Input window
        */
    outwin    = (float *) calloc(FFTLEN, sizeof(float)); /* Output
        window */
    mag       = (float *) calloc(FFTLEN, sizeof(float));
    mag_original= (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));
    min_mag   = (float *) calloc(FFTLEN, sizeof(float));
    p       = (float *) calloc(FFTLEN, sizeof(float));
    p1        = (float *) calloc(FFTLEN, sizeof(float));
    p2        = (float *) calloc(FFTLEN, sizeof(float));
    p3        = (float *) calloc(FFTLEN, sizeof(float));
    x       = (float *) calloc(FILTER_ORDER, sizeof(float));
    x2        = (float *) calloc(FILTER_ORDER, sizeof(float));

   // Initialize ms and alpha array
   for(k=0; k < FFTLEN; k++){
      m1[k] = 10000000;     // Set m1 to a very large value
      alpha_array[k] = alpha; //Initialize alpha_array to alpha
      x2[k] = 0;
   }

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

   /* initialize hardware interrupts */
   init_HWI();

/* initialize algorithm constants */

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

```
   inwin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
   outwin[k] = inwin[k];
   }
   ingain=INGAIN;
   outgain=OUTGAIN;


   /* main loop, wait for interrupt */
   while(1)   process_frame();
}

/********************************* init_hardware()
    *******************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

   /* Function below sets the number of bits in word used by MSBSP
       (serial port) for
   receives from AIC23 (audio port). We are using a 32 bit packet
       containing two
   16 bit numbers hence 32BIT is set for receive */
   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

   /* Configures interrupt to activate on each consecutive available 32
       bits
   from Audio port hence an interrupt is generated for each L & R sample
       pair */
   MCBSP_FSETS(SPCR1, RINTM, FRM);

   /* These commands do the same thing as above but applied to data
       transfers to the
   audio port */
   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
   MCBSP_FSETS(SPCR1, XINTM, FRM);


}
/********************************* init_HWI()
    *************************************/
void init_HWI(void)
{
   IRQ_globalDisable();     // Globally disables interrupts
   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the
       debugger)
   IRQ_map(IRQ_EVT_RINT1,4);  // Maps an event to a physical interrupt
```

```
   IRQ_enable(IRQ_EVT_RINT1);  // Enables the event
   IRQ_globalEnable();        // Globally enables interrupts

}

/****************************** process_frame()
    ***********************************/
void process_frame(void)
{
   int k, m;
   int io_ptr0;

   /* work out fraction of available CPU time used by algorithm */
   cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;

   /* wait until io_ptr is at the start of the current frame */
   while((io_ptr/FRAMEINC) != frame_ptr);

   /* then increment the framecount (wrapping if required) */
   if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;

   /* save a pointer to the position in the I/O buffers
       (inbuffer/outbuffer) where the
   data should be read (inbuffer) and saved (outbuffer) for the purpose
       of processing */
   io_ptr0=frame_ptr * FRAMEINC;

   /* copy input data from inbuffer into inframe (starting from the
       pointer position) */

   m=io_ptr0;
    for (k=0;k<FFTLEN;k++)
   {
      inframe[k] = inbuffer[m] * inwin[k];
      if (++m >= CIRCBUF) m=0; /* wrap if required */
   }

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


   /* please add your code, at the moment the code simply copies the
       input to the
   ouptut with no processing */

   //Making the input complex
   for(k=0; k < FFTLEN; k++){
      cpx[k].r = inframe[k];
      cpx[k].i = 0;
   }
```

```
fft(FFTLEN,cpx);

if(frame_filter_input){
   frame_filter();
}

//average_mag = 0;

// Calculate the magnitude of the frame
// Update frame by frame the mag variable until it is full at 156
     (156 frames)

for(k = 0; k < FFTLEN; k++){
   mag[k] = cabs(cpx[k]);

   // Use an unmodifiable version of mag for its use in different
       enhancements
   mag_original[k] = mag[k];
}

// Enhancement 1

enhancement1();

// Enhancement 2

enhancement2();


// Minimize M1

for(k = 0; k < FFTLEN; k++){
   if(m1[k] > mag[k]){
      m1[k] = mag[k];
   }
}

mag_index++;

//Buffer rotation
p = m4;
m4 = m3;
m3 = m2;
m2 = m1;
m1 = p;

for(k=0; k < FFTLEN; k++){
   m1[k] = OUTGAIN;
