RTDSP Project – Speech Enhancement

Lizhang Lin (00840705) and Yihan Qi (00813873)

Contents

1. Introduction 2
2. Basic Implementation 2
2.1. Input FFT 3
2.2. Estimate Noise Spectrum 3
2.2.1. Noise Buffer 3
2.2.2. Noise Spectrum from Noise Buffers 4
2.3. Subtract Noise Spectrum 4
2.4. Choice of overestimation factor α and floor parameter λ 5
2.5. IFFT and Overlap Add 5
3. Important Feature of the Program 5
4. Enhancements 6
Enhancement One 6
Enhancement Two 6
Enhancement Three 6
Enhancement Four 6
Enhancement Five 7
Enhancement Six 7
Enhancement Seven 8
Enhancement Eight 8
Enhancement Nine 9
5. Additional Enhancements 9
Additional Enhancement One 9
Additional Enhancement Two (VAD) 10
6. Results 11
6.1. Enhancements 2 & 4C 11
6.2. Additional Enhancement One Plus Enhancements 2 & 4C 12
6.3. Final Results - Additional Enhancement Two Plus Previous Enhancements 12
7. Reference 13
8. CODE 13
8.1. Matlab Script: 13
8.2. Final Code: 14

1. Introduction

In modern telecommunications, telephones are increasingly being used and consumer demands are pushing the innovations of noise free communication even in noisy environments, such as streets, factories and airports. However, it's very complicated to implement these noise free systems because engineers may never fully know what sort of noise environment the users are located, and thus difficult to precisely differentiate their speech from noisy background. Therefore, a general situation of voice enhancement is worthy to be discussed in our final RTDSP report.

In this report, we aimed to implement a basic speech enhancer by removing the effect of noise, and then elaborating the algorithm to overcome the advent of pitch distortion as well as 'musical' noise. The method we applied is called frame processing since we needed to process a whole frame of data and do not take another set of data for processing until we have done with the previous frame.

We firstly implemented a basic design, which did little contribution to our goal, and then explored 8 enhancements along with analysis of working principle and performance measurement. Some of those enhancements significantly improved our design while others made the design worse, we thus came up with a best combination of enhancements based on the analysis and reference and provided additional optimisations.

2. Basic Implementation

Before going into detailed explanations about the basic frame processing function, there is something we must understand about the entire programme. The programme goes into a forever-running ***while*** loop in the ***main*** function, which is called function ***process_frame*** as shown in the code below.

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

In the meanwhile, the programme is sampled every $\frac{1}{8000}$ second when it reads data from the DSK board into an input circular buffer, writes the data back to the board from an output circular buffer, and updates input/output buffer pointer for the circular buffers as shown in the code below.

```
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;
}
```

In terms of the ***process_frame*** function, it waits until the I/O buffer pointer is at the start of the current frame, and then copies input data from input circular buffer '*Inbuffer*' into the input frame *'inframe'* as shown in the code next page. Note that the data from the input circular buffer is multiplied with an input window.

```
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 */
  }
```

The above code is followed by the main part of the basic function of frame processing. The implementation of the basic function can be described by figure 1. Firstly, we performed an FFT on the input signal frame and then estimated the noise spectrum which is subtracted from the input spectrum. An inverse FFT will be employed once the frequency domain processing has been completed, and the output then to be written back to output frame. Detailed explanations for the basic function can be found below.

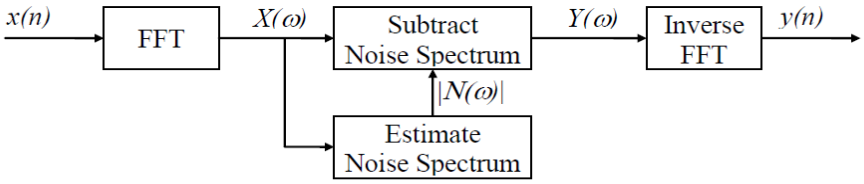

Figure 1. Basic programme functionality.

2.1. Input FFT

A provided header file called '*#include fft_functions.h*' in the skeleton code contains FFT and IFFT functions. In order to perform FFT, we declared an array '*inframe_complex*' of complex data type and saved all the data from the input frame.

```
 for(k=0; k<FFTLEN; k++){
     inframe_complex[k] = cmplx(inframe[k], 0);
  }
fft(FFTLEN, inframe_complex);
```

2.2. Estimate Noise Spectrum

2.2.1. Noise Buffer

Normally, a speaker will pause for a moment every few seconds, which allows some time where only noise is being measured by our program. In this case, we assumed that moment appears at least once every 10 seconds. Therefore, noise could be estimated by finding the minimum spectra over the 10 seconds period (as shown in figure 2, split into four buffers, storing the minimum spectra over the past periods).

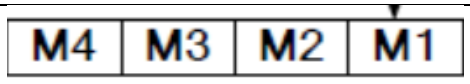

Figure 2. Noise Buffers. M1, M2, M3 and M4 store the minimum spectra occur in the four intervals current to past 2.5 seconds, past 2.5 to 5 seconds, past 5 to 7.5 seconds, and past 7.5 to 10 seconds respectively.

The buffer M1 is continuously updated by the minimum spectrum from the comparison between the most recent frame and the record in the buffer, and after every 2.5 seconds (312 frames) buffers M2, M3 and M4

will be updated as shown in the code below. Note that this will inevitably introduce a delay of noise estimation since all four buffers must be filled (and so after 10 seconds) before the correct minimum spectrum is loaded into the noise estimate array.

```
if(++noise_count >= noise_count_loop){ //wrap after 2.5 seconds, in this case, noise_count_loop = 312.
                                       // noise_count_loop here is calculated by 2.5/(FFTLEN/(OVERSAMPLING*FSAMP))
   noise_count = 0;
   temp = M4;
   M4= M3;
   M3= M2;
   M2= M1;//buffer shifting
   M1 = temp;

   for(k=0;k<NFREQ;k++){
      M1[k] = mag[k];     //at the beginning of buffer shifting, assign input data directly into the noise buffer M1
   }
}
else{
   for(k=0;k<NFREQ;k++){
      if(M1[k]>mag[k]){
         M1[k] = mag[k]; //update with minimum spectrum
      }
   }
}
```

The code above works by assigning the 4 M_i buffers as ***float*** pointers to avoid unnecessary copying (by simply switching the pointers rather than copying each element into each other – another method which would also worked would have been to implement a two-dimensional array [4][FFTLEN] and setup the first index in a way to act circular).

Note that we used NFREQ instead of FFTLEN because of the symmetry property of the FFT spectrum. By doing so, we halved the number of computations. Note that before converting back to time domain the remaining of the samples were copied back in a symmetric manner. This will be demonstrated in section 2.5.

