DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING IMPERIAL COLLEGE LONDON

EE3-19
REAL-TIME DIGITAL SIGNAL PROCESSING
SPEECH ENHANCEMENT PROJECT REPORT

YASMIN BABA - 01196634
ZHI WEN SI - 01243716

22nd of March, 2019

1. Introduction

Speech enhancement is crucial in today's world of signal processing and plays an important role in technology such as telecommunications, hearing aids, recordings, etc. Nowadays, such technology is expected to cancel out background noise to a high standard, leaving only a clear speech signal remaining. The following report explores techniques to implement and improve a real-time speech enhancer, with the aim to remove background noise present in a speech signal via an algorithm known as spectral subtraction.

The basic spectral subtraction algorithm implementation is explored in *Section 2*. However, as discussed in *Section 2.6*, spectral subtraction has been found to introduce distortions that result in an undesirable output. Various enhancements are thus investigated in *Section 3* to try to remove them.

2. Basic Algorithm

Spectral subtraction is an algorithm implemented in the frequency domain and assumes that the input signal's spectrum can be expressed as a sum of the speech spectrum and the noise spectrum. The basic algorithm is illustrated in *Figure 1* and involves taking the FFT of the input signal, subtracting an estimate of the noise spectrum from the signal spectrum and taking the IFFT to, theoretically, produce a noise-free output speech signal.

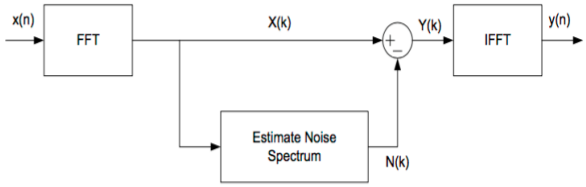

Figure 1: Spectral Subtraction

2.1 Overlap Add Process

Before implementing the basic spectral subtraction algorithm, it is important to note that when the FFT is taken, discontinuity at the frame boundaries will give rise to spectral artefacts. In order to avoid such a phenomenon, the frames of data are multiplied by the square root of a Hamming window:

$$\sqrt{1 - 0.85185\cos((2k+1)\pi/N)} \quad \text{for } k = 0, \dots, N-1$$

This results in the frame's amplitude to reduce to zero at the edges. However, windowing the data will result in a time domain signal with varying amplitude. This is avoided by overlapping frames in the continuous-time domain. Once the frame processing is completed, the frames can then be reassembled to create the continuous output signal. As a sample size of 256 is used, the output signal is formed by adding a continuous stream of 256 frames, each multiplied by an input and output window.

The overlapping windows sum to a constant, theoretically resulting in no distortion of the output signal. Throughout this report, an oversampling ratio of 4 is used; in other words, each frame will begin a quarter of a frame later than the previous. This is done to avoid distortion from the processing, which may change the gain of the frequency bins abruptly between successive frames.

2.2 Fast Fourier Transform

In order to implement the basic algorithm, the real continuous-time input data stored in ***inframe*** is firstly converted into the frequency domain via a FFT, as shown below in *Listing 1*. However, the parameter passed into the **fft** function must be in the form of a complex number. Therefore, ***inframe*** is converted into a complex array ***Y*** with all imaginary parts set to zero. Once the FFT is taken, the magnitude spectrum is found via the **cabs** function and is stored in the array ***X***.

```
//*** Convert Input Data into Complex Number ***//
for(k = 0; k < FFTLEN; k++){
    Y[k] = cmplx(inframe[k],0);
}

//**** FFT ****//
fft(FFTLEN, Y);

for(k = 0; k < FFTLEN; k++){
//**** Magnitude Spectrum ****//
    X[k] = cabs(Y[k]);
```

Listing 1: FFT of Input Signal

2.3 Noise Estimation

In order to estimate the noise, human nature is taken advantage of. It is assumed that the average talking person will need to take a breath, or will generally pause, at least once every 10 seconds. During this pause, only background noise will be present in the signal and the signal's power will be minimal. The estimated noise for each frequency bin will be the minimum magnitude present in any frame over the past 10 seconds. However, there are two problems with this implementation.

Firstly, measuring the speech spectra over 10 seconds will require a lot of data storage and is highly inefficient. Instead, the minimum spectrum is stored over four 2.5 second intervals in four buffers named $\boldsymbol{M1}, \boldsymbol{M2}, \boldsymbol{M3}$ and $\boldsymbol{M4}$. For each frame, $\boldsymbol{M1}$ is updated as follows,

$$\boldsymbol{M1} = min\{\boldsymbol{M1}, \boldsymbol{X}\}$$

Every 2.5 seconds, the buffers rotate, losing the oldest set of data to obtain the newest.

$$\boldsymbol{M4} = \boldsymbol{M3}, \boldsymbol{M3} = \boldsymbol{M2}, \boldsymbol{M2} = \boldsymbol{M1}, \boldsymbol{M1} = min\{\boldsymbol{M1}, \boldsymbol{X}\}.$$

The noise estimate for the past 10 seconds is the minimum of these four buffers. Although less accurate, this approach is more efficient, requiring less processing and storage.

```
//**** Noise Estimation **//
M1[k] = min(X[k], M1[k]);
N[k] = min(min(M1[k], M2[k]), min(M3[k], M4[k]));
```

