Imperial College London

Real Time Digital Signal Processing

Project: Speech Enhancement

Authors:
Adamos Solomou
CID: 00984498
Michalis Lazarou
CID: 00936066

Supervisor:
Prof. Paul D. Mitcheson

February 14, 2018

Declaration: We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the college examination offenses policy.

Signed: Michalis Lazarou & Adamos Solomou

Contents

Abstract

Human beings have an inherent need of communicating with each other throughout the years. Nowadays, the dependence on communication systems is of great importance to everyone. Telephones are increasingly being used in noisy environments such as cars, airports and underground laboratories. Hence, the quality of sound strongly depends on the environment and level of background noise. However, it is possible to enhance the quality of the sound and eliminate any kind of noise in the speech. This project describes a technique that was developed in order to remove any background noise that might be present in the environment. To tackle this problem, spectral subtraction, a well known algorithm, was studied and implemented to eliminate the noise and transmit only the desired speech signal. Several enhancements were considered and their performance was tested in order to achieve an adequate performance. Finally, enhancements were combined in the final design and delivered as the speech enhancer system.

1 Introduction

In many real-world applications, especially in mobile communications, a reliable telecommunication system is required to ensure accurate and efficient communication between people. The quality of sound strongly depends on the location and surrounding environment. In most cases, when people communicate over mobile devices, they do so in a noisy environment. This project aims to implement a real-time system that will reduce the background noise in a speech signal while leaving the speech signal itself intact. This process is generally known as *speech enhancement*.

2 Basic Algorithm

2.1 Spectral Subtraction

Noise signals are in general random and unknown, hence the problem cannot be solved by simply suppressing the frequency band of the noisy component. Many different algorithms have been proposed for speech enhancement, but in the context of this project, focus will be given on the algorithm known as *spectral subtraction*. The technique operates under the assumption that the noise signal $n[n]$ is additive to the speech signal $y[n]$. If this holds, then the noisy speech $x[n]$ can be expressed as:

$$x[n] = y[n] + n[n], \quad for \quad 0 \leq n \leq N-1$$

where n is the time index in the digital domain and $N = 256$ is the number of samples. A more detailed explanation for the selection of N is given in section 2.2. The time domain signals can be expressed in the frequency domain as:

$$Y(w) = X(w) - N(w)$$

where $Y(w)$, $X(w)$, $N(w)$ are the spectra of the output signal (filtered speech), input signal (noisy speech) and noise signal respectively. Ideally, $Y(w)$ should be identical to the clear from noise speech signal spectrum. All of the processing will be performed in the frequency domain using frame processing. The procedure is simple and is illustrated in the following diagram. The Fourier Transform of the input signal is first calculated, then the estimated noise spectrum is subtracted and finally, the filtered spectrum is converted back into time domain.

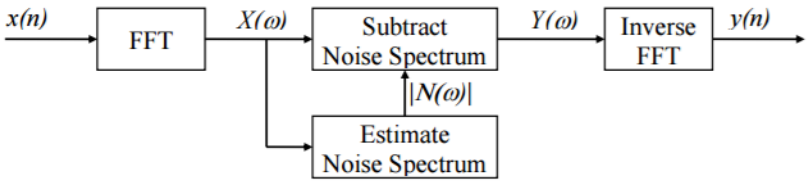

Figure 1: Block diagram illustration spectral subtraction algorithm. Image taken from P. D. Mitcheson

2.2 Frame Processing

To perform frequency domain processing in real time, it is necessary to split the continuous time-domain input signal into a sequence of non-overlapping chunks that will then be processed. These small chunks are called frames. Once the frames are processed, the output signal can be reconstructed by the individual frames. However, there is a problem with this technique. If the FFT

is calculated on a frame, discontinuities at the frame boundaries will give rise to spectral artefacts at the frequency domain. Thus, the FFT is not a true representation of the frequencies present in the continuous input stream from the time domain. Essentially, additional unwanted frequencies have been introduced by splitting the input signal into frames. A way of avoiding spectral artefacts is to multiply the frames by a window, which will smoothly reduce the amplitude of the frame to zero near the edges. The following diagram illustrates this concept, multiplying the frames with a Hamming window.

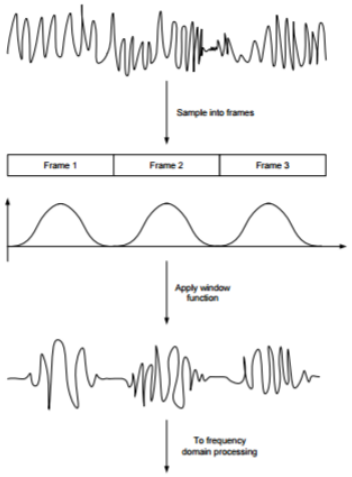

Figure 2: Simple frame processing and windowing. Image taken from P. D. Mitcheson

The diagram above illustrates that while trying to avoid spectral artefacts by introducing the window function, the amplitude of the signal now varies, explicitly changing the original continuous time signal.

A possible way of solving this problem is to use overlap-add processing. The idea is to now split the continuous time-domain signal into overlapping frames and apply overlapping windows, which when added will give the original signal. If the frames overlap in such a way so that the envelope of the overlapping windows always adds to 1, frequency domain processing can be performed on each individual frame without having spectral artefacts and without losing power in the signal. The following diagram illustrates the overlap-add processing with an overlap of half a frame.

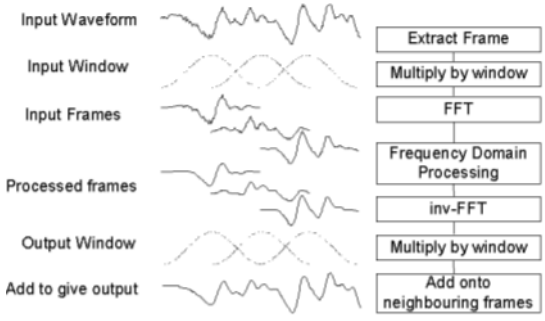

Figure 3: Overlap-add processing Image taken from P. D. Mitcheson

In the diagram above, the input signal is multiplied by three overlapping windows and three overlapping frames are obtained. Each frame is processed separately in the frequency domain and

then converted back to time domain by calculating the inverse-FFT. The processed time domain frames are then multiplied by the same window. This is done since the processed frames may have discontinuities in the time domain which would lead to crackles in the output sound. The frames are then added to obtain the filtered, output signal. Hence, the output signal is formed by adding together a continuous stream of 256-sample frames each of which has been multiplied by both an input and an output window. So far, an overlap of half a frame was assumed. This results to an *oversampling ratio* of 2. This normally gives acceptable results but may introduce distortion. An oversampling ratio of 4 will then be used in which each frame starts a quarter of a frame after the previous one, hence windows overlap by 75%. By choosing the windows to be the square root of a Hamming window, $w[k] = \sqrt{1 - 0.85185 cos((2k+1)\pi/N}$ for $k = 0, ..., N-1$, and using an oversampling ratio of 4, then the overlapped windows will sum to a constant and the output signal will be undistorted by the frame processing. This is a desired result.