2.2.2. Noise Spectrum from Noise Buffers

The noise estimation is derived from the minimum of these 4 buffers. However, by doing so the minimum noise returned is underestimated and thus we introduced an overestimating factor α (known as ***'alpha'*** in the code) in order to give a more accurate estimation of noise spectrum (shown in the code below). In this case, we chose α to be 20.

```
   for(k=0;k<NFREQ;k++){
      N[k] = alpha* min(min(M1[k],M2[k]),min(M3[k],M4[k]));
   }
```

2.3. Subtract Noise Spectrum

From figure 1, we could see that the processed signal $Y(\omega)$ can be mathematically described as $Y(\omega) = X(\omega) - N(\omega)$ Due to the fact that we don't know the phase of the noise signal, we only interested in subtracting the magnitudes and hence we can rearrange the equation below

$Y(\omega) = X(\omega) - \frac{|X(\omega)|-|N(\omega)|}{|X(\omega)|} = X(\omega) \times (1 - \frac{|N(\omega)|}{|X(\omega)|}) = X(\omega) \times g(\omega)$, where $g(\omega)$ is regarded as a frequency-dependent gain factor.

However, there is another problem that $g(\omega)$ may become negative from time to time. We thus set a floor parameter λ (known as ***'lambda'*** in the code) for the value of $g(\omega)$ using the equation $g(\omega) = \max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$. Note that λ is normally set in a range between 0.01 and 0.1 as given in project specification such that the output suffers less from musical noise [1]. In the section below, we will further discuss how λ can be varied along with the overestimation factor α in order to minimise background and musical noise.

The code snippet shown below demonstrates the basic implementation of $Y(\omega) = X(\omega) \times g(\omega)$ within the program, with ***'rmul'*** defined as complex number multiplication.

```
//Y(w)  = X(w)*max(lambda,g(w))
inframe_complex[k] = rmul(max(lambda, 1- N[k]/X[k]), inframe_complex[k]); //complex multiplication
```

2.4. Choice of overestimation factor α and floor parameter λ

Various α and λ values were then tested by the basic implementation of the code to determine how well the basic implementation could remove the noise. We noticed that overestimation factor α had noticeable impact on the background white noise in the range of 10 to 30, however, when it was greater than 35 the speech was distorted with much musical noise mixed with background noise. On the other hand, we discover that increasing λ could remove a large amount of the musical noise but reintroduced some background white noise which was undesirable. Thus, we should choose the correct value of α and λ such that a fair trade-off between musical noise and background white noise can be obtained. Moreover, we found that different values of α and λ suits different levels of background noise (in this experiment, we had several test files at different level of background noise), therefore, a trade-off of α=20 and λ=0.05 was originally chosen for the basic implementation such that for the majority of the audio files most of the background noise was removed while only a small amount of musical noise was introduced. However, in this case a large amount of background noise still exist in the audio file named 'phantom 4'.

2.5. IFFT and Overlap Add

After $Y(\omega)$ has been calculated by the above method, IFFT will be taken to produce the output signal to the audio port $y(n)$. However, as we mentioned previously we only processed half of the spectrum, therefore, before taking IFFT the other half of the spectrum is copied as shown in the code below.

```
//bins 129 - 255 are complex conjugates of bins 127 to 1.
  // bins 0 and 128 are independent.
    for(k=NFREQ;k<FFTLEN;k++){
      inframe_complex[k] = conjg(inframe_complex[FFTLEN-k]);//Y(w)=
    }
```

After that, IFFT is performed, and the real part of the complex arrary ***'inframe_complex'*** is copied to the output frame as shown in the code below.

```
  /*************************inverse FFT of the processed frames*************************/
  ifft(FFTLEN, inframe_complex);
  /*************************output*************************/
  for (k=0;k<FFTLEN;k++)
  {
    outframe[k] = inframe_complex[k].r;/* copy input straight into output */
  }
```

Then the array ***outframe*** that stores the processed frame is multiplied with output window and overlap-add into output buffer such that the output signal will be undistorted by the frame processing (given by template code). Note that in this case, each frame starts quarter a frame later than the previous one giving an *oversampling* ratio of 4 instead of 2. This better avoids distortion from abrupt change of the gain of a particular frequency bin.

3. Important Feature of the Program

Before going into enhancement, we would like to explain some important features of our code.

We have mentioned the method of applying pointers for the noise buffers and using the symmetry of the FFT spectrum to halve the number of computations, and thus to reduce the processing time.

4. Enhancements

In this section, we implemented all the enhancements given in the project description, and chose enhancement 2 and 4c in the end.

Enhancement One

From the estimation algorithm we mentioned before, noise is approximated with the equation $M_1(\omega) = \min(|X(\omega)|, M_1(\omega))$ and $M_1(\omega)$ only contains minimum spectrum components of noise. In fact, we could remove some high frequency noise as soon as we received input sound by implementing a low pass IIR and it is the basic idea in first enhancement. It operates on successive frames not on the speech samples themselves in the form of $P_t(\omega) = (1-k) \times |X(\omega)| + k \times P_{t-1}(\omega)\ where\ k = e^{-\frac{FRAMELENGTH}{\tau}}$. A time constant of 0.852 was chosen by using a time constant of $\tau = 50ms$ giving $e^{-\frac{8ms}{50ms}} \cong 0.85214$. Since the noise has already been removed by low pass filter before the noise estimation, the alpha was able to be reduced from 20 to around 2. The code implementation is shown below.

```
if(enhancement1==1){   // enhancement 1
        LP_mag[k] = (1-K)*(mag[k])+K*LP_mag_prev;
        LP_mag_prev = LP_mag[k];
    }
```

Enhancement Two

The enhancement is very similar to the first one with implementing a same type of low pass filter. However, we change the filter not in manipulating amplitude magnitudes but in manipulating power magnitudes. Thus, the filter equation becomes $P_t^2(\omega) = (1-k) \times |X(\omega)|^2 + k \times P_{t-1}^2(\omega)$. In later testing, we found this enhancement is more effective than first one since our ears are more sensitive in power difference. A more mathematical explaining approach is that squaring all magnitudes leads that larger values in magnitude have a larger weighting since the difference in magnitude is squared as well. So the noise estimation will be more responsive to amplitude increases and perform better estimating than enhancement 1.

```
else if(enhancement2==1){   // enhancement 2
        LP_mag[k] = sqrt((1-K)*(mag[k]*mag[k])+K*LP_mag_prev*LP_mag_prev);
        LP_mag_prev = LP_mag[k];
    }
```