Listing 2: Noise Estimation

```
//**** Buffer Rotation ****//
if(counter >= frame_count){ // frame_count = 2.5/TFRAME
    counter = 0;

    temp = M4;
    M4 = M3;
    M3 = M2;
    M2 = M1;
    M1 = temp;

    for(k = 0; k < FFTLEN; k++){
        M1[k] = X[k];
    }
}
counter++;
```

Listing 3: Buffer Rotation

Furthermore, as the buffers store the minimum magnitudes in each frequency bin that occurs during non-speech intervals, they do not take into account noise found at higher magnitudes. In other words, the noise is underestimated. To compensate for this, a scaling factor $\boldsymbol{\alpha}$ is multiplied with the noise estimation to represent the average noise. For now, $\boldsymbol{\alpha}$ has been set as 20.

```
Nmin[k] = alpha * Nmin[k];
```

Listing 4: Alpha - Over Subtraction Factor

Note that a 2.5s rotation will occur roughly every 312 frames, and thus ***frame_count*** is set to 312. ***frame_count*** is calculated as:

$$frame\_count = round(\frac{time\ inverval}{TFRAME})$$

2.4 Subtracting the Noise Spectrum

It is important to note that the minimum noise buffer only holds values of magnitude and thus the phase of the noise signal is unknown. If the phase was known, the noise could simply be removed from the original signal by subtracting their FFT values (taking into account both magnitude and phase): $Y(k) = X(k) - N(k)$. Instead, only the magnitude of the spectra will be subtracted from one another: $|Y(k)| = |X(k)| - |N(k)|$. This is implemented as follows,

$$Y(k) = X(k)\left(\frac{|X(k)| - |N(k)|}{|X(k)|}\right)$$

$$= X(k)\left(1 - \frac{|N(k)|}{|X(k)|}\right) = X(k)G(k)$$

However, if the noise is over estimated, $\boldsymbol{G(k)}$ may fall negative. In order to ensure positive values at each frequency bin, a minimum spectral floor parameter is set according to $\boldsymbol{\lambda}$, which has been experimentally set to 0.02.

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

```
//*** Spectral Floor ***//
G[k] = max(lambda, 1-(Nmin[k]/X[k]));

//*** Noise Subtraction ***//
Y[k] = rmul(G[k], Y[k]);
```

Listing 5: Spectral Floor Constant and Noise Subtraction

2.5 IFFT

Once the noise has been subtracted, the IFFT of ***Y*** is taken. It was expected that due to the symmetry of an FFT filter, only real numbers will be stored in ***Y*** once the IFFT is taken. However, in reality, the numbers were found to have small complex components which are believed to be due to the finite precision of floats. Only the real parts of ***Y*** are therefore stored in ***outframe***.

```
//**** IFFT ****//
ifft(FFTLEN,Y);

//**** Output ****//
for(k = 0; k < FFTLEN; k++){
    outframe[k] = Y[k].r;
}
```

Listing 6: Output Implementation

2.6 PERFORMANCE

When the simple algorithm implementation was tested against corrupted speech signals, the noise was reduced significantly. In order to visualise the improvement, the spectrograms of the corrupted signal and the signal produced via the simple algorithm are compared in *Figures 2* and *3*.

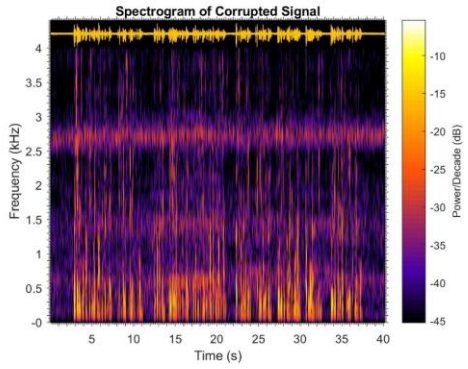

Figure 2: Corrupted Signal

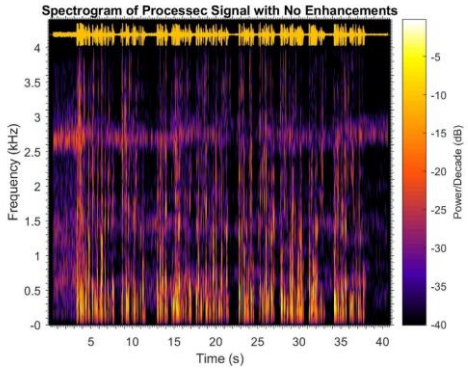

Figure 3: Processed Signal

However, the main drawback of the spectral subtraction method is a phenomenon referred to as musical noise. Musical noise is a type of distortion described as ringing of tonal quality or a metallic noise and exists due to white noise present in the signal [1]. In short-term power spectra, this noise contains 'peaks' and 'valleys' that vary randomly in frequency and amplitude throughout each frame; the noise magnitude spectrum has a high variance. After subtracting the noise estimate from the input signal, these spectral peaks are shifted down whereas values lower than the estimate in the valleys are set to zero (this occurs due to under-subtraction). Musical noise refers to the isolated narrow peaks, as illustrated below in *Figure 4*, and correspond to tones in the time-domain which the human ear can detect easily.

