RTDSP Project: Speech Enhancement

Loek Janssen (lfj10) | Sebastian Grubb (sg3510)

Contents

Introduction

Noise removal

Advances in modern telecommunications, and aggressive improvements towards low voltage technology leaves many devices more susceptible to noise as signal power decreases. However, at the same time consumer demand pushes for practically noise free communication, even when the speaker is in a loud and busy location, meaning that both background and white noise must be filtered from the final signal; with even basic mobile phones requiring noise reduction algorithms on their DSP unit. The development of voice controlled/activated software that requires accurate evaluation of the spoken word further pushes this agenda.

The aim of the project is to implement a basic speech enhancer by removing noise, and then enhancing the algorithm to counter-act the advent of musical noise and speech pitch distortion.

Basic function

The aim of the basic program is to remove noise from an input signal, where both input signal (e.g speech) and noise is unknown to the programme before the measurement time. This is done by performing a Fast Fourier Transform (FFT) on the input signal incoming into the DSK and then estimating the noise spectrum to be removed from this input spectrum. After an Inverse Fast Fourier Transform (IFFT) is performed on the resulting spectrum to then be outputted to the audio port of the DSK as shown in figure 1.

When measuring the noise spectrum we make the assumption that at least once every 10 seconds the speaker will pause, allowing some time where only noise is being measured by our program. This is found by measuring the minimum spectra over the 10s (split into 4 2.5 second frames), this will however only return the minimum noise and thus the spectrum is exaggerated by what is called the subtraction factor α [1]to give a more accurate noise spectrum.

Throughout the 2.5 second frame the spectrum is continuously updated by comparing the minimum from the most recent 256 samples (or variations on the constant FFTLEN in the code) against what is recorded in the buffer, mathematically described as $M_1 = \min(|X(\omega)|, M_1(\omega))$ and then every 2.5 seconds the buffers will be shifted along with $M_1 = |X(\omega)|$. The noise estimate is determined from the minimum of these 4 buffers by the equation

$$|N(\omega)| = \alpha \min\big(M_i(\omega)\big), i = 1,2,3,4$$

This creates a delay in the program as all 4 buffers must be filled (and so 10 seconds pass) before the correct minimum spectrum will be loaded into the noise estimate array.

```
//M1(w) = min(|X(w)|,M1(w))
        if (M1[k]>X[k]){
            M1[k] = X[k];
        }
//N[k] chosen from the 4 frames
    N[k] = alpha*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
```

Code snippet demonstrating how the noise spectrum buffer was filled.

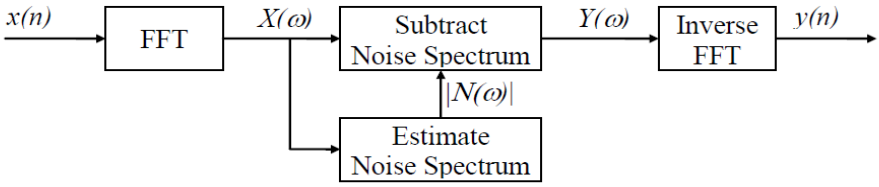

Figure 1. Diagram of basic programme functionality.

As we see above the program function can be mathematically described as $Y(\omega) = X(\omega) - N(\omega)$ after which the Inverse Fourier transform (IFFT) is taken to produce the output signal to the audio port $y(n)$. The lack of information about the phase of the noise signal means that we only subtract the magnitudes and hence we can rearrange the equation to give

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

where $g(\omega)$ is known as the frequency-dependent gain factor.

As this gain factor may sometimes be negative i.e. when $|N(\omega)| > |X(\omega)|$, the value of $g(\omega)$ is given a floor parameter which using the equation $g(\omega) = \max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$ prevents the gain ever decreasing below our chosen spectral floor parameter λ (known as 'lambda' in the code). This parameter is generally set between 0.01 and 0.1 in order to remove some musical noise, as a value of 0 produces large amounts of irritating musical noise [1]. A section below discusses how λ is varied with the subtraction factor to remove the most background and musical noise.

The code snippet below shows the implementation of the $Y(\omega)$ *and* $G(\omega)$ equations within the program, with rmul defined as the multiplication of a complex number (C[k] in this case) with a real number (the result of the $G(\omega)$ equation).