As mentioned above, the frame length N is set to 256 which is an integer power of 2. This enables a more efficient calculation of the DFT by means of the FFT. The FFT only requires $N/2 \times log_2 N$ complex multiplies instead of N^2 required by the simple DFT algorithm. It should be noted that there is a trade off in the decision of frame length. Considering that each frequency bin is sampled once per pair, then long frames result to good frequency resolution but poor time resolution and vice-versa. Long frames also require more processing. Since the sampling frequency is set to $f_s = 8kHz$, then $N = 256$ corresponds to a time interval of $32ms$ in time domain. This choice of N provides a good compromise in the above trade off but also speech signals can be considered to be quasi-stationary over short periods of time of the order $30ms$. Hence this is a generally good choice.

2.3 Noise Subtraction

As stated above, the basic idea is to subtract the noise spectrum from the input signal spectrum:

$$Y(\omega) = X(\omega) - N(\omega)$$

where $X(\omega)$ is the original spectrum of the input signal, $N(\omega)$ is the estimate of the noise spectrum and $Y(\omega)$ is the resulting filtered spectrum. Since the correct phase of the noise signal is unknown, only the magnitude of $N(\omega)$ is subtracted from $X(\omega)$, leaving the phase of $X(\omega)$ unchanged:

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

where $g(\omega)$ is a frequency-dependent gain factor, hence the above filtering is just a form of zero-phase filtering.

A possible problem with this method is that $g(\omega)$ can go negative. To ensure that this does not occur, an alternative solution is considered given by the following formula:

$$g(\omega) = max\Big(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\Big)$$

where λ is between 0.01 and 0.1. This is done because the spectral estimation and subtraction technique gives rise to what is known as "musical noise". This sounds as if rapid random musical notes are playing in the background of the enhanced signal. The musical noise is worse if $\lambda = 0$ and is better if λ is just above zero.

2.4 Noise Estimation

In order to perform the spectral subtraction, an estimate of the noise spectrum must be calculated. A way to do that would be do design a Voice Activity Detector (VAD) which identifies whether or not speech is present in a signal. However, a reliable VAD is very hard to make and an easier approach is chosen. The pause during human speech could be used as the estimate of the background noise. Every human when speaking takes breaks in order to breathe. It is assumed that this happens every 10 seconds and under this assumption the spectral minimum over 10 seconds is taken to be the minimum noise amplitude. However, this will underestimate the average noise magnitude therefore a compensating factor will multiply this minimum to yield an average magnitude.

An efficient way to store and compare the spectral magnitudes over 10 seconds must be used. One option could be to store all the speech spectra over the last 10 seconds. However, this requires to store 1250 frames of data since 32ms per frame means 312.5 frame lengths in 10 seconds and with oversampling ratio of 4 means 1250 frames of data. This requires storage of 1250 FFT frames, each 256 samples long. A more computationally efficient method is to calculate and store the spectral minimum over four 2.5 second intervals. Even though this is less accurate, it requires less storage and processing.

Four buffers, M_i, where $i = 1, 2, 3, 4$, store the minimum spectrum which occurred in the 2.5 second intervals. For example at $t = 9.0$ seconds M_4 stores the minimum spectrum from 0 - 2.5 s, M_3 from 2.50 - 5.00 s, M_2 from 5.00 - 7.5 s and M_1 from 7.5-9s. The estimate of the noise spectrum is found by:

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

where α is the compensating factor for the underestimation of the noise. Every 2.5 seconds the contents of M_i are transferred to M_{i+1} and the contents of $M_4(\omega)$ are transferred to $M_1(\omega)$. Note that as the buffers are copied around, the oldest data, currently in $M_1(\omega)$, (the old $M_4(\omega)$) is overwritten. Then $M_1(\omega)$ is updated continuously by:

$$M_1(\omega) = min(|X(\omega)|, |M_1(\omega)|)$$

2.5 Code Implementation

The spectral subtraction algorithm described above was implemented using C-language. Focus is given on the FFT calculation, noise estimation and spectral subtraction.

To allow the calculation of the FFT, the windowed samples are copied into a complex structure named `intermediate`. This is to accommodate the fact that the Fourier Transform is in general complex and the length of the array is set to $N = 256$. Then the magnitude spectrum, $|X(\omega)|$, is computed from the calculated FFT, taking into consideration the symmetry properties of the FFT. It is known that for real valued time-domain signals, the DFT is complex conjugate symmetric. That is $X(N-k) = X^*(k)$. Hence only half of the magnitude spectrum samples need to be stored, since the other half can be retrieved using the symmetry of FFT. This is significantly important since it reduces memory requirements. Also processing will be performed only on half the samples reducing significantly the required processing power and time which is extremely important for real-time processing applications.

```
//Perform FFT over the intermediate array
fft(FFTLEN,intermediate);

//Calculate |X(w)|, magnitude spectrum of input signal
for (k=0;k<NFREQ;k++) //loop over the number of frequency bins
{
    //calculate the absolute value/magnitude spectrum
    X_magnitude[k] = cabs(intermediate[k]);
}
```

Figure 4: FFT and magnitude spectrum calculation

The next step involves estimating the noise spectrum. This essentially involves calculating the quantity $min_{i=1,...4}(M_i(\omega))$ as described above. First, the function $M_1(w) = min(|X(\omega)|, M_1(\omega))$ needs to be implemented for each frame. The algorithm first checks if the frame index has been reset to zero, and if so it sets the value of the buffer M_1 to the current magnitude. Otherwise, it examines the current magnitude with the value already stored in M_1. It selects the minimum out of them and saves it in M_1. Then the minimum over the four $M_i(\omega)$ is found and stored in the variable `global_min`. This is then used to estimate the noise spectrum according to $|N(\omega)| = \alpha min_{i=1,...4}(M_i(\omega))$, where $min_{i=1,...4}(M_i(\omega))$ is stored in `global_min`. This is multiplied by the correction factor α which for the basic algorithm was set to $\alpha = 20$.

```
    for(k=0;k<NFREQ;k++)
    {
        //Select the minimum
        if (f_index == 0){
            M1[k] = X_magnitude[k]; //For the first frame set the minimum as the
current sample
        }
        else{
            M1[k] = find_min(M1[k],X_magnitude[k]); //for any other frame, compare the
current sample with the stored minimum
        }

        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) );
        //Calculate the noise estimate considering the global minimum magnitude and
        //estimate noise spectrum using correction factror alpha
        noise_estimate[k] = alpha_basic*global_min;
    }
```

Figure 5: Noise estimation

The frame index is incremented each time the frame processing function is invoked. Once 2.5s have passed, the buffers M_1 to M_4 are rotated and the oldest is discarded. This is performed using a counter that counts up to 312 and once this value is reached the buffers are rotated. This is the number of frames that can be processed within 2.5s for an oversampling ratio of 4 and $f_s = 8kHz$, since $(2.5/0.032) \times 4 = 78.125 \times 4 = 312.5$, where 0.032s is the processing time for a single frame.

```
if (++f_index > (ROTATE_TIME/FRAMEINC*FSAMP)){

    f_index = 0;
    tmp_M4 = M4;
    M4 = M3;
    M3 = M2;
    M2 = M1;
    M1 = tmp_M4;

}
```