Enhancement Three

According to our minimization cycle buffer design, there is an unavoidable discontinuities in each rotation. Following the ideas of previous two enhancements, a moving average signal could also be implemented in this enhancement in order to avoid these estimation discontinuities. The code below shows the implementation and how it is based on the same equation used in enhancement 1 & 2. The test result indicates a hardly noticeable improvement when it was added onto enhancement 2. This is because it only works differently in the situations where the noise is varying largely. In our testing file, however, the background noise is not varying very much throughout.

```
        if(enhancement3==1){   // enhancement 3 - this will only have a noticeable effect if the noise level is very variable
        /* low pass filter the overestimated noise */
        //N_prev = 0; // maybe delete
        N[k] = (1-K)*alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k])) + K*N_prev;
        N_prev = N[k];
        }
```

Enhancement Four

In basic implementation, we introduced a way to choose value of our gain factor $g(\omega)$ within the equation $g(\omega) = \max(\lambda, 1 - \frac{N(\omega)}{X(\omega)})$ where λ is normally set as a constant in a range between 0.01 and 0.1 so that

output suffers less from musical noise. In this enhancement, we adjusted 'lambda' λ along with the signal and noise variation and some designs involved using average power signal $P(\omega)$ instead of $X(\omega)$.

The table below shows the different enhancements and their effect upon the audio output, which was tested with enhancement 2 to see whether it improved the most advanced code available.

Enhancement Number	Gain factor equation	Performance based on Enhancement Two
Enhancement 4a	$g(\omega) = \max(\lambda \frac{N(\omega)}{X(\omega)}, 1 - \frac{N(\omega)}{X(\omega)})$	Small change and a bit of distortion
Enhancement 4b	$g(\omega) = \max(\lambda \frac{P(\omega)}{X(\omega)}, 1 - \frac{N(\omega)}{X(\omega)})$	Merely no change
Enhancement 4c	$g(\omega) = \max(\lambda \frac{N(\omega)}{P(\omega)}, 1 - \frac{N(\omega)}{P(\omega)})$	Removes some musical noise and background noise
Enhancement 4d	$g(\omega) = \max(\lambda, 1 - \frac{P(\omega)}{X(\omega)})$	Merely no change

The code below shows the implementation of each optimisation.

```
/* noise subtraction using gain factor */
  for(k=0;k<NFREQ;k++){
    if(enhancement4a == 1){
      // GainFactor[k] = max(lambda, 1 - (N[k]/mag[k])) where lamba is lower floor
      // then multiply noise by gain factor (complex multiplication using rmul function in complex.h)
      inframe_complex[k] = rmul(max(lambda*(N[k]/mag[k]), 1 - (N[k]/mag[k])), inframe_complex[k]);
    }
    else if(enhancement4b == 1){
      inframe_complex[k] = rmul(max(lambda*(LP_mag[k]/mag[k]), 1 - (N[k]/mag[k])), inframe_complex[k]);
    }
    else if(enhancement4c == 1){
      inframe_complex[k] = rmul(max(lambda*(N[k]/LP_mag[k]), 1 - (N[k]/LP_mag[k])), inframe_complex[k]);
    }
    else if(enhancement4d == 1){
      inframe_complex[k] = rmul(max(lambda, 1 - (N[k]/LP_mag[k])), inframe_complex[k]);
    }
    else{   // basic implementation
      inframe_complex[k] = rmul(max(lambda, 1 - (N[k]/mag[k])), inframe_complex[k]);
    }
  }
```

Enhancement Five

Following along the idea used in enhancement 4, we acquired gain factor in the power domain instead of the amplitude domain, by giving the equation $(\omega) = \max(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}})$. The code is shown below and it was implemented in the location adjacent to enhancement 4s' functions. The test results indicated more echoes and more additional musical noise. Therefore, it was worse than enhancement 4c and it would not be used in final program.

```
else if(enhancement5 == 1){
      inframe_complex[k] = rmul(max(lambda, sqrt(1 - (N[k]/mag[k])*(N[k]/mag[k]))), inframe_complex[k]);
    }
```

Enhancement Six

So far, the value of noise level constant, alpha, was set constant along with all frequencies. The idea of enhancement 6 is to change alpha value along with frequency variation. It works by over estimating noise at very low and very high frequency bins for the reason that human cannot produce such high or low frequency voices, which means those outlier sound must be noise. Those outlier sounds were removed by implementing a band pass filter.

By implementing this, variable gamma was introduced and it was defined larger than one in order to overestimate the noise in high or low frequency pins. The code below shows the basic implementation of enhancement 6 and further enhancement, in which the gamma is no longer a constant variable, will be discussed in later section.

```
else if(enhancement6==1){  // enhancement 6 - deliberately overestimate the noise level
    /*reduce the musical noise artifactes that are introduced by spectral subtraction*/
    if(k<enhancement6_lower_bound || k>enhancement6_higher_bound){
      N[k] = gamma*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
    }
```

Enhancement Seven

In this enhancement we tried to optimize the FFT length which is used for overlap add algorithm to achieve best program performance. A small FFT length leads a lower sounding volume with some musical noise added. This is because that the longer FFT length could provide more samples for a specified frequency range, in our program is 0 to 8000 Hz. In other words, more samples stand for a higher transmitting resolution and better hearing experience. Therefore, it is clearly to see a higher FFT length is more desired.

However, there is a trade-off of FFT length due to the limitation of our program running speed. To be noticed, our code process over a half range over frequency bins in frequency domain (FFTLEN) and we deliberately ran all the program over a new length NFREQ, NFREQ = (1+FFTLEN/2). This is because we realized the symmetrical properties in converting a real time signal into frequency time. So after FFT conversion, we only manipulated half-length of FFT and copy the left half to right before IFFT conversion. Finally, the ideal FFT length was sought to be 256 to achieve the best compromise for real time processing speed and frequency resolution. The defining code is shown below:

```
#define FFTLEN 512            /* fft length = frame length 256/8000 = 32 ms*/
```

After double the FFTLEN from 256 to 512, we found a noticeable improvement especially for the speech articulation and reduction in musical noise.