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

The basic implementation of the code was then tested with variable α and λ values to determine how well the code could remove the noise. First it was noticed that as α was increased there was a corresponding change in background noise, however when α was greater than 40 the speech sounded distant and distorted with large amounts of musical noise over the voice. Increasing λ could remove much of the musical noise but reintroduced some background white noise that was also undesirable. Different values of λ and α worked well for the different background noise, so a compromise of α=40 and λ=0.05 was originally chosen as it removed most of the background noise while leaving in only a small amount of musical noise for the majority of the audio files provided, with only phantom 4 still returning large background noise.

Program Function

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

Another second important structure of the program is the use of pre-processor directives to enable or disable different enhancements. This was chosen instead of if statements in order to avoid unnecessary computations to be done at runtime – it was found that if frame processing time was too lengthy crackling could be heard at the output thus explaining the need for code efficiency.

Another enhancement included in the program was to use the symmetry of the Fast Fourier Transform to halve the number of computations. Thus when operating in frequency domain all for loops were of the form: for (k=0;k<FFTLEN/2;k++) and before converting back to time domain the remaining of the samples were copied back in a symmetric manner.

Evaluation of different enhancements

Enhancement 1

The aim of the first enhancement is to tackle the issue that components at in each frequency bin greatly vary. For example each frequency bin can be assumed to have Gaussian distribution, meaning each element can be represented[1] by $N_{measured}(\omega) = \mathcal{N}(\mu_\omega, \sigma_\omega^2)$, where μ_ω and σ_ω^2 vary for each frequency bin. Combining this concept with the equation used to estimate the noise for each noise buffer estimate, $M_1(\omega) = \min\big(|X(\omega)|, M_1(\omega)\big)$[2], it can be seen that M_1 will only contain the low value spectrum components of noise (i.e. the values located in the lower part of the Gaussian distribution), explaining the need for a high alpha. While a high alpha somewhat fixes these problems there is the issue that when the mean or variance is different a high alpha will not fix this issue as the various frequency bins would need to be increased by a different value of alpha to represent the average value of noise (used to remove it from the signal). For example it can be assumed that the captured noise is (approximately) represented[3] by $M_1(\omega) = \mu_\omega - 2\sigma_\omega$ showing how if μ_ω and σ_ω changes for each sample, simply increasing by alpha will not work. As the average μ_ω is what is sought for reading a moving average would be more effective. This is the idea behind the first enhancement.

A moving-average (or low pass filter) can be implemented by a first order exponentially weighted moving average (EWMA [2], which is a type of IIR filter) of the form $P_t(\omega) = (1-k) \times |X(\omega)| + k \times P_{t-1}(\omega)$ by choosing an appropriate k value. To choose k we can use $e^{-\frac{FRAMELENGTH}{\tau}}$ with τ being the time constant essentially representing the amount of memory (i.e. how long the system would, to an extent, forget this past input). A time constant of 0.85 was chosen by using a time constant of $\tau = 50ms$ giving $e^{-\frac{8ms}{50ms}} \cong 0.85214$.

With this enhancement alpha was able to be reduced from around 30 to 2 in addition to noise being better removed, due to being estimated more accurately. The enhancement was then implemented in code as shown below, being done on the magnitude of the input spectrum straight after it is placed in the buffer X[k]

```
#if optim1 == 1
//low pass filtering of input
X[k] = X[k]*(1-K)+K*Ptm1[k];
//time delay output for use next loop
Ptm1[k] = X[k];
#endif
```

Enhancement 2

The second proposed enhancement is very similar to the previous one with the change that the low pass filter is done not in amplitude domain but in power domain, effectively squaring the amplitudes,

[1] While the noise frequency bins distribution is not actually Gaussian this is the best way to think about this solution

[2] where $X(\omega) = S(\omega) + N(\omega)$, i.e. the signal and noise, respectively

[3] This is a reasonable approximation as 95% of values in a Gaussian distribution lie within two standard deviations.

meaning the equation thus becomes $P_t^2(\omega) = (1-k) \times |X(\omega)|^2 + k \times P_{t-1}^2(\omega)$. This one was found to be more effective for the reasons that sound is heard by humans in the power, meaning that the estimate of noise corresponds more to what humans will hear it as.