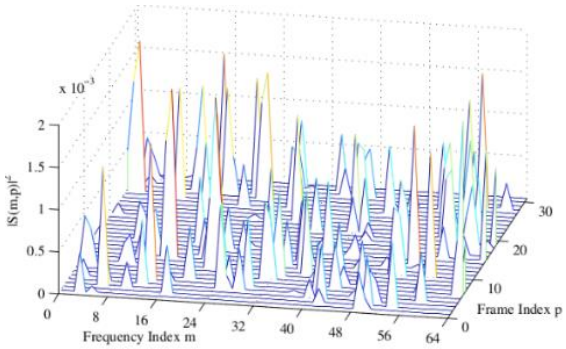

Figure 4: Musical Noise Visualisation [2]

Referring back to *Figure 3*, the musical noise can be seen in the spectrogram as random speckling in colour, due to the randomly distributed energy.

Finally, it was discovered that the performance heavily depended on the value of $\boldsymbol{\alpha}$. A higher value of $\boldsymbol{\alpha}$ reduces the amount of musical noise present within the signal; however, it does so at the cost of reducing the intelligibility of the speech.

3. ENHANCEMENTS

Overall, the basic algorithm of spectral subtraction has not produced a high quality speech enhancer. In order to improve it, various enhancements have been explored and evaluated as follows.

3.1 ENHANCEMENT 1

The first enhancement aims to reduce the musical noise present in the signal by applying a low-pass filter across the magnitude spectrum of successive frames in the magnitude domain. As discussed earlier, musical noise is due to a randomly varying noise spectrum. The low-pass filter will average out the time-varying magnitudes of each frequency bin in order to reduce this high variance of noise via the following algorithm:

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

where $k = \exp(-\frac{frame\ length}{\tau})$.

```
//**** Enhancement 1 ****//
if (en1 == 1){
    //**** LPF in magnitude domain ****//
    P[k] = (1-filterK)*X[k] + filterK*P[k];
    X[k] = P[k];

}
```

Listing 7: Low-Pass Filter in the Magnitude Domain

The low pass filter is an exponentially weighted moving average filter with an order of 1. The filter's transfer function is:

$$H(z) = \frac{P_t(\omega)}{|X(\omega)|} = \frac{1-k}{1-kz^{-1}}$$

The time constant of the filter is represented by $\boldsymbol{\tau}$ and plays a role in determining the location of the pole of the filter and thus the desired corner frequency. Different values of $\boldsymbol{\tau}$ were tested, however, the difference between the given range of $20-80\ ms$ was not noticeably clear. *Figures 5* and *6* display the magnitude spectra when $\tau = 20ms$ and $\tau = 80\ ms$ respectively.

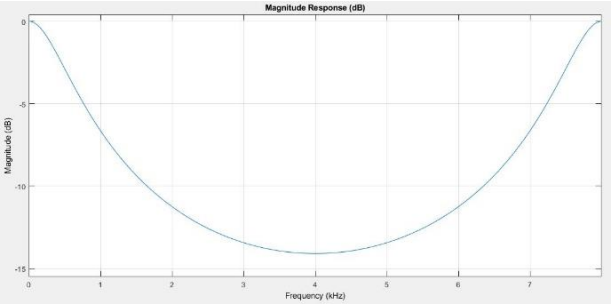

Figure 5: $\tau = 0.02$

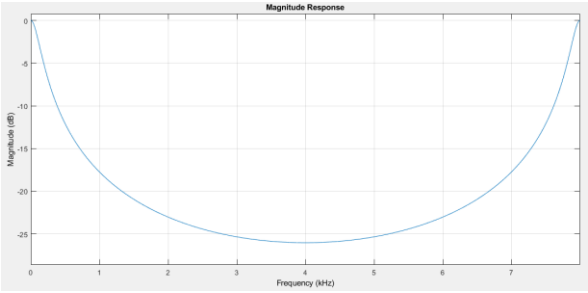

Figure 6: $\tau = 0.08$

As high frequencies are attenuated, the magnitude spectrum of the current frame at each frequency bin will not change abruptly compared to the previous frame. A higher time constant results in a larger delay effect on changes in magnitude and thus is desirable in order to reduce musical noise. However, there is a trade-off between reducing the musical noise and having the flexibility to deal with fast changing signals, which a speech signal naturally is. Considering this trade-off, a midpoint value of $\tau = 50ms$ will be implemented.

Overall, when this enhancement was tested, both the speech and the noise became much softer. This is due to the gain factor, $\boldsymbol{G(k)}$, over-hitting the spectral floor. This was fixed by decreasing $\boldsymbol{\alpha}$ which is now possible because the enhancement allows for a more accurate estimation of the noise. The value of $\boldsymbol{\alpha}$ was found to be best at a value of 3.5 - the speech was heard clearly with the background noise greatly reduced.

3.2 ENHANCEMENT 2

The low-pass filter can also be performed in the power domain as follows.

$$P_t(\omega) = \sqrt{(1-k) \times |X(\omega)|^2 + k \times P_{t-1}(\omega)^2}$$

```
//**** Enhancement 2 ****//
else if (en2 == 1){
    //**** LPF in pow domain ****//
    P[k] = sqrt((1-filterK)*X[k]*X[k] + filterK*P[k]*P[k]);
    X[k] = P[k];
}
```

Listing 8: Low-Pass Filter in the Power Domain

As the magnitude spectra are now squared, changes in magnitude between the successive frames will be more significant. This filter will therefore be able to smoothen out the noise

spectrum more effectively as it will be able to better identify rapidly changing frequencies.