Enhancement Eight

In enhancement 8, we focused on how to reduce the noise residue throughout the speech. Noise residual is the uncorrelated noise exhibiting itself in the spectrum as randomly spaced narrow bands of magnitude spikes. This noise residual will have a magnitude between zero and a maximum value measured during nonspeech activity. The noise residual can be reduced by taking advantage of its frame-to-frame randomness. Specifically, at a given frequency bin, since the noise residual will randomly fluctuate in amplitude at each analysis frame, it can be suppressed by replacing its current value with its minimum value chosen from the adjacent analysis frames. [2]

So a function was used to detect residue if $\frac{N(\omega)}{X(\omega)} > 'residue'$ and we adjusted the $residue$ value to remove as much as musical noise as possible. In order to choose the adjacent frames to replace the current value, we found minimum value among adjacent three frames and maintained the output always as a delay signal to any avoid discontinuities. A set signal was used to indicate whether the residue reduction function is triggered on. It is running along with the looping by representing the comparison result between if $\frac{N(\omega)}{X(\omega)}$ and $residue$. The implementation code is shown below.

```
if(enhancement8 == 1){ // enhancement 8
  for(k=0;k<NFREQ;k++){
    if(set_on == 1){
      //find adjacent min
      Delay[k]= minComplex(minComplex(inframe_complex[k-1],inframe_complex[k-2]),inframe_complex[k]);
      //deactivate
      set_on = 0;
      if(N[k]/mag[k]>residue)set_on = 1;
```

```
            else if(N[k]/mag[k]>residue){
              //activate
              set_on = 1;
              Delay[k] = inframe_complex[k-1];
            }
            else{//do nothing but replace the frame
              Delay[k] = inframe_complex[k-1];
            }
          }
        }
      }
```

Enhancement Nine

By default setting and assumption, a single noise buffer takes 2.5 seconds to record sound since we assume the speaker takes a pulse for every 10 seconds. Actually, each period could be shorted in order to make system respond more quickly to update noise level more promptly. In our exploration, we found that if buffer period took 0.5 second, the program achieved best performance and even shorter period would take distortion into speech. The spectrum graph shown below is the clean file, and from which we found that the speaker stops speaking within every 2 sections.

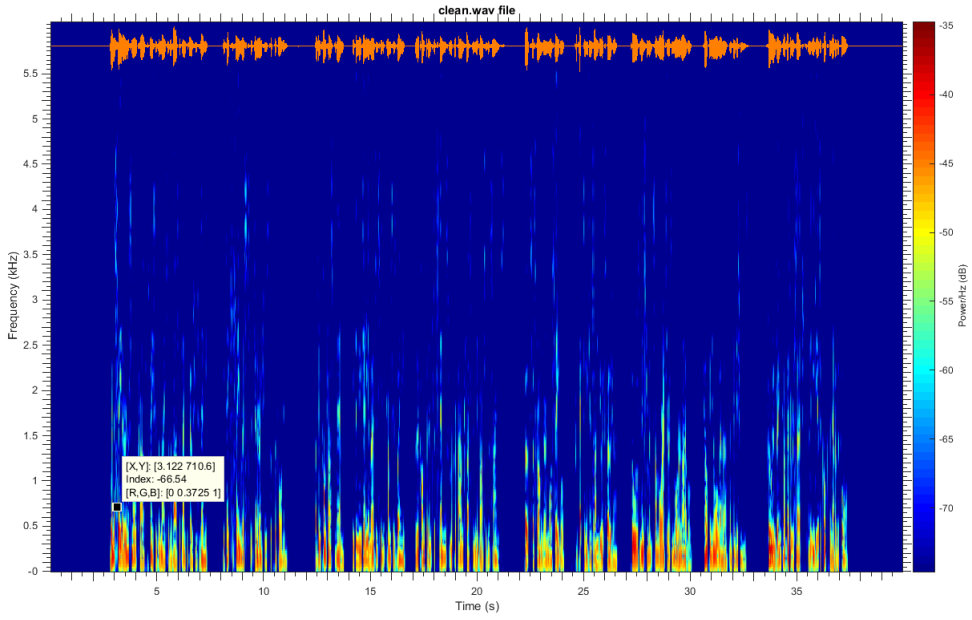

Figure 3: Spectrum of clean wave

5. Additional Enhancements

In this section, we worked out additional enhancements (Our code only performed enhancement 2, 4c and 6 for the listening test without any additional enhancements), and thus added them into our code and tested all the audio files given. Though these tests were carried out after the listening test, you can still look at spectrogram and see how much we enhanced the speech.

Additional Enhancement One

Since our previous version of code worked well on most of the audio files given except “phantom4.wav” where musical noise arose repeatedly. Theoretically, musical noise occurs after processing when isolated peaks are left alone in a spectrum. Figure 4 in the next page should be a nice illustration of musical noise.

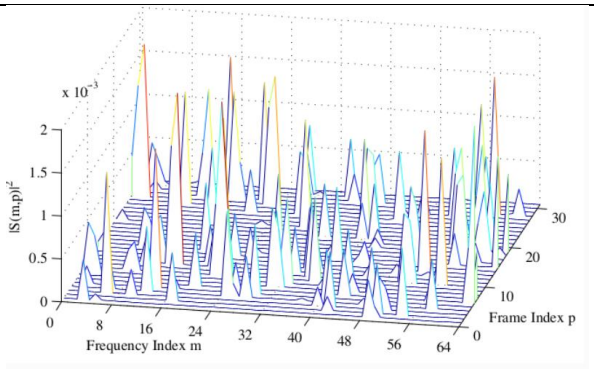

Figure 4: musical noise illustration[2]

Therefore, we focused on frequency bins outside human speech range in order to remove musical noise. Basically, the idea is that we could improve speech intelligibility to some extent if frequencies outside the range of speech frequency have noise is removed with greater coefficients. We considered this enhancement as a trade-off of creating a pass band filter only allowing the fundamental speech frequencies through while smoothly removing harmonics.

Typically, the voiced speech of an adult male will have a fundamental frequency from 85 to 180 Hz, and that of a typical adult female from 165 to 255 Hz[3][4]. And looking at the spectrum of the original clean audio file (figure 3), we confirmed that in this particular case, the speech frequency mainly ranges from 80 to 700 Hz. Therefore, instead of having a constant overestimation coefficient outside the human speech frequency bins, we came up with exaggerating overestimation coefficients which were generated using Matlab script, found in appendix.

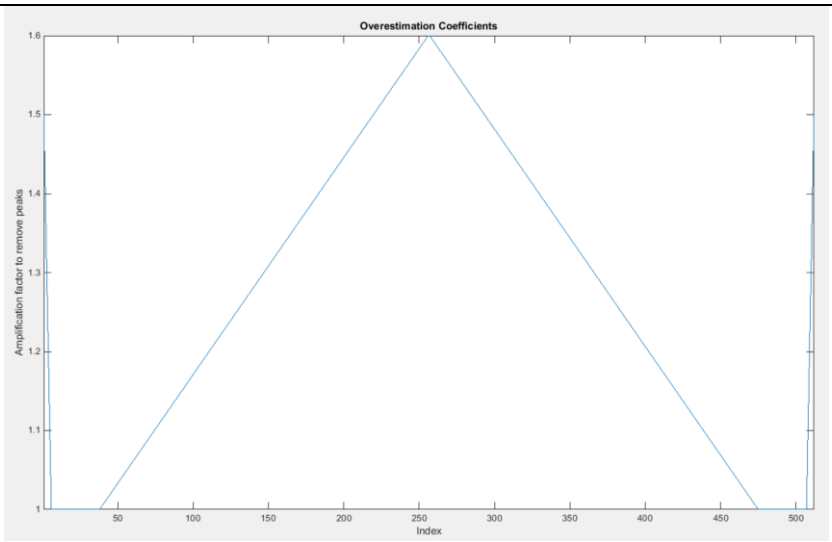

Figure 5: Matlab generated overestimation coefficients

Additional Enhancement Two (VAD)

(A step by step process can be found in the appendix, final code.)