A more mathematical explanation is that squaring all values means that larger values have a larger weighting. Thus the noise estimate will be more responsive to amplitude increases and will generally return a higher estimate of noise than the previous enhancement.

```
#elif optim2 == 1
//Low pass filter in power domain (squaring magnitude signals)
X[k] = (X[k]*X[k])*(1-K)+K*(Ptm1[k]*Ptm1[k]);
//transform back to magnitude
X[k] = sqrt(X[k]);
//time delay output for next loop
Ptm1[k] = X[k];
#endif
```

Enhancement 3

Following on from enhancement 1 where the input was low-pass filtered, the noise estimate resulting from the equations above was then also filtered in a similar manner. Once again this would create a moving average signal which would avoid the abrupt discontinuities of random noise, therefore decreasing the probability of a random spike causing disruption to the speech. The code below shows the implementation and how it is based on the same equation used in enhancement 1.

When tested however little or no improvement could be heard when it was added onto enhancement 2. This is because it is mainly useful for situations where the noise was highly variable unlike the sounds files tested, where the background noise was generally similar throughout.

```
#if optim3 == 1
//N(w) = alpha*min(M1,M2,M3,M4)
N[k]=(1-K)*alpha* min(min(M1[k],M2[k]),min(M3[k],M4[k]))+K*Ntm1[k];
Ntm1[k] = N[k];
```

Enhancement 4

In this enhancement the spectral floor parameter is adjusted so that less information is lost, this is because 'lambda' λ is cutting out any signal below our chosen value for our gain factor $g(\omega)$. When the noise amplitude is lower than the signal amplitude, we could lower our λ accordingly and so for low noise amplitude allow the output to retain the majority of information even for low energy speech.

The adjustment of λ to reflect the change in noise to signal ratio could be done in several ways which were explored in the code that produces the final gain factor. The basis of the design was adjusting λ within the equation $g(\omega) = \max(\lambda, 1 - \frac{N(\omega)}{X(\omega)})$ in the code. Some designs involved using the moving average signal $P(\omega)$ instead of $X(\omega)$ to test whether an averaged signal rather than rapidly changing input signal would produce a better λ (as frame to frame noise randomness could produce very different λ values over a short spectrum) .

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

Enhancement in code	Equation	Effect upon sound upon enhancement 2

optim4a	$g(\omega) = \max(\lambda\frac{N(\omega)}{X(\omega)}, 1-\frac{N(\omega)}{X(\omega)})$	Very little change – less crackling however
optim4b	$g(\omega) = \max(\lambda\frac{P(\omega)}{X(\omega)}, 1-\frac{N(\omega)}{X(\omega)})$	Little noticeable change
optim4c	$g(\omega) = \max(\lambda\frac{N(\omega)}{P(\omega)}, 1-\frac{N(\omega)}{P(\omega)})$	Removes some echo and some musical noise behind the voice
optim4d	$g(\omega) = \max(\lambda, 1-\frac{P(\omega)}{X(\omega)})$	Little noticeable change

The code below shows the implementation of each optimisation, with the pre-processor being used to set which was being tested.

```
//Y(w) variations
#if optim4a == 1
C[k] = rmul(max(lambda*(N[k]/X[k]),1-(N[k]/X[k])),C[k]);
#elif optim4b == 1
C[k] = rmul(max(lambda*(Ptm1[k]/X[k]),1-(N[k]/X[k])),C[k]);
#elif optim4c == 1
C[k] = rmul(max(lambda*(N[k]/Ptm1[k]),1-(N[k]/Ptm1[k])),C[k]);
#elif optim4d == 1
C[k] = rmul(max(lambda,1-(N[k]/Ptm1[k])),C[k]);
#else
//Y(w)  = X(w)*max(lambda,g(w))
C[k] = rmul(max(lambda,1-(N[k]/X[k])),C[k]);
#endif
```

Enhancement 5

Following along the idea used in enhancement 2, the gain factor was calculated in the power domain instead of the amplitude domain, giving the equation $\sqrt{1-\frac{|N(\omega)|^2}{|X(\omega)|^2}}$ which would then be compared to our λ to generate (ω) .