However, as all frequency bins experience the same delaying effect, the magnitude of the speech is affected as well, causing the speech interval of the output audio to sound slightly slurred.

This applies to both enhancement 1 and 2, however, although not obviously heard, the effect will be more prominent in enhancement 2.

3.3 Enhancement 3

Enhancement 3 explores the effect of applying the low-pass filter across the noise estimate. In other words, instead of filtering every frequency bin of the signal, only the noisy frequency bins are targeted. Doing so will avoid abrupt discontinuities when the minimum noise buffers are rotated.

```
//**** Enhancement 3 ****//
if (en3 == 1){
    //**** LPF in magnitude domain ****//
    Nt[k] = (1-filterK)*Nmin[k] + filterK*Nt[k];
    Nmin[k] = Nt[k];
}
```

Listing 9: Low-Pass Filtering the Noise Spectrum

It was difficult to notice whether this enhancement had a large effect overall as it is only noticeable when the noise level is highly variable. However, when both $\boldsymbol{\lambda}$ and $\boldsymbol{\alpha}$ are increased such that the noise is more prominent, this enhancement was found to reduce it very slightly.

3.4 Enhancement 4

This enhancement involves varying the spectral floor parameter $\boldsymbol{\lambda}$ dynamically with noise and/or the magnitude spectrum.

```
//**** Enhancement 4 ****//
if(en4_a == 1){
    G[k] = max(lambda*(Nmin[k]/X[k]),1-(Nmin[k]/X[k]));
}
else if(en4_b == 1){
    G[k] = max(lambda*(P[k]/X[k]),1-(Nmin[k]/X[k]));
}
else if(en4_c == 1){
    G[k] = max(lambda*(Nmin[k]/P[k]),1-(Nmin[k]/P[k]));
}
else if(en4_d == 1){
    G[k] = max(lambda,1-(Nmin[k]/P[k]));
}
```

Listing 10: Dynamic Spectral Floor in Magnitude Domain

The spectral floor parameter aims to set any negative frequency values to a very small positive number, resulting in a reduction in the varying range of the spectral peaks referred to as musical noise. Having too high of a constant will result in a background humming sound and musical noise whereas having too low of a constant will make the speech sound unnatural as it cuts off any ambient noise. It is beneficial to dynamically alter the parameter with the noise-to-signal ratio; for example, if the noise component at a certain frequency bin is higher than the speech component, a larger floor value is desired to avoid losing too much of the speech information.

The different implementations were tested and the findings were recorded in the table below.

Enhancement	Result
4a	Reduction in musical noise Introduces a slight fuzzy noise
4b	Louder fuzzy noise
4c	Slight echo effect
4d	Crackling sound

Table 1: Enhancement 4 Comparisons

Enhancement 4c was found to be best. It produced a very slight echo effect, however, the speech quality was improved overall. The others introduced speech distortions and crackling sounds.

3.5 Enhancement 5

Enhancement 5 repeats the algorithms used in enhancement 4 to dynamically change the spectral floor value in the power domain.

```
//**** Enhancement 5 ****//
else if(en5_a == 1){
    G[k] = max(lambda,sqrt(1-(Nmin[k]*Nmin[k]/(X[k]*X[k]))));
}
else if(en5_b == 1){
    G[k] = max(lambda*(Nmin[k]/X[k]),1-(Nmin[k]*Nmin[k]/X[k]*X[k]));
}
else if(en5_c == 1){
    G[k] = max(lambda*(P[k]/X[k]),1-(Nmin[k]*Nmin[k]/X[k]*X[k]));
}
else if(en5_d == 1){
    G[k] = max(lambda*(Nmin[k]/P[k]),1-(Nmin[k]*Nmin[k]/P[k]*P[k]));
}
else if(en5_e == 1){
    G[k] = max(lambda,1-(Nmin[k]*Nmin[k]/P[k]*P[k]));
}
```

Listing 11: Dynamic Spectral Floor in Power Domain

All the above options resulted in a worse output whereby the audio signal became distorted with the background noise being significant increased.

3.6 Enhancement 6

Enhancement 6 involves reducing musical noise by overestimating the noise at low frequency bins that have a poor signal-to-noise ratio (SNR). The human voice typically ranges from $85 - 255Hz$ [3], corresponding to the frequency bins 2.71 - 8.16. Therefore, to implement this enhancement, the frequency bins 0 - 9 will be overestimated.

However, it is important to note that taking the FFT results in symmetry of frequency bins. This means frequency bins 246 - 255 must too be overestimated. As illustrated in *Listing 12*, the ratio of the input signal's and the noise's magnitude spectra are compared against a threshold named ***SNR_thres***. If the ratio lies below this threshold then $\boldsymbol{\alpha}$ is increased by tenfold; else, $\boldsymbol{\alpha}$ remains at 3.5.

```
//**** Enhancement 6 ****//
if (en6 == 1){
    if ((k <= 9)|| (k >= 246)){
        if(X[k]/Nmin[k] < SNR_thres){
            alpha = alpha*10;
        }
        else {
            alpha = alpha;
        }
    }
    else{
        alpha = alpha;
    }
    Nmin[k] = alpha*Nmin[k];
}
```

Listing 12: Overestimating the Noise at Low Frequencies

The ***SNR_thres*** has been set to $-5\ dB$, corresponding to $10^{\frac{-5}{20}} = 0.56234$. It is set as such in order to save computation.

$$\alpha = \begin{cases} 10\alpha, & \frac{|X(k)|}{|N(k)|} < 0.56234 \\ \alpha, & otherwise \end{cases}$$

This enhancement is relatively similar to the algorithm discussed Berouti [1], which will be implemented in enhancement 10. For now, this enhancement has no noticeable effect.