The second additional enhancement implementation worked as basic voice activity detection (VAD) algorithm. Note that our implementation is not strictly a voice detection algorithm, it is more of a constant background noise detector, and its functionality still needs further improvements, which could be possible by building upon this algorithm.

The method firstly calculated the SNR of the signal across all frequencies, which can be described by a mathematical equation: $SNR_{in} = 10\log_{10}\left(\frac{\sum_{i=0}^{FFTLEN/2} S_i^2}{\sum_{i=0}^{FFTLEN/2} N_i^2}\right)$, where S and N are the amplitude of the input and noise respectively.

Note that due to the fact SNR various changes very often, therefore, a low pass filter was implemented using the equation given in enhancement one, and the filtered values can be described by mathematical equation: $LP\_SNR = SNR \times (1-k) + k \times SNR_{previous}$. The chosen k value was 0.852 (corner frequency at around 700Hz) since the speech frequency mainly ranges from 80 to 700Hz.

Then, we established two buffers storing the maximum and minimum SNRs, and similar to the noise buffers implemented previously in the basic design, these maximum and minimum SNR buffers are then switched every 0.5 seconds, with the oldest value being the past 2 seconds.
Therefore, we could calculate the minimum and maximum SNR over the 2 seconds period, found in the code below.

```
max_snr = max(max(Max_SNR[0],Max_SNR[1]),max(Max_SNR[2],Max_SNR[3]));
min_snr = min(min(Min_SNR[0],Min_SNR[1]),min(Min_SNR[2],Min_SNR[3]));
```

After that, we used a range between the maximum and minimum SNR to calculate the threshold, which can be described by the mathematical expression below:

$$\text{SNR\_Threshold} = Threshold\_coef \text{ *snr\_gap+ min\_snr;}$$

Note that SNR threshold is set to the lower 18% in this particular case or to your liking (in between 10% to 20%).

At last, the code detects and attenuates those amplitudes under the threshold, and the final results were demonstrated in the following section.

6. Results

This section summaries the effects on “phantom4” by applying basic enhancements and added extra enhancements. A full set of final outcomes of all audio files can be found in appendix with corresponding spectra.

6.1. Enhancements 2 & 4C

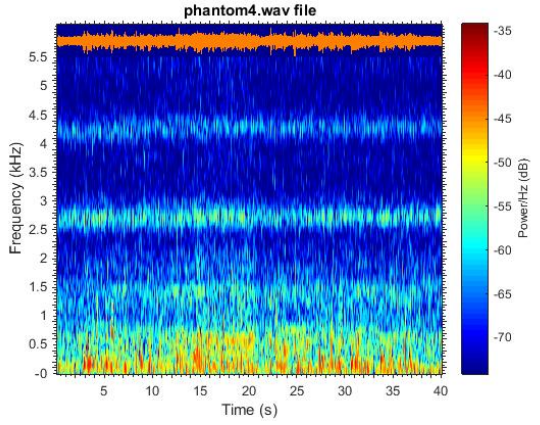

Figure 6: spectrum of original phantom4 audio file

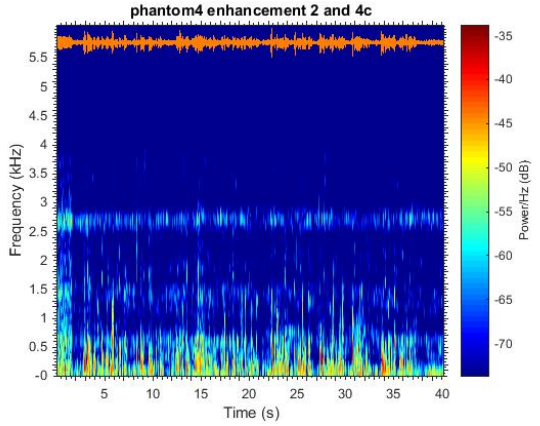

Figure 7: spectrum of “phantom4” using enhancements 2 and 4C

Enhancement 2 and 4c are the choice we made from the enhancements given in the project specification. They worked well on most of the provided audio files, but found to have a large amount of musical noise and a bit of background noise when listing to “Phantom4”. The results were shown in figure 6 and 7, we found that from frequencies around 2.2kHz to 2.7kHz, a large amount of musical noise exists. Therefore, in order to filter out these effects, we added additional enhancement one based on the current settings, and provided results in the following subsection.

6.2. Additional Enhancement One Plus Enhancements 2 & 4C

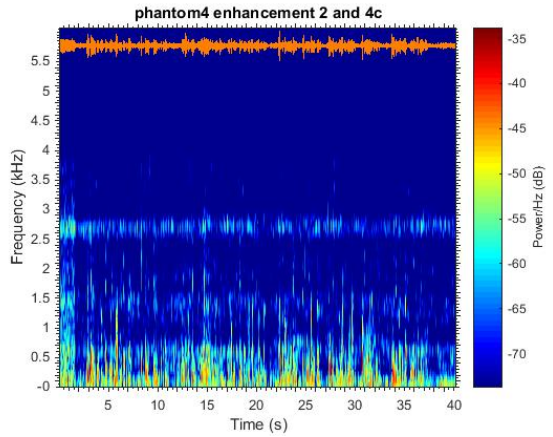

Figure 8: spectrum of "phantom4" applying enhancements 2 and 4C

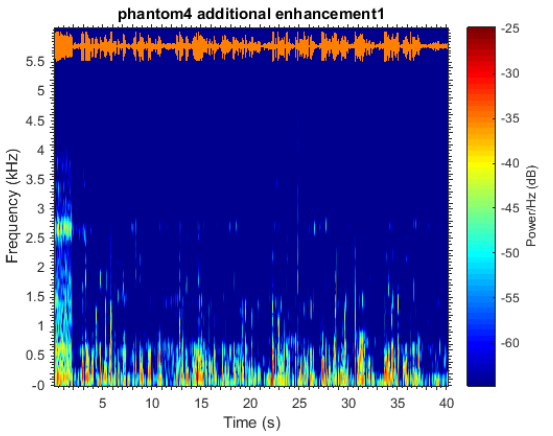

Figure 9: spectrum of "phantom4" using enhancements 2 & 4C plus additional enhancement1

After adding additional enhancement one based on existing enhancements 2&4C, we hardly noticed changes to other audio files, but even more musical noise reduction in "lynx2" and "phantom4". However, this enhancement only works with noise of frequency outside human speech band.