This was then implemented in code (shown below) at the same location as the previous enhancement 4s as shown below. When tested however the enhancement was found to give a large echo of the voice with the addition of more musical noise when enhancements 1 or 2 were used. This was significantly worse than enhancement 4c which 5 would replace and so was not used in final program.

```
#elif optim5 == 1
final =sqrt(1-((N[k]*N[k])/(X[k]*X[k])));
C[k] = rmul(max(lambda,final),C[k]);
#else
//Y(w)  = X(w)*max(lambda,g(w))
C[k] = rmul(max(lambda,1-(N[k]/X[k])),C[k]);
#endif
```

Enhancement 6

The suggested enhancement works by over estimating noise at low frequency bins, by creating in effect a filter; we were able to boost the value multiplied by the minimum spectrum (as described above) at low frequencies. Unlike at high frequencies where increasing α distorted the voice as the information contained is mainly upon the pitch of the voices, lower frequencies generally only contain

first order harmonics and the majority of the speech signal that the human ear listens to which one would like to amplify.

The code below shows the general implementation of enhancement 6, where a file which contained the array 'lpfcoef' was included which contained the coefficients to increase N for low frequencies. Improvements where then made upon this design which are discussed in the additional optimisations section later.

```
//if optim6 is active lpfcoef increase N for small K otherwise
//lpfcoef = 1 for all k
//N[k] chosen from the 4 frames
N[k] = alpha*lpfcoef[k]*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
```

Enhancement 7

This enhancement relies on changing the size of the FFT length used for the overlap add algorithm:

```
#define FFTLEN 256
```

Decreasing the FFT length resulted in the signal sounding tinnier with musical noise added on top. The reason for this is that by decreasing the FFT length means frequencies are being quantized even more which will transform back into signals with less fundamental frequency components. For example when taking 256 samples the first frequency bin represents all frequencies of the signal from 0 to 31.25 Hz whereas taking 128 samples the first frequency bin represents frequencies from 0 to 62.5 Hz and in the extreme case of having a length of 16 the first frequency bin would represent frequencies from 0 to 500Hz. It is easy to see why this would create problems and why a higher FFT length is more desired as sound is perceived logarithmically and low FFT length cause the hearing experience to be worse.

Increasing the FFT length caused the lag between input and output to be increased but more importantly it quantizes the sound in sound domain in a similar manner to what happened in frequency domain when decreasing the FFT length.

Unfortunately both good time and frequency resolutions are sought for and in this situation 256 was found to be the best compromise available.

Enhancement 8

The 8th enhancement is based around the idea of residual noise reduction as espoused by Boll [3], where through taking advantage of the randomness of noise from frame-to-frame we can remove cases where the signal noise amplitude to signal amplitude is greater than a chosen threshold (the variable 'residue' in the code demonstrating Enhancment 8 below) where previous noise filtering has already happened. This is done by replacing $Y(\omega)$ with minimum calculated $Y(\omega)$ over the adjacent and chosen frames as there is a high probability that the previous noise filtering has reproduced some frames which are still mainly noise but the minimum of the adjacent ones will supress this noise.

Implementation in code (shown below) was done by using an if function to detect if $\frac{N(\omega)}{X(\omega)} > 'residue'$ and adjusting the residue to remove as much musical and random noise as possible. In order to measure the minimum over the adjacent frames and maintain a constant continuity in the signal a delay of one frame was added at the output, a set signal was used to indicate if the output should take the minimum of the adjacent frames or not. This set signal would go around the for loop as `N[k]/X[k]` represents the result for C[k] not C[k-1] which would be output at that time, and so the minimum would be taken over the correct 3 frames at the next output.

The addition of the enhancement was to remove musical noise; there was a noticeable decrease when no other enhancements were used, however with the addition of enhancements 2,4c and VAD (discussed later) it was found that no noticeable difference could be heard. This is most likely because the other enhancements removed the majority of the noise which would activate enhancement 8.

```
#if optim8 == 1
if( set == 1)
{//if N[k-1]/X[k-1] > residue for C[k-1] then take min of adjacent frames
    D[k] = minC(minC(C[k-1],C[k-2]),C[k]);
    //output variable D[k]
        set = 0;
        //then check N[k]/X[k] for current future frame
    if( (N[k]/X[k]) > residue )
    {//set=1 so take min of adjacent frames in next loop
        set = 1;
    }
}
else if( (N[k]/X[k]) > residue )
{// check N[k]/X[k] for current future frame
    set = 1;
    //if > residue set = 1 for next loop
    D[k] = C[k-1];
    //set output that is IFFTed
}
else
{// else set remains 0 and output is delayed C[k-1]
    D[k] = C[k-1];
}
#endif
```

Additional Enhancements

Two enhancements were added, both variations on the 6th suggested enhancement, which was to deliberately increase alpha for low frequency bins, thus overestimating the noise collected for in those bins.