Figure 6: Buffer rotation

Next, the frequency dependent gain factor $g(\omega)$ needs to be calculated. This is done by comparing λ and $1 - \frac{|N(\omega)|}{|X(\omega)|}$ and selecting the maximum argument, where λ is set to 0.1. This is then used to subtract the estimated noise spectrum by implementing the zero-phase filtering $Y(\omega) = X(\omega) \times g(\omega)$. Finally, the result is stored back into the `intermediate` array and the symmetry of FFT is again used to fill the rest of the array.

```
//Subtract noise spectrum
for (k=0;k<NFREQ;k++)
{
    //calculate frequency dependent gain factor. Prevent frin going negative
    g = find_max((lambda*1.0), (1-(noise_estimate[k]/X_magnitude[k])));

    //implement zero-phase filter
    //Using symmetry in FFT to fill the values [FFTLEN-k]
    intermediate[k] = rmul(g, intermediate[k]);
    intermediate[FFTLEN - k].r = intermediate[k].r;
    intermediate[FFTLEN - k].i =- intermediate[k].i;
}
```

Figure 7: Noise subtraction

Finally, the inverse FFT of the complex structure `intermediate` is taken and the result is written to the output frame.

```
iff=(FFTLEN,intermediate);
//write to the output frame
for (k=0;k<FFTLEN;k++)
{
    outframe[k] = intermediate[k].r;
}
```

Figure 8: Inverse-FFT and writing result to the output

3 Evaluation of the Basic Algorithm and Enhancements

The performance of the basic algorithm will now be evaluated using the "Sailor Passage" corrupted by noise from different environments. This a passage used for accent classification in English. Several enhancements that researches have suggested are also implemented and their performance is also evaluated. The C-code containing the implementation of all the enhancements along with the basic algorithm can be found in Appendix 1.

3.1 Basic algorithm

The basic algorithm works reasonably well only with high compensating factor $\alpha = 20$. In general it copes well with the low noise signals but does not work well with "lynx2" and "phantom4". In general noise is not removed completely with the basic algorithm. Since there are clear trade offs between background noise and musical noise the removal of one type of noise gives rise to the other type in the basic algorithm.

3.2 Enhancement 1

For this enhancement a low-pass filtered version of $|X(\omega)|$ is used when calculating the noise estimate.

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

where $k = exp(-T/\tau)$ is the z-plane pole for constant τ and frame rate T. This enhancement takes advantage of the fact that the human ear is more sensitive to lower frequencies, hence the low pass filter will filter out the high frequency components of the noise while leave human voice unchanged. Also using this enhancement the value of α was decreased and after testing with different recordings the optimum value of α was chosen to be 2. An echo was introduced when the value of τ was set smaller and k was close to 1. Empirically the optimum value of τ was set to 25ms which lies within the suggested range of 20ms to 80ms.

3.3 Enhancement 2

This enhancement is very similar to the previous one but it is performed in the power domain, using a low pass filter for $|X(\omega)|^2$.

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

The optimum value of α that produced the clearer voice was 3 and overall the performance of this enhancement is better than enhancement 1. This is because calculations based in the power domain are better because of the way humans hear. This enhancement allows the average noise power to be subtracted rather than the average noise amplitude.

3.4 Enhancement 3

It is suggested to low pass filter the noise estimate $|N(\omega)|$ to avoid abrupt discontinuities when the minimization buffers rotate. If $N_{LP(t)}(\omega)$ is the low-pass filtered noise estimate for frame t, then

$$N_{LP(t)}(\omega) = (1-k_n) \times |N(\omega)| + k_n \times N_{LP(t-1)}$$

where $k_n = exp(-T/\tau_n)$. The time constant τ_n was empirically set to 0.08 (80ms). It is suggested that this enhancement will have noticeable effect if the noise level is very variable. A noticeable noise reduction was observed only for test signals "factory 1,2". Hence these environments will be considered as noise-varying.

3.5 Enhancement 4

For this enhancement various versions of the frequency-dependent gain factor were designed and compared against each other:

- $g(\omega) = max\Big(\lambda\frac{N(\omega)}{X(\omega)}, 1 - \frac{|N(\omega)|}{|X(\omega)|}\Big)$
- $g(\omega) = max\Big(\lambda\frac{P(\omega)}{X(\omega)}, 1 - \frac{|N(\omega)|}{|X(\omega)|}\Big)$
- $g(\omega) = max\Big(\lambda\frac{N(\omega)}{P(\omega)}, 1 - \frac{|N(\omega)|}{|P(\omega)|}\Big)$
- $g(\omega) = max\Big(\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|}\Big)$

It was found through testing that the best choice for $g(\omega)$ was the third version that produced the cleaner sound. This enhancement was implemented using a switch statement and each time the wanted version for $g(\omega)$ is chosen. The best value for α was found empirically to be 3 and τ to be 80ms. The optimum τ for this enhancement is larger than the τ used in enhancement 1 as suggested in [1].

3.6 Enhancement 5

The enhancement above was redesigned in order to operate in the power domain and gives rise to the equations below:

- $g(\omega) = max\Big(\lambda\frac{N(\omega)}{X(\omega)}, \sqrt{1-\frac{|N(\omega)|^2}{|X(\omega)|^2}}\Big)$
- $g(\omega) = max\Big(\lambda\frac{P(\omega)}{X(\omega)}, \sqrt{1-\frac{|N(\omega)|^2}{|X(\omega)|^2}}\Big)$
- $g(\omega) = max\Big(\lambda\frac{N(\omega)}{P(\omega)}, \sqrt{1-\frac{|N(\omega)|^2}{|P(\omega)|^2}}\Big)$
- $g(\omega) = max\Big(\lambda, \sqrt{1-\frac{|N(\omega)|^2}{|P(\omega)|^2}}\Big)$

As before, this enhancement was implemented using a switch statement and each time a version of $g(\omega)$ is selected. Through testing it was found that the operation in the power domain for calculating the frequency-dependant gain factor produced worse results. Similarly as before empirically the best values for α was set to be 3 and τ to be 80ms.

3.7 Enhancement 6

It is suggested to deliberately overestimate the noise level by increasing the value of α at the low frequency bins that have a poor signal to noise ratio (SNR). This technique is known as *oversubtraction* and can reduce the "musical noise" artifacts that are introduced by spectral subtraction. As suggested in [3], the correction factor α should vary with respect to the SNR according to the relationship shown below.

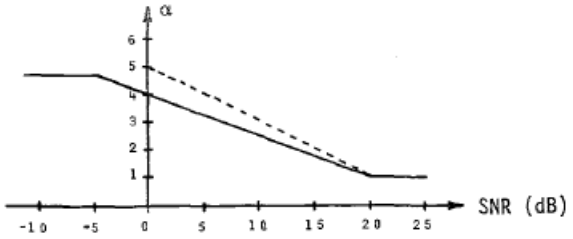

Figure 9: Value of the coefficient α versus SNR. Image taken from [3]

The above relation is implemented and the factor α is set according to:

$$\alpha = \begin{cases} 4.9 & \text{if } SNR < -5dB \\ 1.0 & \text{if } SNR > 20dB \\ 4 - \frac{20}{3}SNR & if\ -5dB \leq SNR \leq 20dB \end{cases}$$

where $SNR = 20 \times log\Big(\frac{|X(\omega)|}{|N(\omega)|}\Big)$. No noise reduction could be noticed on test signals. This is because α is very small to cause any noise reduction. An improved version of this enhancement is implemented in enhancement 10.