From figure 8 and 9, we found that additional enhancement one further removed musical noises from frequencies around 2.2kHz to 2.7kHz while did little change to the human speech band. In the next subsection (final results), we are going to add VAD to the existing enhancements, and discuss its effects.

6.3. Final Results - Additional Enhancement Two Plus Previous Enhancements

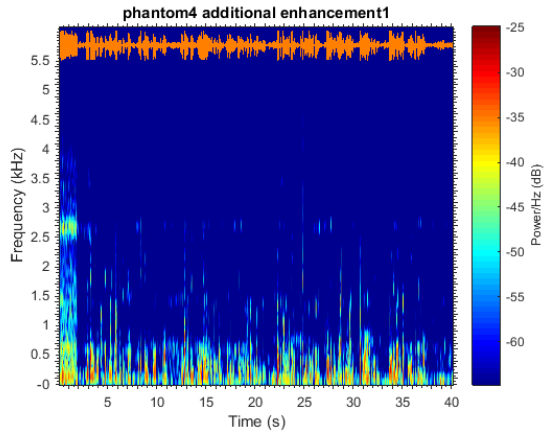

Figure 10: spectrum of "phantom4" using enhancements 2 & 4C plus additional enhancement1

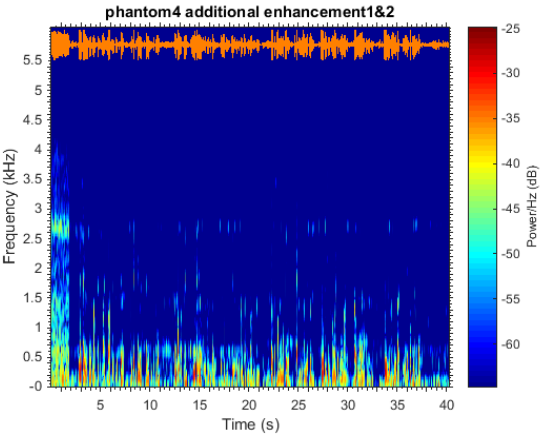

Figure 11: spectrum of "phantom4" using enhancements 2 & 4C + additional enhancement1&2

Figure 11 shows the spectrum of "phantom4" after applying enhancements 2&4 plus additional methods 1&2. As a result, in terms of musical noise we could see and hear little change in phantom4. However, when we listened to the result, we found more background noise was removed, and the results for all files are shown below.

Sound file	Observation	Sound file	Observation
car1.wav	Worked well	factory1.wav	Worked well
factory2.wav	Worked well – sometimes let a bit of noise though	lynx1.wav	Worked well
lynx2.wav	Worked well	phantom1.wav	Worked well
phantom2.wav	Worked well	phantom4.wav	Worked to an extent – reduced background and musical noise but sometimes deactivated itself

Table: the effects of VAD on all the provided sound samples.

7. Reference

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

[2] URL: http://www.vocal.com/noise-reduction/musical-noise/

[3] Boll,S.F.,"Suppression of Acoustic Noise in Speech using Spectral Subtraction", IEEE Trans ASSP 27(2):113-120, April 1979.

[4] Baken, R. J. (1987). Clinical Measurement of Speech and Voice. London: Taylor and Francis Ltd. (pp. 177), ISBN 1-5659-3869-0.

8. CODE

8.1. Matlab Script:

```
%%additional enhancement to remove musical noise
musical_coef = ones(512,1);
fs=8000;
%choose what factor to set the values to
%and where this should start
lowerbound_ampli = 2;
lowerbound_freq = 100;%we want to attenuate signals with frequency below
50Hz
normalised_lowerbound_freq = round(lowerbound_freq/fs*512);
upperbound_ampli = 2;
upperbound_freq = 600;%we want to attenuate signals with frequency higher
than 600Hz
normalised_upperbound_freq = round(upperbound_freq/fs*512);
for i=1:(normalised_lowerbound_freq)
    musical_coef(i) = (normalised_lowerbound_freq*lowerbound_ampli-
(lowerbound_ampli-1)*i)/normalised_lowerbound_freq;
    musical_coef(513-i) = (normalised_lowerbound_freq*lowerbound_ampli-
(lowerbound_ampli-1)*i)/normalised_lowerbound_freq;
end
a_eq = (upperbound_ampli-1)/(256-normalised_upperbound_freq);
b_eq = 1 - a_eq*normalised_upperbound_freq;
for i=normalised_upperbound_freq:256
    musical_coef(i) = a_eq*i+b_eq;
    musical_coef(513-i) = a_eq*i+b_eq;
end
plot(1:512,musical_coef)
axis tight
title('Overestimation coefficients')
xlabel('Index')
ylabel('Amplification factor to remove peaks')
fid=fopen('\\icnas3.cc.ic.ac.uk\ll4113\MATLAB\additional_filter.txt','w');
%open file
%define a and b coefficient arrays
fprintf(fid,'float musical_coef[] = {');
fprintf(fid,formatSpec,musical_coef(1:length(musical_coef)-1));
fprintf(fid,'%1.16f};\n',musical_coef(length(musical_coef)));
fclose(fid); %close file
```

8.2. Final 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"

// 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 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 FFTLEN 512                /* fft length = frame length 256/8000 = 32 ms*/
#define OUTGAIN 16000.0           /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)      /* Input gain for ADC  */

// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME (FRAMEINC/FSAMP)     /* time between calculation of each frame */
#define NoiseDetectPeriod 0.5       /* time for noise buffer shifting */
#define noise_count_loop (NoiseDetectPeriod/(FFTLEN/(OVERSAMP*FSAMP))) /*number of frames processed by each noise buffer
during a fixed period*/

//switches that turn on the implemented enhancements
int enhancement1 = 0;
int enhancement2 = 1;
int enhancement3 = 0;
int enhancement4a= 0;
int enhancement4b= 0;
int enhancement4c= 1;
```

```
int enhancement4d= 0;
int enhancement5 = 0;
int enhancement6 = 0;
int enhancement7 = 0;
int enhancement8 = 0;
int extra_enhancement1 = 0;
int additional_enhancement1 = 1;
int additional_enhancement2 = 1;
/****************************** 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 *M1,*M2,*M3,*M4;           /* Buffers used to store past noise estimates*/