First proposed enhancement

Description

The first modification involved increasing alpha as it moved away from frequency bins containing the fundamental frequency of human speech, which are from around 80 Hz to 260Hz [3]. Due to speech harmonics being important for speech intelligibility [4], alpha was chosen to be less exaggerated at higher frequencies. From this the coefficients were generated using the Code Snippet 1, found in the appendix, and can be seen in Figure . The idea behind this implementation is that if frequencies which do not contain speech frequencies have noise a bit more aggressively removed then speech intelligibility may be improved. Thus this enhancement is the compromise between creating a pass band filter only allowing the fundamental speech frequencies through (which would not work due to removing all harmonics).

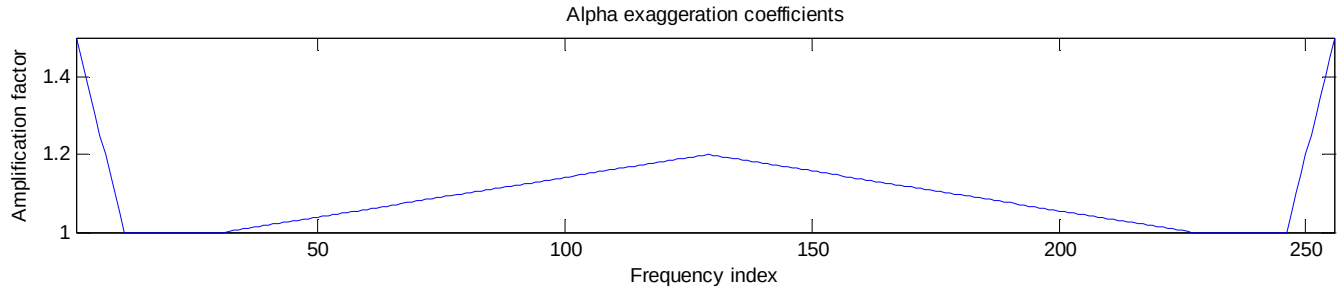

Figure 2 Visual plot of alpha coefficients use to exaggerate the noise at certain frequencies. The reason for the symmetric coefficients is due to the input of the FFT being symmetric. All coefficients are normalized thus can be multiplied with alpha to obtain the desired effect.

From the coefficients which we can store in an array called alpha_coefs[] we can then use them in the following way when defining the noise estimate for the currently processed frame:

```
N[k] = alpha*alpha_coefs [k]*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
```

Results

Figure 3 shows the spectrogram with and without the proposed enhancement. While it is difficult to see, the speech components (mainly in red and blue) are conserved across both tests. However the background noise (identifiable by the constant and across time blue components) can be seen to be less present (the blue shades are overall less intense). Listening to these samples does reveal that noise is less present.