   alpha_array[k] = alpha;
```

```
}

if(mag_index == 156){
   // Initialize the counter
   mag_index = 0;

   // Calculate which is smaller: M1, M2, M3 or M4?
   k = 0;
   while(k < FFTLEN){
      min_mag[k] = m1[k];
      if(min_mag[k] > m2[k]){
         min_mag[k] = m2[k];
      }
      if(min_mag[k] > m3[k]){
         min_mag[k] = m3[k];
      }
      if(min_mag[k] > m4[k]){
         min_mag[k] = m4[k];
      }
      k++;
   }
}

// Enhancement 6

if (enhancement_6) {
   enhancement6();
}

// Enhancement 3

enhancement3();

// Substract the noise from the signal if desired

if(switch_on && !enhancement_4 && !enhancement_5){
   substract_noise();
}

if(enhancement_4 && !enhancement_5 && !switch_on){
   enhancement4();
}

if(enhancement_5 && !enhancement_4 && !switch_on){
   enhancement5();
}

if(frame_filter_output){
   frame_filter();
}
```

```
   ifft(FFTLEN,cpx);

   for(k=0; k < FFTLEN; k++){
      outframe[k] = cpx[k].r;
   }

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

    /* multiply outframe by output window and overlap-add into output
        buffer */

   m=io_ptr0;

    for (k=0;k<(FFTLEN-FRAMEINC);k++)
   {                              /* this loop adds into outbuffer */
      outbuffer[m] = outbuffer[m]+outframe[k]*outwin[k];
      if (++m >= CIRCBUF) m=0; /* wrap if required */
   }
    for (;k<FFTLEN;k++)
   {
      outbuffer[m] = outframe[k]*outwin[k]; /* this loop over-writes
          outbuffer */
       m++;
   }
}
/************************** INTERRUPT SERVICE ROUTINE
    ****************************/

// Map this to the appropriate interrupt in the CDB file

void ISR_AIC(void)
{
   short sample;
   float out_sample;
   /* Read and write the ADC and DAC using inbuffer and outbuffer */

   sample = mono_read_16Bit();
   if (filter_input) {
      y = filter((float)sample); // Call function to implement
          difference equation
      inbuffer[io_ptr] = y*ingain*input_gain;
   }
   else {
      inbuffer[io_ptr] = ((float)sample)*ingain;
   }
   /* write new output data */
   if(filter_output || compressor_on){
      if(filter_output){
         out_sample = filter2(outbuffer[io_ptr]);
```

```
      }
      if(compressor_on){
         out_sample = (compressor(outbuffer[io_ptr]));
      }
      mono_write_16Bit((int)out_sample*outgain*output_gain);
   }
   else{
      mono_write_16Bit((int)(outbuffer[io_ptr] * outgain));
   }

   /* update io_ptr and check for buffer wraparound */

   if (++io_ptr >= CIRCBUF) io_ptr = 0;
}

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

void enhancement1(){
   float ex;
   int n;
   ex = pow(2.71828, (-0.25/taus)); //Calculate exponential
   p1[0] = (1 - ex) * mag[0];       // First iteration of the equation
   for(n = 1; n < FFTLEN; n++){
      p1[n] = (1 - ex) * mag[n] + ex * p1[n-1]; //Difference equation
   }
   if(enhancement_1 && !enhancement_2 ){ // Enhancement 1 must be on
      for(n = 0; n < FFTLEN; n++){
         // New value of magnitude is the low pass filtered version
         mag[n] = p1[n];
      }
   }
}

void enhancement2(){
   float ex;
   int n;
   ex = pow(2.71828, (-0.25/taus)); // Calculate the exponential
   p2[0] = sqrt((1 - ex) * mag[0]*mag[0]); // First iteration of the
       equation
   for(n = 1; n < FFTLEN; n++){
      p2[n] = sqrt((1 - ex) * mag[n]*mag[n] + ex * p2[n-1]*px[n-1]);
   }
   if(enhancement_2){ // Implement the enhancement if desired
      for(n = 0; n < FFTLEN; n++){
         mag[n] = p2[n];
      }
   }
}
 void enhancement3(){
   float ex;
```

```
   int n;
   ex = pow(2.71828, (-0.25/taun)); // Compute the exponential
   p3[0] = (1 - ex) * min_mag[0];  //First iteration of difference
       function
   for(n = 1; n < FFTLEN; n++){
       // Calculate the difference equation
      p3[n] = (1 - ex) * min_mag[n] + ex * p3[n-1];
   }
   if(enhancement_3){ // Implement the enhancement if desired
      for(n = 0; n < FFTLEN; n++){
         min_mag[n] = p3[n]; //Assign to the noise estimator its new
             value
      }
   }
}
void substract_noise(){
   int k;
   for(k = 0; k < FFTLEN; k++){
       // Second term of the g[k] equation
      g[k] = (1 - alpha_array[k]*(min_mag[k]/mag[k]));
      if(lambda > g[k]){ // Find maximum
         g[k] = lambda; // First term of g[k] equation
      }
      cpx[k].r = cpx[k].r * g[k];
      cpx[k].i = cpx[k].i * g[k];
   }
}
void enhancement4(){
   int k;
   if (enhancement_4_1){ //Implement the enhancement if desired
      for(k = 0; k < FFTLEN; k++){