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 inframe_complex[FFTLEN];   /* buffer that processes input frames */
float mag[FFTLEN];               /* array that stores the magnitudes of input frame spectrum */
float N[FFTLEN];                /* array that stores the noise spectrum */

int noise_count = 0;             /* counter for the number of frames that have been processed */

float alpha = 1.65;              /* noise overestimation factor - typically ranges from 1 to 40 */
float lambda = 0.02;             /* floor of gain factor - typically ranges from 0.01 to 0.1 */

/* enhancement 1&2 */
float K = 0.852;                /* exp(-TFRAME/0.05) */
float LP_mag[FFTLEN];            /* array that stores the magnitudes of input frame spectrum */
float LP_mag_prev = 0;           /* previous element of LP_mag[k] */

/* enhancement 3 */
float N_prev = 0;                /* previous value of noise[k] */

/* enhancement 6 */
int enhancement6_lower_bound = 5;  /* lower bound for noise overestimation */
int enhancement6_higher_bound = 24; /* upper bound for noise overestimation */
float gamma = 1.65;              /* noise overestimation factor for selected lower and upper frequency spectra */

/*enhancement 8*/
complex Delay[FFTLEN];           /* stores the result */
int set_on = 0;                 /* activate residue */
float residue = 0.5;             /* set residue value */

/* additional enhancement1 */
#include "additional_filter.txt"
```

```
/* additional enhancement2 */
float S_Power = 0;                  /* Signal Power */
float N_Power = 0;                  /* Noise Power */
float SNR = 1;                      /* Calculated signal noise ratio */
float LP_SNR = 1;                   /* Low pass filtered SNR */
//used value K in enhancement 1&2
float LP_SNR_Threshold = 0;         /* threshold for LPFed SNR*/
float threshold = 0.20;             /* Calclated/estimated SNR*/
float Max_SNR[4] = {0,0,0,0};       /* Buffers that stores max SNR */
float Min_SNR[4] = {0,0,0,0};       /* Buffers that stores min SNR */
float VAD_coef = 0.1;               /* Voice activity detection factor */
float min_snr = 0;                  /* current minimum value of SNR */
float max_snr = 0;                  /* current maximum value of SNR */
float snr_gap = 0;                  /* current SNR range */
int   VAD_on = 1;                   /* turn VAD on/off */
int   SNR_buffer_index = 0;         /* index for snr min max buffers */
int   snr_count = 0;                /* count buffer processes */


 /****************************** 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 max(float a, float b);
float min(float a, float b);
complex minComplex(complex a, complex b);
/********************************* Main routine *********************************/
void main()
{

   int k; // used in various for loops

/*  Initialize and zero fill arrays */

   inbuffer    = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
   outbuffer   = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
   inframe     = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
   outframe    = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
   inwin       = (float *) calloc(FFTLEN, sizeof(float));  /* Input window */
   outwin      = (float *) calloc(FFTLEN, sizeof(float));  /* Output window */

   M1          = (float *) calloc(FFTLEN, sizeof(float)); /*past noise spectra */
   M2          = (float *) calloc(FFTLEN, sizeof(float)); /*past noise spectra */
   M3          = (float *) calloc(FFTLEN, sizeof(float)); /*past noise spectra */
   M4          = (float *) calloc(FFTLEN, sizeof(float)); /*past noise spectra */

   for(k=0;k<NFREQ;k++){
      M1[k]=0;
      M2[k]=0;
      M3[k]=0;
      M4[k]=0;
   }
   //mag         = (float *) calloc(FFTLEN, sizeof(float));
   //GainFactor  = (float *) calloc(FFTLEN, sizeof(float));
   /* initialize board and the audio port */
   init_hardware();

   /* initialize hardware interrupts */
   init_HWI();

/* initialize algorithm constants */

   for (k=0;k<FFTLEN;k++)
   {
   inwin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
   outwin[k] = inwin[k];
   }
   ingain=INGAIN;
```

```
    outgain=OUTGAIN;


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

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

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

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

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

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


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

}

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

/* copy input frame to frame processing buffer*/
for(k=0;k<FFTLEN;k++){
   inframe_complex[k] = cmplx(inframe[k], 0);
}
/* FFT of input frames */
fft(FFTLEN, inframe_complex);

//VAD on
if(additional_enhancement2==1){
 //increase SNR buffer count
 snr_count++;
 //reset input power variables to 0
 S_Power = 0;
 //every defined number of frames reset the noise power estimate to be recalculated
 if (snr_count >= noise_count_loop/5)//
 {
     N_Power = 0;
 }
}

for(k=0;k<NFREQ;k++){
   /* find magnitude of input frame spectrum */
   // basic implementation
   mag[k] = cabs(inframe_complex[k]); // find magnitude spectrum
   if(additional_enhancement2==1){//VAD on
      S_Power += mag[k]*mag[k];//calculate signal power
   }
   if(enhancement1==1){    // enhancement 1
      LP_mag[k] = (1-K)*(mag[k])+K*LP_mag_prev;
      LP_mag_prev = LP_mag[k];
   }
   else if(enhancement2==1){   // enhancement 2
      LP_mag[k] = sqrt((1-K)*(mag[k]*mag[k])+K*LP_mag_prev*LP_mag_prev);
      LP_mag_prev = LP_mag[k];
   }

   /* place minimum of input spectrum into noise buffer M1*/
   if(enhancement1 == 1 || enhancement2 == 1){//enhancement 1 or 2
      if(M1[k]>LP_mag[k]){
         M1[k] = LP_mag[k]; // assign minimum
      }
   }
   else if(M1[k]>mag[k]){  // basic implementation
      M1[k] = mag[k]; // assign minimum
   }
}

/* noise buffers*/
if(++noise_count >= noise_count_loop){ // wrap after a fixed period
   noise_count = 0;    // rest noise frame counter
   // shifting POINTERS
   temp = M4;
   M4= M3;
   M3= M2;
   M2= M1;
   M1 = temp;  // M1 points to M4
   /* assign inputs to noise buffer*/
   for(k=0;k<NFREQ;k++){
      if(enhancement1==1 || enhancement2 == 1){// enhancement 1 or 2
         M1[k] = LP_mag[k];
      }
      else{ // basic implementation
         M1[k] = mag[k];
      }
   }
}

/* estimate noise */
```

```
for(k=0;k<NFREQ;k++){
  if(enhancement3==1){   // enhancement 3 - this will only have a noticeabnle effect if the noise level is very variable
    /* low pass filter the overestimated noise */
    //N_prev = 0; // maybe delete
    N[k] = (1-K)*alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k])) + K*N_prev;
    N_prev = N[k];