3.8 Enhancement 7

This enhancement suggests to evaluate different frame lengths. This approach was tested on enhancements 2,3 and 4. The frame length was changed by varying the value of the parameter `FFTLEN`. It was halved and then doubled. The value of time frame, controling the rotation of buffers, changed to 626 and 156 respectively in order to adapt to the buffer rotation every 2.5s. It was observed that the shorter frame produced rough sound with increased musical noise whereas the longer frame reduced the musical noise but sound slurred. This confirms the proposal by [3] that "using an analysis frame shorter than 20ms results roughness" and "if the frame is too long, slurring results". Hence it was decided to keep the frame length equal to $32ms$.

3.9 Enhancement 8

The purpose of this enhancement is to reduce musical noise by applying the "residual noise reduction" method described in [4]. This can be achieved if $\frac{|N(\omega)|}{|X(\omega)|}$ exceeds some threshold then $Y(\omega)$, (which is equated to the FFT values of the intermediate buffer), is replaced by the minimum calculated value in three adjacent frames. This enhancement works because if the signal to noise ratio is above the threshold value, this can be either because possible of the presence of musical noise or because of a low power speech signal with noise that corresponds to high SNR. If the SNR is high due to the presence of musical noise, by choosing the minimum value of noise across the three frames, musical noise would be discarded efficiently. This is because musical noise is represented by peaks in the frequency spectrum that vary across different frames. However if this is due to low power speech, since speech is stationary across a small amount of frames speech would be retained. The algorithm can be implemented as shown below:

$$Y(\omega) = min(Y_{t-1}(\omega), Y_t(\omega), Y_{t+1}(\omega)) for \frac{|N(\omega)|}{|X(\omega)|}$$

3.10 Enhancement 9

It is suggested to estimate the noise by taking the minimum spectrum over a shorter period, thus change the time before buffer rotation occurs. This is currently set to 2.5s. To set it to 2s, $2sec \times \frac{f_s}{N} \times$oversampling$= 2 \times \frac{8000}{256} \times 4 = 250$. Hence, 250 frames need to be processed before shifting. To set the rotation time to 1.5s, 188 need to elapse. This enhancement was tested and indeed it was found that this change made the system to response faster to a rise in noise level but introduced distortion in the speech. This is because taking the noise estimate over a shorter period allows the estimate to be found more quickly. However, if the speaker continues speaking for more than the length of the shorter buffer period, higher noise estimates will be made resulting in distortion to the speech signal after the noise is removed.

3.11 Enhancement 10

The performance of enhancement 6 will now be improved. Firstly, the `log10f()` function available from the `math.h` C library will not be used since this proved to be computationally expensive. Hence instead of using the logarithmic scale in dB, the SNR in the linear scale will be used, that is the ratio $\frac{|X(\omega)|}{|N(\omega)|}$. To accommodate this, the SNR thresholds used before in dB must be converted to the equivalent values in the linear space. The same is applied for the linear relationship between α and SNR_{dB}. The linearity only exists in the log scale. Hence the previous relationship is translated

to:

$$\alpha = \begin{cases} 4.9 & \text{if } SNR < -0.5 \\ 1.0 & \text{if } SNR > 10 \\ 3.5 + 2.5 \times SNR & if \ -0.5 \leq SNR \leq 10 \end{cases}$$

where $SNR = \frac{|X(\omega)|}{|N(\omega)|}$. Testing this enhancement on test signals confirmed the initial proposal that this oversubtraction technique can reduce the musical noise artifacts introduced by spectral subtraction.

4 Enhancement Combinations and Performance

Several enhancement combinations were explored in order to achieve a good performance for the speech enhancement. Enhancements 5,6,7 and 9 were not considered due to their poor performance on the test signals. Alternative considerations include but are not limited to:

- Enhancement 1 and 2 cannot be combined since method 1 operates in time domain whereas method 2 operates in power domain.
- Enhancement 4 can be combined with either 1,2 or 3

Different combinations were tested. Through trial and error, it was found that the best performance is achieved using the combination of enhancements 2,3 and 4 with the coefficient $\alpha = 3$ and $\lambda = 0.0003$. The equation $g(\omega) = max\Big(\lambda\frac{N_{LP}(\omega)}{P(\omega)}, 1 - \frac{|N_{LP}(\omega)|}{|P(\omega)|}\Big)$, with $\tau = 80ms$ was selected from enhancement 4. This combination implies that $|X(\omega)|$ is low-pass filtered in the frequency domain to yield $P(\omega)$. Also, the noise estimate $|N(\omega)|$ is low pass filtered and the combination of the two was used to evaluate the frequency dependent gain factor $g(\omega)$. The specified combination was tested using all the available test signals and the performance was very good apart from some speech distortion encountered with the test signal "phantom 4". The C-code containing the combination of the enhancements can be found in Appendix 2. It is worth noting that since the combination of enhancements was determined, all the switch or if statements were removed in order to avoid unnecessary computation and speed up the program. Recall that an if (or switch) statement corresponds to a conditional branch in Assembly Language which may cause the pipeline to stall hence additional instruction cycles are introduced slowing down the program.

To assess the performance of the implemented filter, spectrogram plots of the corrupted and filtered signals were obtained. These are shown in the rest of the section.

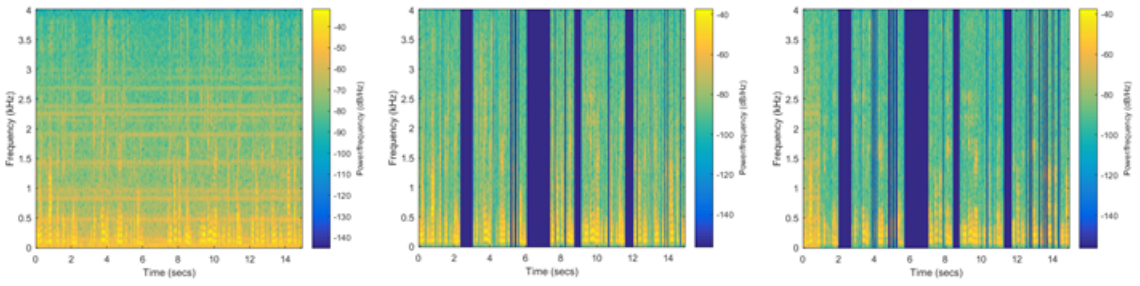

Figure 10: Spectrograms for "Lynx 1" *Left:* Corrupted signal *Middle:* Clean Signal *Right:* Filtered signal

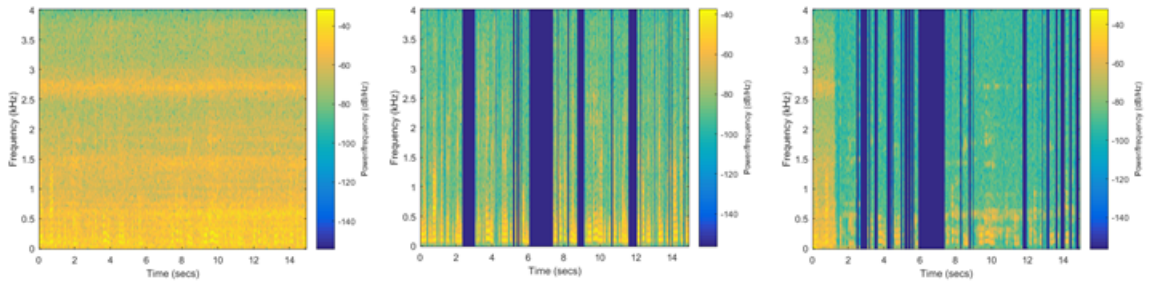

Figure 11: Spectrograms for "Phantom 4" *Left:* Corrupted signal *Middle:* Clean Signal *Right:* Filtered signal