3.7 Enhancement 7

This enhancement explores the effect of altering ***FFTLEN*** which defines both the frame length and the number of FFT points.

```
#define FFTLEN 256
```

Listing 13: FFTLEN

If the ***FFTLEN*** is reduced, shorter frames are produced allowing for a more frequent signal processing. However, the amount of FFT points is reduced as well, lowering the number of frequency bins. Despite being able to process the input signal more frequently, the actual processing is deteriorated which is undesired. When ***FFTLEN*** is defined as 128, a buzzing sound was introduced and the speech was attenuated.

If ***FFTLEN*** is increased, more FFT points can be computed, increasing the frequency resolution. Yet, when ***FFTLEN*** is defined as 512, the output audio sounded slurred because the frames are now longer and cause an increased delay between the input and output.

Since both increasing and decreasing ***FFTLEN*** have a negative impact on the overall audio quality, the original ***FFTLEN*** of 256 is preserved.

3.8 Enhancement 8

This enhancement looks to remove musical noise from the magnitude spectrum of the output signal. Since musical noise is random from frame-to-frame in both magnitude and frequency, the minimum of 3 adjacent output frames at each frequency bin are taken when the noise-to-signal ratio is bigger than a threshold named **residual** [4]. Therefore, peaks that occur for only one or two frames will be removed. These peaks will mainly be musical noise components as speech components tend to last longer than 3 frames.

By implementing this enhancement, a 1-frame delay between the input and the output is introduced. This is because the algorithm requires

the comparison between the current, previous, and next frames.

```
//**** Enhancement 8 ****//
if (en8 == 1){
    if (Nmin[k]/X[k] > residual){
        Yout[k] = minComplex(minComplex(Yprev[k],Ynext[k]),Y[k]);
    }
    else{
        Yout[k] = Ynext[k];
    }
    Yprev[k] = Y[k];
    Y[k] = Ynext[k];
}
```

Listing 14:Taking the Minimum of 3 Adjacent Frames

Instead of visualising the three frames as $y(n) = \min(y(n-1), y(n), y(n+1))$, one can think of it as $y(n) = \min(y(n-2), y(n-1), y(n))$, keeping in mind that $y(n)$ is the output for $x(n-1)$.

Considering that the noise at this point is already overestimated, the threshold ***residual*** is initially set to 7. However, whilst testing, it was changed in real time to try and find the best value. However, overall, the enhancement introduced a crackling sound and a chopped speech signal.

The ***cpufrac*** variable provided was used to monitor whether the signal processing was being completed in time before the arrival of the next frame. It was observed that when enhancement 8 was turned on, the processing was indeed not being completed in time, resulting in the distorted output.

In order to overcome this problem, the code must be made more efficient. This can be done by exploiting the symmetry property of the FFT. Since the magnitude spectrum is an even function and is periodic at every sampling frequency, the ***for*** loop used for processing can be reduce to half its current size.

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

Listing 15: FFT Symmetry at the Input

However, before carrying out the IFFT of the processed output frames, the second half of the output frame must be created and is done so by using the first half as conjugate pairs.

```
//**** Symmetry Property ****//
for (k = FFTLEN/2+1; k < FFTLEN; k++){
    Y[k] = cmplx(Y[FFTLEN-k].r,-1*Y[FFTLEN-k].i);
}
```

Listing 16: FFT Symmetry at the Output

Once resolved, this enhancement did indeed reduce the musical noise, especially in the audio file *Factory*. However, it made the quality of the speech worse.

3.9 Enhancement 9

Instead of rotating the noise buffers at intervals of 2.5s, a shorter period can be used to estimate the noise spectrum more quickly. In order to find the optimal rotation period, ***frame_count*** was altered and was found to be best at a value of 156 (resulting in a buffer rotation every 1.25s).

3.10 Enhancement 10 (Additional)

This enhancement is an additional enhancement and follows the Berouti's algorithm [1]. Similar to enhancement 6, the over subtraction factor $\boldsymbol{\alpha}$ is overestimated, however, it is now done dynamically according to the frequency bin's SNR value. The value of $\boldsymbol{\alpha}$ will change linearly in the range $-5dB \leq SNR \leq 20dB$ as shown below.

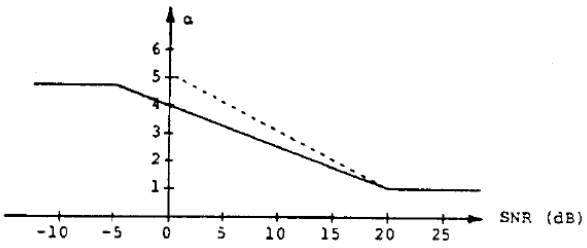

Figure 7: Berouti's Dynamic Alpha

$$\alpha = \begin{cases} \alpha_0 - \frac{1-\alpha_0}{4}, & SNR < -5dB \\ \alpha_0 + \frac{1-\alpha_0}{20} SNR, & -5dB \leq SNR \leq 20dB \\ 1, & otherwise \end{cases}$$