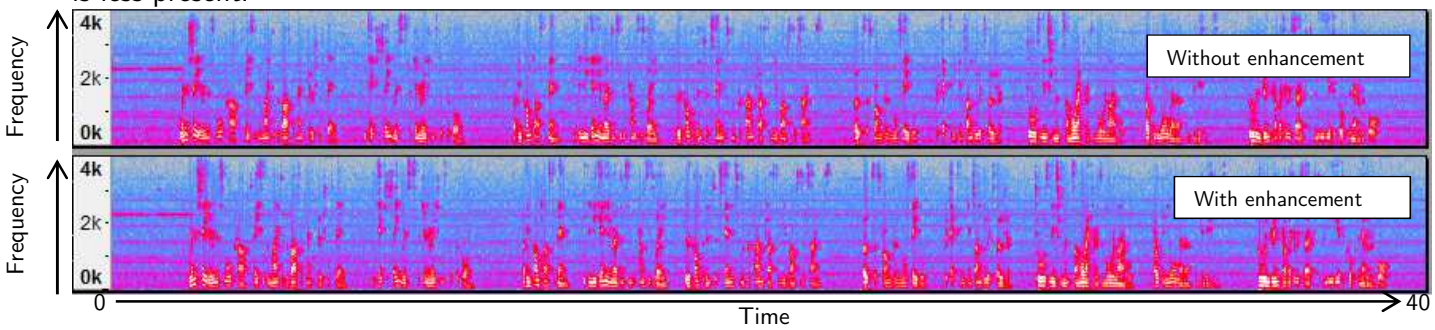

Figure 3 Spectrogram of the file lynx1.wav with and without the enhancement. The more intense the colour indicates a high presence of the frequency whereas a grey/blue colour implies the frequency components at the frequency are very low.

Other alpha amplification values were tested and higher values, as expected, also "muffled out" the voice decreasing intelligibility whereas lower values simply did not help with the implementation. Additionally it can be noted that this technique is ineffective at improving performance when the noise is at similar frequencies to speech.

Second proposed enhancement (VAD)

Implementation and description

The second implementation functioned as basic voice activity detection algorithm. During periods where voice was not detected the output was highly attenuated thus removing a majority of noise during non-speech periods. While this implementation does not increase sound intelligibility it does increase the listening experience – for example removing the occasional noises found in the "factory" files[4]. Additionally this algorithm is not strictly a voice detection algorithm, it is more of a constant background noise detector. Had more time been available methods to improve speech intelligibility would have been possible by building upon this first effort.

[4] The occasional "clunks" are what we are referring to here.

This enhancement first relied on calculating the SNR of the signal across all frequencies. Thus signal power was calculated by S_Power += X[k]*X[k] (done inside an if loop to add up from each frequency) and recursively looping through each element and noise was similarly calculated by N_Power += N[k]*N[k]/(alpha*alpha*lpfcoef[k]*lpfcoef[k]) (the division by alpha*alpha*lpfcoef[k]*lpfcoef[k]is intentional as N[k] is amplified by these values beforehand and we want the basic noise estimate to calculate SNR). From this SNR can be calculated as:

$SNR_{in} = 10\log_{10}\left(\frac{\sum_{i=0}^{FFTLEN/2} S_i^2}{\sum_{i=0}^{FFTLEN/2} N_i^2}\right)$, where S and N are, respectively, the amplitude of the input and noise.[5]

Due to the SNR value changing very often, a low pass filter value is used by implementing the following: $SNR_{lpf} = SNR_{calc} \times (1-k) + k \times SNR_{lpf\_prev}$. SNR_{lpf} is used for all future steps which involve SNR. The chosen k value was 0.95 but this can be change to liking.

A system similar to the noise M buffer is then set up with 2 buffers storing the maximum and minimum SNR:

```
SNR_min[snr_index] = min(SNR_tm1,SNR_min[snr_index]); //snr_index is what allows a circular buffer implementation
SNR_max[snr_index] = max(SNR_tm1,SNR_max[snr_index]);
```

These buffers are then switched every 2.5 seconds, with the oldest value being 10 seconds old. An across time min and max are then calculated to allow us to calculate a range:

```
snr_inter_min = min(min(SNR_min[0],SNR_min[1]),min(SNR_min[2],SNR_min[3]));
snr_inter_max = max(max(SNR_max[0],SNR_max[1]),max(SNR_max[2],SNR_max[3]));
snr_inter_range = snr_inter_max - snr_inter_min; //gives an idea of how the SNR range for the signal currently is
```

The range is extremely useful. Firstly if it is found to be less than 3dB the VAD is disabled as the range is determined to be too small to accurately determine whether speech exists or not and we would risk removing speech as well as noise.

Secondly the range is used to calculate the cut-off SNR:

```
SNR_Threshold = snr_inter_min+0.2*snr_inter_range;
```

In this case the SNR threshold would then be set to the lower 20%. The current implementation is a bit more complex and is similar to this SNR_Threshold = snr_inter_min+Threshold_coef*snr_inter_range, where the value Threshold_coef changes between 15% and 20% according to the SNR range.