Figure 10 shows the spectrogram for the corrupted, clean and noisy signal for test signal "lynx1". It can be clearly seen that the filtered spectrogram is almost the same as the clean one indicating the functionality of the system. The dark blue lines in the clean spectrogram correspond to the time interval where the human breathes. Since there is no signal observed at that interval the power is low and this is reflected by the dark blue colour. The implemented system correctly estimates and subtracts the noise spectrum, hence the output signal has almost the same spectrogram as the clean one. A similar pattern is observed in figure 11 for the test signal "phantom 4". Again the filter spectrogram approaches the clean one. Indeed the filtered spectrogram is not the same as the clean one since this test signal was heavily corrupted by noise. This is reflected by the dark yellow colour in the spectrogram of the corrupted signal. Spectrograms for test signals "lynx1" and "factory1" can be found in Appendix 3. It should be noted that there is some "delay" between the dark blue lines in the clean and filtered periodogram. This is due to the fact that the spectrograms were obtained manually, hence there is some miss-alignment in their starting point.

5 Conclusion

This report provides a guide to the implementation of a real time speech enhancement algorithm based on spectral subtraction along with some additional enhancements. Additional considerations like the efficiency of the algorithm were consider which are of extreme importance for real-time processing applications. Assuming that the speaker pauses to take a breath, the noise estimation can be very successful. There was a lag delay between the start of the noisy speech and the moment when the spectral subtraction was enabled. This was managed to be reduced to 6s. The performance of the implemented system was assessed by listening tests and spectrograms. The system operates very well under various noisy environments without distorting the speech signal. Test signals like "car 1", "factory 1-2", "lynx 1-2" and "phantom 1" were completely cleaned from noise, with some musical noise observed for some signals like "phantom 4". The spectral subtraction algorithm is successful.

6 References

1. Mitcheson, P., (2017), Real-Time Digital Signal Processing Project: Speech Enhancement[PDF]. Retrieved from http://learn.imperial.ac.uk

2. Mitcheson, P., (2017), Real-Time Digital Signal Processing Section 8 - Speech Enhancement[PDF]. Retrieved from http://learn.imperial.ac.uk

3. Berouti, M., Schwartz, R., and Makhoul, J. Enhancement of speech corrupted by acoustic noise. In Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '79, volume 4, pages 208-211, Apr 1979

4. Boll, S., Suppression of acoustic noise in speech using spectral subtraction. Acoustics, Speech and Signal Processing, IEEE Transactionson, 27(2):113-120, Apr 1979.

5. Lockwoord, P. Boudy, J., "Experiments with a Nonlinear Spectral Subtractor (NSS), Hidden Markov Models and the projection, for robust speech recognition in cars", Speech Communication, 11, pp215-228, Elsevier 1992.

6. Martin, R., "Spectral Subtraction Based on Minimum Statistics", Signal Processing VII: Theories and Applications, pp1182-1185, Holt, M., Cowan, C., Grant, P. and Sandham, W. (Eds.), 1994

7 Appendix

7.1 Appendix 1: Basic Algorithm & Enhancements Code

H:\RTDSPlab\project_pt2\RTDSP\enhance.c 24 March 2017 22:53

```
/*************************************************************************************
                    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 *X_magnitude;             /* Array to hold the magnitudes of frames currently
processed*/
//ENHANCEMENT 8*********************************
float *Y_8;
complex *Y_minus1;
complex *Y_minus2;
complex *Y_current;
complex *f_switch;
//***************************************************
float *Pt;
float *Pt_prev;
float *low_pass_noise;
float *low_pass_noise_prev;
float *M1;
float *M2;
float *M3;
float *M4;
float *tmp_M4;
float *noise_estimate;
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 */
volatile int f_index=-1;            /* Counts until 312 corresponding to 2.5 secs*/

complex *intermediate;              //Array to hold the processing samples
float alpha1 = 2;
float alpha_basic = 20;                     //correction factor for noise estimation
float lambda = 0.1;
```

```
float g;
float tau = 0.025; //set tau=25ms
float tau_4 = 0.06;//tau for enhancement 4
float noise_tau3 = 0.08; //set tau=80 ms for enhancement 3
float kappa; //kappa variable for enhancement 1,2
float kappa_noise; //kappa variable for enhancement 3
float kappa_4;
float global_min;
float ROTATE_TIME = 2;

//******ENHANCEMENT 8**********
int frame;
float Y_min;
float residualth =0.6;

//Choose enhancements
int mode_4 = 2;
int mode_5 = 2;
int choose_enhancement = 4;

 /****************************** Function prototypes ******************************/
void init_hardware(void);       /* Initialize codec */
void init_HWI(void);            /* Initialize hardware interrupts */
void ISR_AIC(void);             /* Interrupt service routine for codec */
void process_frame(void);       /* Frame processing routine */
void basic(void);
void enhancement1(void);
void enhancement2(void);
void enhancement3(void);
void enhancement4(void);
void enhancement5(void);
void enhancement8(void);
void enhancement6(void);
void enhancement10(void);
float find_min(float a, float b);
float find_max(float a, float 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(NFREQ, sizeof(float));   /* M1 */
    M2                      = (float *) calloc(NFREQ, sizeof(float));   /* M2 */
    M3                      = (float *) calloc(NFREQ, sizeof(float));   /* M3 */
    M4                      = (float *) calloc(NFREQ, sizeof(float));   /* M4 */
    intermediate            = (complex *) calloc(FFTLEN, sizeof(complex));  /* complex
```

```
    buffer */
    X_magnitude             = (float *) calloc(NFREQ, sizeof(float));   /* magnitude
    spectrum */
    low_pass_noise          = (float *) calloc(NFREQ, sizeof(float));   /* low_pass_noise
    estimate */
    low_pass_noise_prev     = (float *) calloc(NFREQ, sizeof(float));   /* low_pass_noise
    estimate previous */
    noise_estimate          = (float *) calloc(NFREQ, sizeof(float));
    Pt                      = (float *) calloc(NFREQ, sizeof(float));   /* Pt */
    Pt_prev                 = (float *) calloc(NFREQ, sizeof(float));   /* Pt_prev */

    //ENHANCEMENT 8
    Y_minus1                = (complex *) calloc(FFTLEN, sizeof(complex)); /* Array holding
    previous Y(n-1) frame ' s output*/
    Y_minus2                = (complex *) calloc(FFTLEN, sizeof(complex)); /* Array holding
    Y(n-2) frame ' s output */
    f_switch                = (complex * ) calloc(FFTLEN, sizeof(complex)) ; /* Array to f a
    c i l i t a t e the pointer switch */
    Y_current               = (complex * ) calloc(FFTLEN, sizeof(complex)) ; /* Array
    holding current output */
    Y_8                     = (float *) calloc(NFREQ, sizeof(float));

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

    /* initialize hardware interrupts */
    init_HWI();

/* initialize algorithm constants */
    kappa       = exp(-TFRAME/tau); //kappa for enhancement 1,2
    kappa_noise = exp(-TFRAME/noise_tau3); //kappa for enhancement 3
    kappa_4     = exp(-TFRAME/tau_4);

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

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

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

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

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