The value $\boldsymbol{\alpha_0}$ has been set to 4. Unlike enhancement 6, frequency bins with a high SNR value will not be as attenuated as those with low SNR value.

```
//**** Enhancement 10 ****//
else if (en10 == 1){
    SNR = 20*log10f(X[k]/Nmin[k]);
  //**** Berouti ****//
      if(SNR < -5){
            alpha = alpha_0 - (1-alpha_0)/4;
      }
      else if (SNR <= 20){
            alpha = alpha_0 + SNR*(1-alpha_0)/20;
      }
      else{
            alpha = 1;
      }
    Nmin[k] = alpha*Nmin[k];
}
```

Listing 17: Dynamic Noise Overestimation

However, the enhancement had no noticeable advantage over enhancement 6 or the basic processing algorithm. Moreover, the logarithm function is expensive, resulting in a large ***cpufrac*** value.

4. Final Speech Enhancer

In order to augment the quality of the speech enhancer as much as possible, multiple enhancements have been cascaded. After testing various combinations, the final speech enhancer chosen implements enhancements 1, 3, 4c and 9.

In order to visualise the noise reduction implemented by the final algorithm, the spectrogram of the processed signal is compared against the corrupted and clean signals. Notice how high frequencies have indeed been removed and the grainy artefacts that refer to the musical noise.

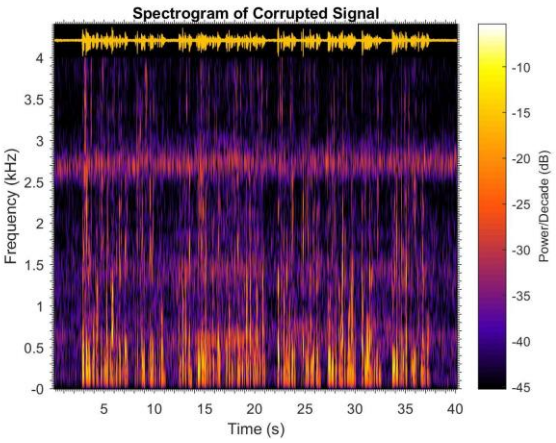

Figure 8: Corrupted Signal

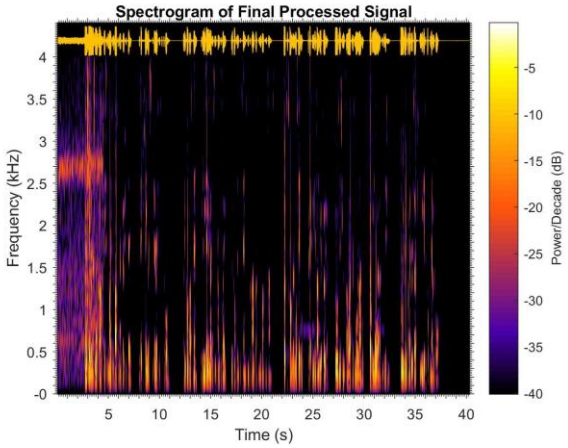

Figure 9: Processed Signal

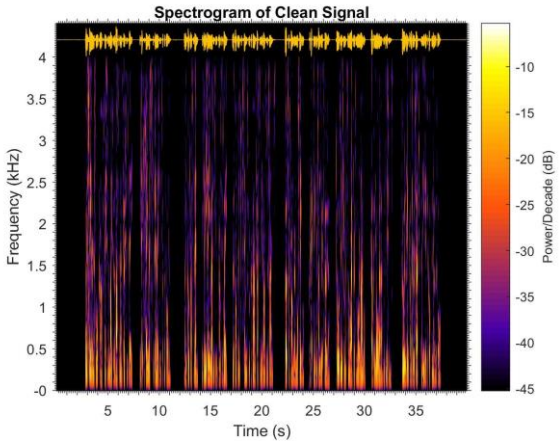

Figure 10: Clean Signal

5. Conclusion

In conclusion, the speech enhancer has not been able to completely remove the musical noise. Further algorithms could be explored to try and improve the enhancer overall, including non-linear spectral subtraction, signal subspace approaches, Wiener filtering, etc. However, a ‘perfect’ output signal is subjective and depends upon each individual. Overall, the implemented speech enhancer does not distort the speech signal and removes a large amount of background noise.

6. References

[1] Berouti, M., Schwartz, R. and Makhoul, J. (1979). *Enhancement of Speech Corrupted by Acoustic Noise*. Proc ICASSP, pp.208-211.

[2] VOCAL Technologies. (2017). *Musical Noise*. [online]. Available at: https://www.vocal.com/noise-reduction/musical-noise/ [Accessed 15th March 2019].

[3] Wikipedia. (2018). *Voice Frequency*. [online]. Available at: https://en.wikipedia.org/wiki/Voice_frequency [Accessed 19th March 2019].

[4] Boll, S.F. (1979). *Suppression of Acoustic Noise in Speech using Spectral Subtraction*. IEEE Trans ASSP 27(2), pp.113-120.