  }
  else if(enhancement6==1){   // enhancement 6 - deliberately overestimate the noise level
    /*reduce the musical noise artifactes that are introduced by spectral subtraction*/
    if(k<enhancement6_lower_bound || k>enhancement6_higher_bound){
      N[k] = gamma*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
    }
    else{
      N[k] = alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
    }
  }
  else if(additional_enhancement1==1){//additional enhancement1 based on enhancement 6
    N[k] = musical_coef[k]*alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
  }
  else{   // basic implementation
    N[k] = alpha* min(min(M1[k],M2[k]),min(M3[k],M4[k]));
  }
}

//VAD on
if(additional_enhancement2==1){
  if (snr_count >= noise_count_loop/5){
    //calculate power
    N_Power += N[k]*N[k]/alpha;
  }
}

/* noise subtraction using gain factor */
for(k=0;k<NFREQ;k++){
  if(enhancement4a == 1){
    // GainFactor[k] = max(lambda, 1 - (N[k]/mag[k])) where lamba is lower floor
    // then multiply noise by gain factor (complex multiplication using rmul function in complex.h)
    inframe_complex[k] = rmul(max(lambda*(N[k]/mag[k]), 1 - (N[k]/mag[k])), inframe_complex[k]);
  }
  else if(enhancement4b == 1){
    inframe_complex[k] = rmul(max(lambda*(LP_mag[k]/mag[k]), 1 - (N[k]/mag[k])), inframe_complex[k]);
  }
  else if(enhancement4c == 1){
    inframe_complex[k] = rmul(max(lambda*(N[k]/LP_mag[k]), 1 - (N[k]/LP_mag[k])), inframe_complex[k]);
  }
  else if(enhancement4d == 1){
    inframe_complex[k] = rmul(max(lambda, 1 - (N[k]/LP_mag[k])), inframe_complex[k]);
  }
  else if(enhancement5 == 1){
    inframe_complex[k] = rmul(max(lambda, sqrt(1 - (N[k]/mag[k])*(N[k]/mag[k]))), inframe_complex[k]);
  }
  else if(extra_enhancement1 == 1){
    inframe_complex[k] = rmul(max(lambda*(N[k]/LP_mag[k]), sqrt(1 - (N[k]/mag[k])*(N[k]/mag[k]))), inframe_complex[k]);
  }
  else{   // basic implementation
    inframe_complex[k] = rmul(max(lambda, 1 - (N[k]/mag[k])), inframe_complex[k]);
  }
}

//VAD on
if(additional_enhancement2==1){
  //calculate SNR
  SNR = 10*log10(S_Power/N_Power);
  //LPF the SNR like enhancement 1&2
  LP_SNR = (1-K)*SNR+K*LP_SNR;
  //Keep the result within a range in order to avoid overflows
  if ( (LP_SNR >= 9) || (LP_SNR <= -18)){
    LP_SNR = 0;
  }
  //Process noise if speech not detected
```

```
    if ((LP_SNR <= LP_SNR_Threshold) && (VAD_on==1)){
      for (k=0;k<NFREQ;k++){
        inframe_complex[k]= rmul(VAD_coef,inframe_complex[k]);
      }
    }
    //calculate the minimim and maximum SNRs
    Max_SNR[SNR_buffer_index] = max(LP_SNR,Max_SNR[SNR_buffer_index]);
    Min_SNR[SNR_buffer_index] = min(LP_SNR,Min_SNR[SNR_buffer_index]);
    //reset SNR count
    if (snr_count >= noise_count_loop/5){
      snr_count = 0;
      //update min and max SNR
      max_snr = max(max(Max_SNR[0],Max_SNR[1]),max(Max_SNR[2],Max_SNR[3]));
      min_snr = min(min(Min_SNR[0],Min_SNR[1]),min(Min_SNR[2],Min_SNR[3]));
      snr_gap = max_snr - min_snr;
      if (N_Power <= 1){//power too small
        VAD_on = 0;//turn off to avoid miscalculation
        threshold = 0.06;
      }else{
        if (snr_gap <= 3){//if gap too small
          VAD_on = 0;//turn off VAD
          threshold = 0; //set to 0
        }else{
          //turn VAD on
          VAD_on = 1;
          //decrease any non-speech inputs by 0.22
          threshold = 0.22;
        }
      }
      //calculate dynamic threshold
      LP_SNR_Threshold = min_snr + snr_gap*threshold;
      // shift min and max SNR buffers
      if (++SNR_buffer_index >= 4){
        SNR_buffer_index = 0;//reset index
      }
      //assign to min and max SNR buffers
      Min_SNR[SNR_buffer_index] = LP_SNR;
      Max_SNR[SNR_buffer_index] = LP_SNR;
    }
  }  /*VAD end*/


  if(enhancement8 == 1){ // enhancement 8
    for(k=0;k<NFREQ;k++){
      if(set_on == 1){
        //find adjacent min
        Delay[k]= minComplex(minComplex(inframe_complex[k-1],inframe_complex[k-2]),inframe_complex[k]);
        //deactivate
        set_on = 0;
        if(N[k]/mag[k]>residue)set_on = 1;
        else if(N[k]/mag[k]>residue){
          //activate
          set_on = 1;
          Delay[k] = inframe_complex[k-1];
        }
        else{//do nothing but replace the frame
          Delay[k] = inframe_complex[k-1];
        }
      }
    }
  }


  /* copy the other half of the buffer (the FFT spectrum is symmetric; details can be found below) */
  // bins 129 - 255 are complex conjugates of bins 127 to 1.
  // bins 0 and 128 are independent.
  if(enhancement8 == 1){
    for(k=0;k<NFREQ;k++){
      inframe_complex[k] = Delay[k];
    }
    for(k=NFREQ;k<FFTLEN;k++){
```

```
            inframe_complex[k] = conjg(Delay[FFTLEN-k]);
        }
    }
    else{
        for(k=NFREQ;k<FFTLEN;k++){
            inframe_complex[k] = conjg(inframe_complex[FFTLEN-k]);
        }
    }

    /******************************** end of frequency domain processing ***********************************/

    /* inverse FFT of the spectrum of the processed frame */
    ifft(FFTLEN, inframe_complex);

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


    /******************************** overlap add ********************************************************/
    /* 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;
}

/********************************* finding minmum and maximum
functions*****************************************/

float min(float a, float b){
    if(a>b)return b;
    else return a;
}
float max(float a, float b){
    if(a>b)return a;
    else return b;
}
complex minComplex(complex a, complex b){
    if(cabs(a)<cabs(b))return a;
    else return b;
}
```