```
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

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

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


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

}

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

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

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

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

    //Check for time condition and rotate buffers when necessary
    if (++f_index > (ROTATE_TIME/FRAMEINC*FSAMP)){

        f_index = 0;
        tmp_M4 = M4;
        M4 = M3;
        M3 = M2;
        M2 = M1;
        M1 = tmp_M4;

    }

    /* 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];
        intermediate[k] = cmplx(inframe[k],0); //copy samples from inframe to intermediate
        array
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }

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

    //Perform FFT over the intermediate array
    fft(FFTLEN,intermediate);

    //Calculate |X(w)|, magnitude spectrum of input signal
    for (k=0;k<NFREQ;k++) //loop over the number of frequency bins
    {
        //calculate the absolute value/magnitude spectrum
        X_magnitude[k] = cabs(intermediate[k]);
    }
    //Implement software switches
    switch(choose_enhancement)
    {
        case(0):
        basic();
        break;

        case(1):
        enhancement1();
        break;

        case(2):
        enhancement2();
        break;

        case(3):
        enhancement3();
        break;

        case(4):
        enhancement4();
        break;

        case(5):
        enhancement5();
        break;

        case(6):
        enhancement6();
        break;

        case(8):
        enhancement8();
        break;

        case(10):
        enhancement10();
```

```
        break;

    }

    //Perform inverse fft
    if(choose_enhancement != 8){
        ifft(FFTLEN,intermediate);
        //write to the output frame
        for (k=0;k<FFTLEN;k++)
        {
            outframe[k] = intermediate[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;
}

/************************************************************************/
void basic(void)
{
    int k;
    for(k=0;k<NFREQ;k++)
    {
```

```
        //Select the minimum
        if (f_index == 0){
            M1[k] = X_magnitude[k]; //For the first frame set the minimum as the current
            sample
        }
        else{
            M1[k] = find_min(M1[k],X_magnitude[k]); //for any other frame, compare the
            current sample with the stored minimum
        }

        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) );
        //Calculate the noise estimate considering the global minimum magnitude and
        //estimate noise spectrum using correction factror alpha
        noise_estimate[k] = alpha_basic*global_min;
    }

    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        //calculate frequency dependent gain factor. Prevent frin going negative
        g = find_max((lambda*1.0), (1-(noise_estimate[k]/X_magnitude[k])));

        //implement zero-phase filter
        //Using symmetry in FFT to fill the values [FFTLEN-k]
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }

}
    /*****************************************************************************/

void enhancement1(void)
{
    //Usimg low pass filter version of |X(w)|
    int k;

    for(k=0;k<NFREQ;k++)
    {
        Pt[k] = (1 - kappa)*X_magnitude[k] + kappa*Pt[k];

        //Select the minimum
        if (f_index == 0){
            M1[k] = Pt[k]; //For the first frame set the minimum as the current sample
        }
        else{
            M1[k] = find_min(M1[k],Pt[k]); //for any other frame, compare the current sample
            with the stored minimum
        }

        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min
        noise_estimate[k] = alpha1*global_min;
    }
```

```
    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        g = find_max((lambda*1.0), (1-(noise_estimate[k]/Pt[k])));

        //implement zero-phase filter
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
}

void enhancement2(void)
{
    //Usimg low pass filter version of |X(w)|
    int k;

    for(k=0;k<NFREQ;k++)
    {
        Pt[k] = sqrt(((1 - kappa)*X_magnitude[k]*X_magnitude[k] +
        kappa*Pt_prev[k]*Pt_prev[k]));

        //Select the minimum
        if (f_index == 0){
            M1[k] = Pt[k]; //For the first frame set the minimum as the current sample
        }
        else{
            M1[k] = find_min(M1[k],Pt[k]); //for any other frame, compare the current sample
            with the stored minimum
        }

        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min
        noise_estimate[k] = alpha1*global_min;
    }

    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        g = find_max((lambda*1.0), (1-(noise_estimate[k]/Pt[k])));

        //implement zero-phase filter
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
    Pt_prev = Pt;
}

void enhancement3(void)
{
    //Usimg low pass filter version of |X(w)|
    int k;
    float alpha1;

    for(k=0;k<NFREQ;k++)
```

```
    {
        //Select the minimum
        if (f_index == 0){
            M1[k] = X_magnitude[k]; //For the first frame set the minimum as the current
            sample
        }
        else{
            M1[k] = find_min(M1[k],X_magnitude[k]); //for any other frame, compare the
            current sample with the stored minimum
        }

        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min
        noise_estimate[k] = alpha1*global_min;

        //low pass filter the noise to avoid discontinuities when the minimization buffers
        rotate
        low_pass_noise[k] = (1-kappa_noise)*noise_estimate[k] +
        kappa_noise*low_pass_noise_prev[k];
    }

    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        g = find_max((lambda*1.0), (1-(low_pass_noise[k]/X_magnitude[k])));

        //implement zero-phase filter
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
    low_pass_noise_prev = low_pass_noise;
}

void enhancement4(void){
    int k;

    for(k=0;k<NFREQ;k++)
    {
        Pt[k] = sqrt(((1-kappa_4)*(X_magnitude[k]*X_magnitude[k]) +
        kappa_4*Pt_prev[k]*Pt_prev[k])); // P
        //Select the minimum
        if (f_index == 0){
            M1[k] = Pt[k]; //For the first frame set the minimum as the current sample
        }
        else{
            M1[k] = find_min(M1[k],Pt[k]); //for any other frame, compare the current sample
            with the stored minimum
        }
        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min
        noise_estimate[k] = alpha1*global_min;
    }
    for(k=0; k<NFREQ; k++){
        switch(mode_4)
```

```
            {
            case 1:
                    g = find_max(lambda*(noise_estimate[k]/X_magnitude[k]),
                    (1-(noise_estimate[k]/X_magnitude[k])));
            break;

            case 2:
                    g = find_max(lambda*(Pt[k]/X_magnitude[k]),
                    (1-(noise_estimate[k]/X_magnitude[k])));
            break;

            case 3:
                    g = find_max((lambda*(noise_estimate[k]/Pt[k])),
                    (1-(noise_estimate[k]/Pt[k])));
            break;

            case 4:
                    g = find_max(lambda, (1-(noise_estimate[k]/Pt[k])));
            break;
        }
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
    Pt_prev = Pt;
}

void enhancement5(void){
    int k;
    float temporary;

    for(k=0;k<NFREQ;k++)
    {
        Pt[k] = sqrt(((1-kappa_4)*(X_magnitude[k]*X_magnitude[k]) +
        kappa_4*Pt_prev[k]*Pt_prev[k])); // P
        //Select the minimum
        if (f_index == 0){
            M1[k] = Pt[k]; //For the first frame set the minimum as the current sample
        }
        else{
            M1[k] = find_min(M1[k],Pt[k]); //for any other frame, compare the current sample
            with the stored minimum
        }
        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min
        noise_estimate[k] = alpha1*global_min;
    }
    for(k=0; k<NFREQ; k++){
        switch(mode_5){
            case 1:
                    temporary= sqrt(1 -
                    (noise_estimate[k]*noise_estimate[k])/(X_magnitude[k]*X_magnitude[k]));
                    g = find_max((lambda*(noise_estimate[k]/X_magnitude[k])), temporary);
            break;