APPENDIX

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

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

              PROJECT: Frame Processing

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

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


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

*************************************************************************
***/
/*
 * You should modify the code so that a speech enhancement project is built
 *  on top of this template.
 */
/***************************** Pre-processor statements
******************************/
//  library required when using calloc
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

/* Some functions to help with Complex algebra and FFT. */
#include "cmplx.h"
#include "fft_functions.h"

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

#define WINCONST 0.85185   /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0  /* sample frequency, ensure this matches Config for AIC */
#define FFTLEN 256     /* fft length = frame length 256/8000 = 32 ms*/
#define NFREQ (1+FFTLEN/2)   /* number of frequency bins from a real FFT */
```

```
#define OVERSAMP 4     /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP) /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC) /* length of I/O buffers */

#define OUTGAIN 16000.0    /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)  /* Input gain for ADC  */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP       /* time between calculation of each frame */

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

/* Audio port configuration settings: these values set registers in the AIC23
audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
   /*************************************************************************/
   /*   REGISTER             FUNCTION         SETTINGS         */
   /*************************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB
*/\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB
*/\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB
*/\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB
*/\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost
20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off
*/\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on
*/\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit
*/\
    0x008d,  /* 8 SAMPLERATE Sample rate control        8 KHZ-ensure matches
FSAMP */\
    0x0001   /* 9 DIGACT     Digital interface activation    On
*/\
   /*************************************************************************/
};

// 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 *Y; /* output complex buffer */
complex *Yout; /* output complex buffer */
complex *Yprev; /* previous frame */
complex *Ynext; /* next frame */
float Ytemp;
float *X;     /* X(w) input magnitude spectrum */
float *P;     /* P(w) filtered input magnitude spectrum */
```

```
float *Nmin;  /* Minimum noise estimate */
float *Nt;    /* Used in filtering algorithm */
float *M1, *M2, *M3, *M4;   /* Minimum noise buffers */
float *G;   /*Final noise estimate */
float filterK;
float tau = 0.05; /*time constant of LPF */
float alpha = 3.5;  /* over subtraction factor */
int alpha_0 = 4;
float lambda = 0.02;  /*spectral floor constant */
int frame_count = 156;
int counter = 0;
float residual = 0.5;   /*residual noise limit */
float SNR;
float SNR_thres = 0.56234;
/****************************** Enhancement Switches
******************************/
int en1 = 1;
int en2 = 0;
int en3 = 1;
int en4_a = 0;
int en4_b = 0;
int en4_c = 1; //turn on en1/en2
int en4_d = 0; //turn on en1/en2
int en5_a = 0;
int en5_b = 0;
int en5_c = 0; //turn on en1/en2
int en5_d = 0; //turn on en1/en2
int en5_e = 0;
int en6 = 0;
int en7 = 0;
int en8 = 1;
int en10 = 1;
 /****************************** 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 */
/****************************** Added Function Prototypes
************************/
float max(float a, float b);   /*Function to return the maximum of two numbers*/
float min(float a, float b);  /*Function to return the minimum of two numbers*/
complex minComplex(complex a, complex b);
/********************************* Main routine
*********************************/
void main()
{

int k; // used in various for loops

//LPF
filterK = expf(-TFRAME/tau);

/*  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 */
```

```
    Y    = (complex *) calloc(FFTLEN, sizeof(complex));
    Yout    = (complex *) calloc(FFTLEN, sizeof(complex));
    Yprev     = (complex *) calloc(FFTLEN, sizeof(complex));
    Ynext    = (complex *) calloc(FFTLEN, sizeof(complex));
    X    = (float *) calloc(FFTLEN, sizeof(float)); /*  store past noise spectra
*/
    P   = (float *) calloc(FFTLEN, sizeof(float));
    Nmin   = (float *) calloc(FFTLEN, sizeof(float)); /*  Estimate of noise window
*/
    Nt   = (float *) calloc(FFTLEN, sizeof(float));
    M1    = (float *) calloc(FFTLEN, sizeof(float)); /*  store past noise spectra
*/
    M2    = (float *) calloc(FFTLEN, sizeof(float)); /*  store past noise spectra
*/
    M3    = (float *) calloc(FFTLEN, sizeof(float)); /*  store past noise spectra
*/
    M4    = (float *) calloc(FFTLEN, sizeof(float)); /*  store past noise spectra
*/
    G     = (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
*************************/
/************************ ADDED PROCESSING CODE HERE ************************/

//**** Convert Input Data into Complex Number ****//
for (k = 0; k < FFTLEN; k++){
      //**** Enhancement 8 ****//
      if (en8 == 1){
            //**** 'to visualise frame delay' ****//
            Ynext[k] = cmplx(inframe[k],0);
```

```
      }
      else {
            Y[k] = cmplx(inframe[k],0);
      }
}

//**** FFT ****//
if(en8 == 1){
      fft(FFTLEN, Ynext);
}
else{
      fft(FFTLEN, Y);
}

for (k=0;k<=FFTLEN/2;k++){
      //**** Magnitude Spectrum ****//
      if(en8 == 1){
            X[k] = cabs(Ynext[k]);
      }

      else{
            X[k] = cabs(Y[k]);
      }
      //**** Enhancement 1 ****//
      if (en1 == 1){
            //**** LPF in mag domain ****//
            P[k] = (1-filterK)*X[k] + filterK*P[k];
            X[k] = P[k];

      }
    //**** Enhancement 2 ****//
    else if (en2 == 1){
      //**** LPF in pow domain ****//
            P[k] = sqrt((1-filterK)*X[k]*X[k] + filterK*P[k]*P[k]);
            X[k] = P[k];
    }
    //**** Noise Estimation **//
    M1[k] = min(X[k], M1[k]);
      Nmin[k] = min(min(M1[k],M2[k]),min(M3[k],M4[k]));