       // Multiply noise estimator by scaling factor
         min_mag[k] = alpha_array[k]*min_mag[k];
          // Second term of g[k] function
         g[k] = (1 - (min_mag[k]/mag[k]));
         if((lambda*(min_mag[k]/mag[k])) > g[k]){ //Find maximum
         //First term of g[k] function
            g[k] = lambda*(min_mag[k]/mag[k]);
         }
         cpx[k].r = cpx[k].r * g[k]; //Multiply signal by gain factor
         cpx[k].i = cpx[k].i * g[k];
      }
   }
   else if (enhancement_4_2){
      for(k = 0; k < FFTLEN; k++){
         min_mag[k] = alpha_array[k]*min_mag[k];
         g[k] = (1 - (min_mag[k]/mag[k]));
         if((lambda*(p1[k]/mag[k])) > g[k]){
            g[k] = lambda*(p1[k]/mag[k]);
         }
```

```
            cpx[k].r = cpx[k].r * g[k];
            cpx[k].i = cpx[k].i * g[k];
        }
    }
    else if(enhancement_4_3){
        for(k = 0; k < FFTLEN; k++){
            min_mag[k] = alpha_array[k]*min_mag[k];
            g[k] = (1 - (min_mag[k]/p1[k]));
            if((lambda*(min_mag[k]/p1[k])) > g[k]){
                g[k] = lambda*(min_mag[k]/p1[k]);
            }
            cpx[k].r = cpx[k].r * g[k];
            cpx[k].i = cpx[k].i * g[k];
        }
    }
    else if(enhancement_4_4){
        for(k = 0; k < FFTLEN; k++){
            min_mag[k] = alpha_array[k]*min_mag[k];
            g[k] = (1 - (min_mag[k]/p1[k]));
            if(lambda > g[k]){
                g[k] = lambda;
            }
            cpx[k].r = cpx[k].r * g[k];
            cpx[k].i = cpx[k].i * g[k];
        }
    }
}
void enhancement5(){
    int k;
    for(k = 0; k < FFTLEN; k++){
        min_mag[k] = alpha_array[k]*min_mag[k]; //Use the scaling factor
            alpha
        g[k] = sqrt(1 - ((min_mag[k]*min_mag[k])/(mag[k]*mag[k])));

        if(lambda > g[k]){
            g[k] = lambda;
        }
        //Subtract noise from the signal
        cpx[k].r = cpx[k].r * g[k];
        cpx[k].i = cpx[k].i * g[k];
    }
}

float filter(float sample) {
    int i;
    y = 0;
     // First iteration of difference equation
    y = x[0]+b[0]*sample;
    for(i=0; i < FILTER_ORDER - 1; i++){
        // Calculate difference equation of direct form 2 transposed
```

```
      x[i] = x[i+1] + b[i+1]*sample - a[i+1]*y;
   }
   // Last iteration
   x[FILTER_ORDER-1] = b[FILTER_ORDER]*sample - a[FILTER_ORDER]*y;
   return y;
}
float filter2(float sample){
   int i;
   y = 0;

   y = x2[0]+b[0]*sample;

   for(i=0; i < FILTER_ORDER - 1; i++){
      x2[i] = x2[i+1] + b[i+1]*sample - a[i+1]*y;
   }
   x2[FILTER_ORDER-1] = b[FILTER_ORDER]*sample - a[FILTER_ORDER]*y;

   return y;
}
void enhancement6() {
   int k;
   for (k = 0; k < FFTLEN; k++) {
      // Initialize alpha_array
      alpha_array[k] = alpha;
      snr = mag_original[k] / min_mag[k]; //Calculate the SNR
      if (snr < threshold_snr) { //If SNR is too bad
         alpha_array[k] = new_alpha; //Assign new alpha to that bin
      }
   }
}
float compressor(float sample){
    // Calculate magnitude in dB of the sample
    db = 20*log10(abs(sample));
    if(db > c_threshold){ // If it exceeds the threshold
        // Reduce by a ratio
        db = (db - c_threshold)/c_ratio + c_threshold;
        if(sample < 0){ //We lost sign information, put it back
            sample = -pow(10,(db/20));
        }
        else{
            sample = pow(10,(db/20));
        }
    }
   else{
        return sample;
    }
    return sample;
}
void frame_filter(){
   int n;
```

```
   for(n = 0; n < FFTLEN; n++){
      cpx[n].r = cpx[n].r;
      if(n < lower_bins){ // High pass filter
         cpx[n].r = 0;
         cpx[n].i = 0;
      }
      if(n > upper_bins){ //Low pass filter
         cpx[n].r = 0;
         cpx[n].i = 0;
      }
      if(n < upper_bins && n > lower_bins){ // Amplifier
         cpx[n].r *= output_gain;
      }
   }
}
```

6.2 MATLAB code for IIR Filter - Enhancement 9.2

```
%Definition of filter specifications
n = 4;
fs = 8000;
wp = [300 3000]/fs/2;
Rp = 0.3;
Rs = 20;

%creation and lot of the filter itself
[b,a] = ellip(n/2, Rp, Rs, wp, 'bandpass')
figure
freqz(b,a)

%rappresentation on the z-plane of poles and zeros
figure
zplane(b,a)

%obtaining coefficients
%csvwrite("num_coeff.txt", b);
%csvwrite("denom_coeff.txt", a);

fileID = fopen("noise_filter_coeff.txt");
fprintf(fileID, 'double a[]={');
fprintf(fileID, '%d, \r\n', a);
fprintf(fileID, '};');

fprintf(fileID, 'double b[]={');
fprintf(fileID, '%d, \r\n', b);
fprintf(fileID, '};');
fclose(fileID);
```