            case 2:
```

```
                    temporary = sqrt(1 -
                    (noise_estimate[k]*noise_estimate[k])/(X_magnitude[k]*X_magnitude[k]));
                    g = find_max((lambda*(Pt[k]/X_magnitude[k])), temporary);
            break;

            case 3:
                    temporary = sqrt(1 - (noise_estimate[k]*noise_estimate[k])/(Pt[k]*Pt[k]));
                    g = find_max((lambda*(noise_estimate[k]/Pt[k])), temporary);
            break;

            case 4:
                    temporary = sqrt(1-(noise_estimate[k]*noise_estimate[k])/(Pt[k]*Pt[k]));
                    g = find_max(lambda*1.0, temporary);
            break;
        }
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
    Pt_prev = Pt;
}

void enhancement6(void)
{
    int k;
    float SNR, alpha6;
    for(k=0;k<NFREQ;k++)
    {
        //Select the minimum
        if (f_index == 0){
            M1[k] = X_magnitude[k]; //For the first frame set the minimum as the current
            sample
        }
        else{
            M1[k] = find_min(M1[k],X_magnitude[k]); //for any other frame, compare the
            current sample with the stored minimum
        }
        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min

        SNR = 20*log10f((X_magnitude[k]/global_min));
        if(SNR < -5)        alpha6 = 4.75;
        else if(SNR > 20)   alpha6 = 1.0;
        else                alpha6 = (4.0 - 0.15*SNR);

        noise_estimate[k] = alpha6*global_min; //calculate noise based on evaluated SNR
    }

    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        g = find_max((lambda*1.0), (1-(noise_estimate[k]/X_magnitude[k])));

        //implement zero-phase filter
        intermediate[k] = rmul(g, intermediate[k]);
        intermediate[FFTLEN - k].r = intermediate[k].r;
```

```
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
}

void enhancement8(void){
    int k;
        for(k=0;k<NFREQ;k++)
    {
        //Select the minimum
        if (f_index == 0){
            M1[k] = X_magnitude[k]; //For the first frame set the minimum as the current
            sample
        }
        else{
            M1[k] = find_min(M1[k],X_magnitude[k]); //for any other frame, compare the
            current sample with the stored minimum
        }
        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min
        noise_estimate[k] = alpha_basic*global_min;
    }
    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        g = find_max((lambda*1.0), (1-(noise_estimate[k]/X_magnitude[k])));
        //implement zero-phase filter
        intermediate[k].r       *= g;
        intermediate[k].i       *= g;
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i = -intermediate[k].i;

        Y_8[k] = g*X_magnitude[k];
        Y_current[k]= intermediate[k];
        Y_current[FFTLEN-k] = intermediate[FFTLEN - k];

        if (noise_estimate[k]/X_magnitude[k] >  residualth) {
            Y_min = Y_8[k];
            frame = 0;
            if (cabs(Y_minus1[k]) <  Y_min) {
                Y_min = cabs(Y_minus1[k]);
                frame = -1;
            }
            if (cabs(Y_minus2[k]) <  Y_min) {
                Y_min = cabs(Y_minus2[k]);
                frame = -2;
            }
            if(frame!=0){
                switch(frame){
                    case(-1):
                        Y_current[k].r = Y_minus1[k].r;
                        Y_current[k].i = Y_minus1[k].i;
                        Y_current[FFTLEN-k].r = Y_current[k].r;
                        Y_current[FFTLEN-k].i = -Y_current[k].i;
                        break;
                    case(-2):
                        Y_current[k].r = Y_minus2[k].r;
```

```
                        Y_current[k].i = Y_minus2[k].i;
                        Y_current[FFTLEN-k].r = Y_current[k].r;
                        Y_current[FFTLEN-k].i = -Y_current[k].i;
                break;
                }
            }
        }
    }
    f_switch=Y_minus2 ; //swaps pointers using aux i l iar y pointer
    Y_minus2=Y_minus1 ;
    Y_minus1=intermediate ;
    intermediate=f_switch ;

    ifft (FFTLEN, Y_current ) ; //computes the IFFT
    for ( k=0;k<FFTLEN; k++)
    {
        outframe[k] = Y_current[k].r ; /* copy input s t raight into output */
     }
}

void enhancement10(void)
{
    int k;
    float SNR, alpha10;

for(k=0;k<NFREQ;k++)
    {
        //Select the minimum
        if (f_index == 0){
            M1[k] = X_magnitude[k]; //For the first frame set the minimum as the current
            sample
        }
        else{
            M1[k] = find_min(M1[k],X_magnitude[k]); //for any other frame, compare the
            current sample with the stored minimum
        }

        //Find the global minimum out of 4 Mi
        global_min = find_min( find_min(M1[k],M2[k]), find_min(M3[k],M4[k]) ); //find the
        minimum between M1,M2 the min between M3,M4 and then the global min

        SNR = (X_magnitude[k]/global_min);
        if(SNR < -0.5)       alpha10 = 4.75;
        else if(SNR > 10)    alpha10 = 1.0;
        else                 alpha10 = (3.5 + 2.5*SNR);

        noise_estimate[k] = alpha10*global_min;
    }


    //Subtract noise spectrum
    for (k=0;k<NFREQ;k++)
    {
        g = find_max((lambda*1.0), (1-(noise_estimate[k]/X_magnitude[k])));

        //implement zero-phase filter
        intermediate[k] = rmul(g, intermediate[k]);
```

```
        intermediate[FFTLEN - k].r = intermediate[k].r;
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
}

float find_min(float a, float b)
{
    if (a <= b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
float find_max(float a, float b)
{
    if (a >= b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