Finally the last step is to activate the amplitude decrease when the input is detected to be under the calculated threshold:

```
if (VAD_on&(SNR_tm1 <= SNR_Threshold)){
    for (k=0;k<FFTLEN/2;k++){
        C[k]= rmul(VAD_coef,C[k]);//VAD_coef is a value between 0 and 0.5 which reduces signal amplitude
    }
}
```

A step by step of this process can be found in the appendix under Code Snippet **2**.

[5] Note that here S does not represent signal as such but rather signal and noise.

Results

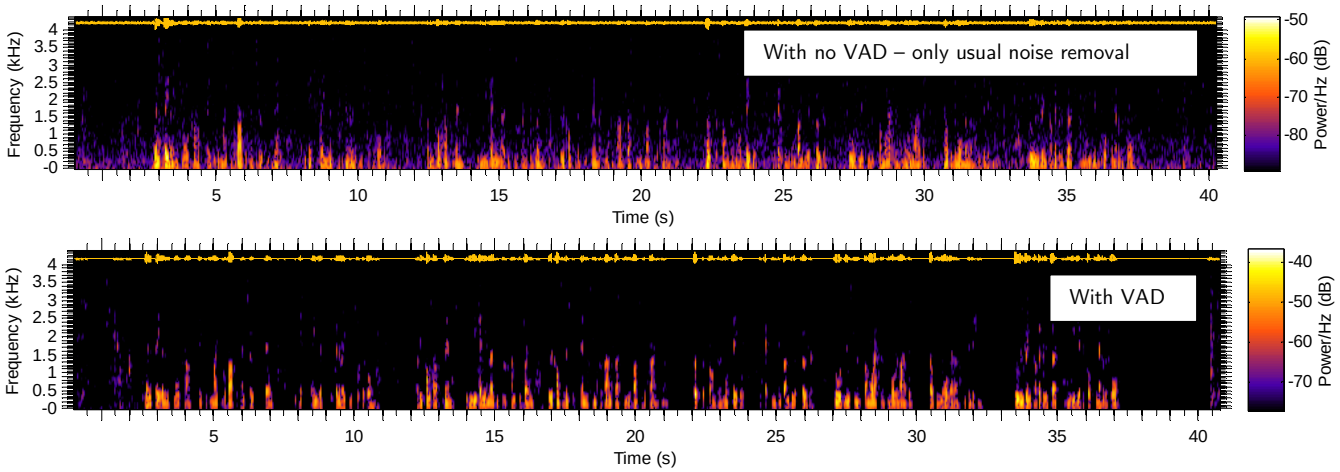

Figure 4 Spectrogram of the file "factory2.wav" with and without VAD.

Figure 4 shows factory 2 being processed using the proposed VAD algorithm. As can be seen at silent periods the noise is effectively completely removed (simply being black indicating no sound presence). Applying this other sound samples also worked effectively as seen from **Table 1**.

Sound file	Observation	Sound file	Observation
car1.wav	VAD sometimes deactivated due to low range	lynx2.wav	Worked well
factory1.wav	Worked well	phantom1.wav	Worked well
factory2.wav	Worked well – sometimes let a bit of noise through	phantom2.wav	Worked well – sometimes deactivated itself
lynx1.wav	Worked well	phantom4.wav	Worked to an extent – can get a bit confused and clip speech

Table 1 shows how the VAD worked across all provided sound samples.

Implementing an adaptation algorithm which set the threshold % based on the range and average SNR values also helped the VAD lock in on more appropriate thresholds according to the input.

It was found that without correct adaptation algorithms some speech could be clipped, an effect commonly observed in SNR based voice activity detectors in low SNR conditions [5] (thus the motivation to disable the VAD when the SNR or SNR range is too low). However this VAD implementation can still prove useful in better estimating noise by only taking samples during periods in which speech is not detected.

Final Program

The final program incorporated enhancements 2, 4c and the VAD into the design (shown as optim6a in the code). This decision was taken as it was found that they enhancements produced a better sounding output than the basic program and other enhancements tested.

During the final noise test it was found the system was very effective at removing the background noise across all sound files, with the VAD working effectively to remove all noise during silent parts of the sound file. Only in phantom 4 was the balance difficult to strike, which despite the ramped attenuation of the signal produced a small amount of clipping straight after the noise was removed as the VAD adjusted to the sudden change. Despite this the addition of the VAD and enhancements

made significant improvements to the program, severely lowering musical noise while maintaining a very low background noise at output.