    //**** Enhancement 6 ****//
    if (en6 == 1){
            if ((k <= 9)|| (k >= 246)){
            if(X[k]/Nmin[k] < SNR_thres){
                  alpha = alpha*10;
                  }
            else {
                  alpha = alpha;
            }
            }
            else{
            alpha = alpha;
            }
            Nmin[k] = alpha*Nmin[k];
    }

    //**** Enhancement 10 ****//
    else if (en10 == 1){
            SNR = 20*log10f(X[k]/Nmin[k]);
      //**** Berouti ****//
```

```
        if(SNR < -5){
              alpha = alpha_0 - (1-alpha_0)/4;
        }
        else if (SNR <= 20){
              alpha = alpha_0 + SNR*(1-alpha_0)/20;
        }
        else{
              alpha = 1;
        }
  Nmin[k] = alpha*Nmin[k];
  }

  else {
  Nmin[k] = alpha*Nmin[k];
  }

  //**** Enhancement 3 ****//
if (en3 == 1){
  //**** LPF in mag domain ****//
    Nt[k] = (1-filterK)*Nmin[k] + filterK*Nt[k];
    Nmin[k] = Nt[k];
}

  //**** Noise Subtraction ****//
  //**** Enhancement 4 ****//
  if(en4_a == 1){
        G[k] = max(lambda*(Nmin[k]/X[k]),1-(Nmin[k]/X[k]));
  }
  else if(en4_b == 1){
        G[k] = max(lambda*(P[k]/X[k]),1-(Nmin[k]/X[k]));
  }
  else if(en4_c == 1){
        G[k] = max(lambda*(Nmin[k]/P[k]),1-(Nmin[k]/P[k]));
  }
  else if(en4_d == 1){
        G[k] = max(lambda,1-(Nmin[k]/P[k]));
  }
  //**** Enhancement 5 ****//
  else if(en5_a == 1){
        G[k] = max(lambda,sqrt(1-(Nmin[k]*Nmin[k]/(X[k]*X[k]))));
  }
  else if(en5_b == 1){
        G[k] = max(lambda*(Nmin[k]/X[k]),1-(Nmin[k]*Nmin[k]/X[k]*X[k]));
  }
  else if(en5_c == 1){
        G[k] = max(lambda*(P[k]/X[k]),1-(Nmin[k]*Nmin[k]/X[k]*X[k]));
  }
  else if(en5_d == 1){
        G[k] = max(lambda*(Nmin[k]/P[k]),1-(Nmin[k]*Nmin[k]/P[k]*P[k]));
  }
  else if(en5_e == 1){
        G[k] = max(lambda,1-(Nmin[k]*Nmin[k]/P[k]*P[k]));
  }
  else {
        G[k] = max(lambda,1-(Nmin[k]/X[k]));
  }

  if (en8 == 1){
        Ynext[k] = rmul(G[k],Ynext[k]);
  }
  else{
```

```
            Y[k] = rmul(G[k],Y[k]);
      }

      //**** Enhancement 8 ****//
      if (en8 == 1){
            if (Nmin[k]/X[k] > residual){
                  Yout[k] = minComplex(minComplex(Yprev[k],Ynext[k]),Y[k]);
            }
            else{
                  Yout[k] = Ynext[k];
            }
            Yprev[k] = Y[k];
            Y[k] = Ynext[k];
      }
}

//**** Buffer Rotation ****//
if (counter >= frame_count){
counter = 0;
      temp = M4;
      M4 = M3;
      M3 = M2;
      M2 = M1;
      M1 = temp;

      for(k=0;k<FFTLEN ;k++){
            M1[k] = X[k];
      }
}
counter++;

//**** Symmetry Property ****//
for (k = FFTLEN/2+1; k < FFTLEN; k++){
 if(en8 == 1){
      Yout[k] = cmplx(Yout[FFTLEN-k].r,-1*Yout[FFTLEN-k].i);
 }
 else{
      Y[k] = cmplx(Y[FFTLEN-k].r,-1*Y[FFTLEN-k].i);
 }
}

//**** Output ****//
if(en8 == 1){
      //**** IFFT ****//
      ifft(FFTLEN,Yout);
      for (k=0;k<FFTLEN;k++){
      outframe[k] = Yout[k].r;
      }
}
else{
      ifft(FFTLEN,Y);
      for (k=0;k<FFTLEN;k++){
      outframe[k] = Y[k].r;
      }
}


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

```
m=io_ptr0;

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

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

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

sample = mono_read_16Bit();
inbuffer[io_ptr] = ((float)sample)*ingain;
 /* write new output data */
mono_write_16Bit((int)(outbuffer[io_ptr]*outgain));

/* update io_ptr and check for buffer wraparound */

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

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

/**** MAX ****/
float max(float a, float b){
  if(a>b)
  {
    return a;
  }
  else{
    return b;
  }
}
/**** MIN ****/
float min(float a, float b){
  if(a<b)
  {
    return a;
  }
  else{
    return b;
  }
}
/**** MIN COMPLEX ****/
complex minComplex(complex a, complex b){
if(cabs(a) <= cabs(b))
      {
            return a;
```

```
        }
        else
        {
                return b;
        }
}
```