7.2 Appendix 2: Enhancement Combinations Code

H:\RTDSPlab\project_pt2\RTDSP\nadal.c 24 March 2017 23:14

```
/*************************************************
                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 k/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 */
/************* NECESSARY ARRAYS ************/
float *X_magnitude;                                 /* magnitude spectrum */
float *low_pass_noise,*low_pass_noise_prev;         /* low_pass_noise estimate */
float *noise_estimate;                              //noise estimate
float *M1, *M2, *M3, *M4;                           /* 2.5 sec buffers to find minimum noise
amp */
float *Pt_prev,*Pt;                                 //low pass filter input
complex *intermediate;                              /* Complex array to perform FFT/IFFT
calculations */
/*******************************************/
float ingain, outgain;              /* ADC and DAC gains */
float cpufrac;                      /* Fraction of CPU time used */

/********** CONSTANTS ************************/
float g;                            /* frequency dependent gain factor*/
float alpha = 3.0;                  /* alpha factor */
float kappa_4;                      /* Low pass filter constant*/
float kappa_noise;                  //konstant for enhancement 3
float tau = 0.08;                   //Time constant
float tau_noise = 0.08;             //Time constant for enhancement 3
float lambda=0.0003;                //lambda value
float ROTATE_TIME = 2;              //parameter to control rotation of frames
/*******************************************/
volatile int io_ptr=0;              /* Input/ouput pointer for circular buffers */
volatile int frame_ptr=0;           /* Frame pointer */
volatile int f_index=-1;            /* Counts until 312 corresponding to 2.5 secs*/

 /****************************** 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 find_min(float a, float b);   /* Minimum function*/
float find_max(float a, float b);   /* Maximum function*/
/********************************* 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(NFREQ, sizeof(float));   /* M1 */
    M2                      = (float *) calloc(NFREQ, sizeof(float));   /* M2 */
    M3                      = (float *) calloc(NFREQ, sizeof(float));   /* M3 */
    M4                      = (float *) calloc(NFREQ, sizeof(float));   /* M4 */
    intermediate            = (complex *) calloc(FFTLEN, sizeof(complex));  /* complex
    buffer */
    X_magnitude             = (float *) calloc(NFREQ, sizeof(float));   /* magnitude
    spectrum */
    low_pass_noise          = (float *) calloc(NFREQ, sizeof(float));   /* low_pass_noise
    estimate */
    low_pass_noise_prev     = (float *) calloc(NFREQ, sizeof(float));   /* low_pass_noise
    estimate previous */
    noise_estimate          = (float *) calloc(NFREQ, sizeof(float));   /* Noise estimate
    array */
    Pt                      = (float *) calloc(NFREQ, sizeof(float));   /* Pt */
    Pt_prev                 = (float *) calloc(NFREQ, sizeof(float));   /* Pt_prev */

    /* 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;

    //Calculate kappa constants
    kappa_4 = exp((double)(-(TFRAME)/tau));
    kappa_noise = exp((double)(-(TFRAME)/tau_noise));

    /* 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; //used for loop counters
    int io_ptr0;
    float *tmp_M4; // used for buffer rotation
    float global_min; //variable to hold the min out of all frames
    //float temporary;

    /* 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;
```

```
    //Check for time condition and rotate buffers when necessary
    if (++f_index > (ROTATE_TIME/FRAMEINC*FSAMP)){

        f_index = 0;
        tmp_M4 = M4;
        M4 = M3;
        M3 = M2;
        M2 = M1;
        M1 = tmp_M4;

    }

    /* save a pointer to the position in the k/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];
        intermediate[k] = cmplx(inframe[k],0); /* store input to a complex form in order to
        calculate FFT */
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }

    /************************* DO PROCESSING OF FRAME  HERE *************************/
    /* please add your code, at the moment the code simply copies the input to the
    ouptut with no processing */

    //Perform FFT over the intermediate array
    fft(FFTLEN,intermediate);

    //Calculate |X(w)|, magnitude spectrum of input signal
    for (k = 0; k < NFREQ; k++){ //loop over the number of frequency bins
        X_magnitude[k] = cabs(intermediate[k]); // X

         //Enhancement 2
         //Low pass filter version of |X(w)|^2 -> Low pass filtering in the power domain
         Pt[k] = sqrt(((1-kappa_4)*(X_magnitude[k]*X_magnitude[k]) +
         kappa_4*Pt_prev[k]*Pt_prev[k]));
    }

    //Estimate noise spectrum
    for (k = 0; k < NFREQ; k++){
        if (f_index == 0){
            M1[k] = Pt[k];
        }
        else{
            M1[k] = find_min(M1[k], Pt[k]);
        }

        global_min = find_min( find_min(M1[k], M2[k]), find_min(M3[k], M4[k]) ); //calculate
        the minimum out of all Mi
        noise_estimate[k] = alpha*global_min; //estimate the noise spectrum
```

```
        //Enhancement 3
        low_pass_noise[k] = (1-kappa_noise)*noise_estimate[k] +
        kappa_noise*low_pass_noise_prev[k]; //low pass the noise estimate calculated above
    }

    //Subtract the noise spectrum
    for (k = 0; k < NFREQ; k++){
        //Enhancement 4, mode 3
        g = find_max(lambda*(low_pass_noise[k]/Pt[k]), (1-(low_pass_noise[k]/Pt[k])));

        //Perform zero-phase filtering -> subtract the noise spectrum
        intermediate[k] = rmul(g, intermediate[k]); //output NFREQ samples
        intermediate[FFTLEN - k].r = intermediate[k].r; //explore symmetry of FFT to output
        rest/symmetrical samples
        intermediate[FFTLEN - k].i =- intermediate[k].i;
    }
    //update previous versions of arrays
    low_pass_noise_prev = low_pass_noise;
    Pt_prev = Pt;

    //Perform inverse FFT
    ifft(FFTLEN,intermediate);

    //write the filtered from noise signal to the output frame
    for (k = 0; k < FFTLEN; k++){
        outframe[k]=intermediate[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;
}
/*****************************************************************************/
//Function to calculate the minimum between the two input arguments
float find_min(float a, float b){

    if (a > b)
        return b;
    else
        return a;
}
//Function to calculate the maximum between the two input arguments
float find_max(float a, float b){

    if (a > b)
        return a;
    else
        return b;
}
```

7.3 Appendix 3: Spectrograms

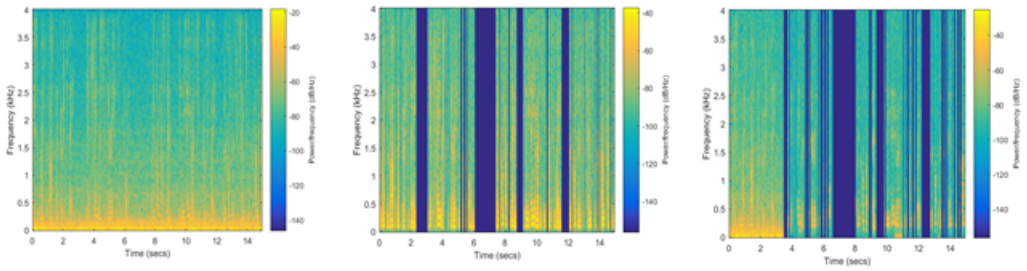

Figure 12: Spectrograms for "Car 1" *Left:* Corrupted signal *Middle:* Clean Signal *Right:* Filtered signal

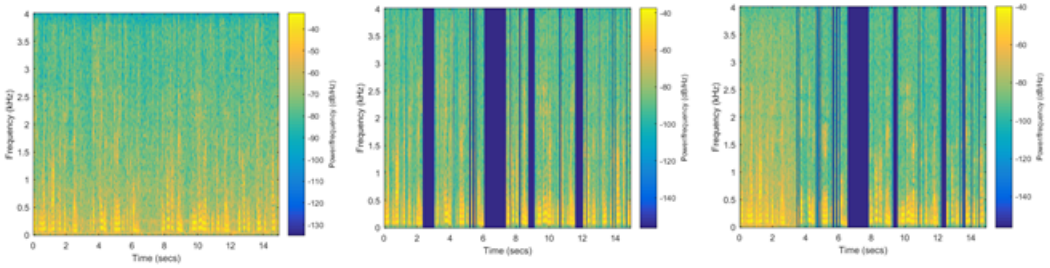

Figure 13: Spectrograms for "Factory 1" *Left:* Corrupted signal *Middle:* Clean Signal *Right:* Filtered signal