Using a spectrogram we can see the full effect of a final design upon the sound files by comparing the original clean file, the 'noisy' file, and the DSK output of the noisy file. Figure 5 shows the spectrograms of the unfiltered phantom2.wave, the output of the DSK when phantom2.wave is run through our final program and then the clean file provided to identify exactly how well our speech enhancer worked.

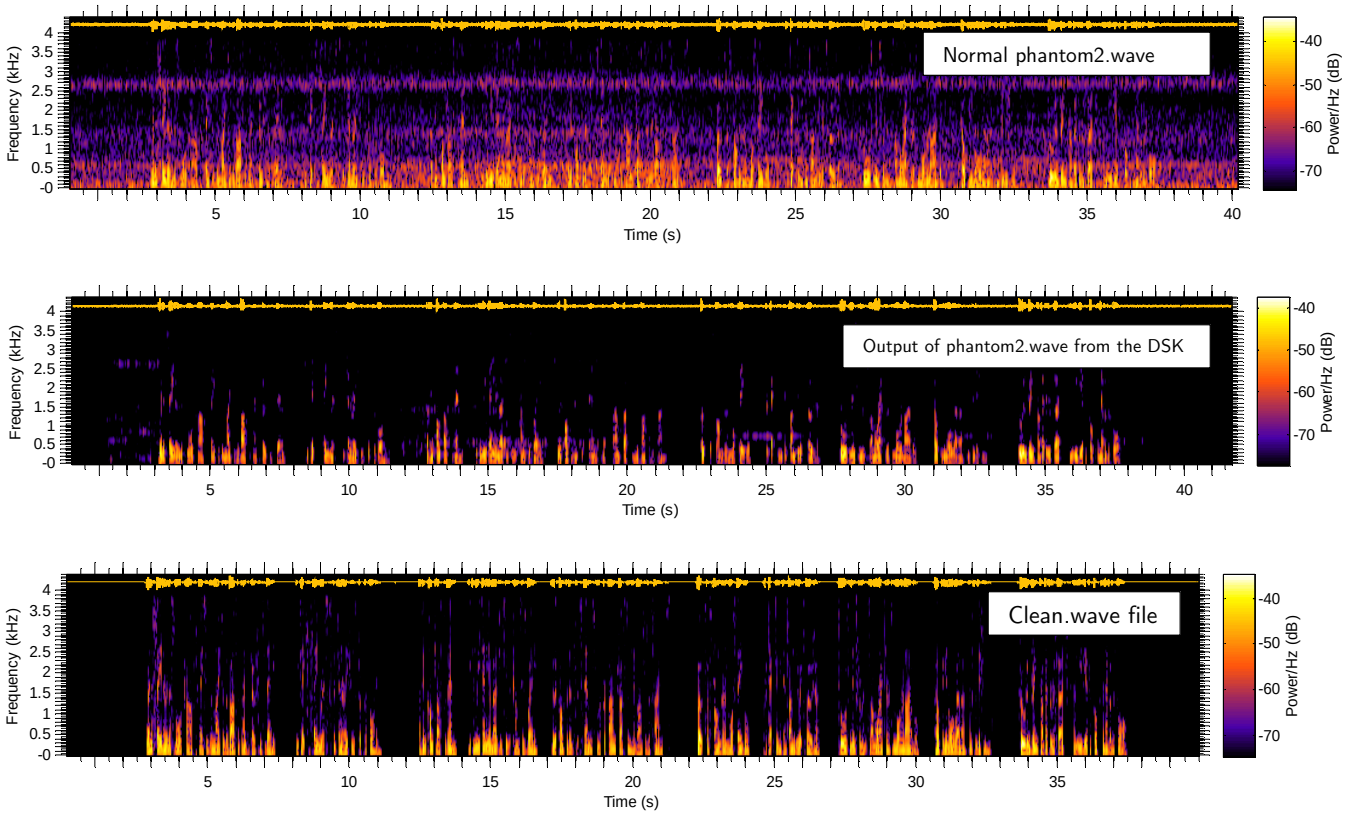

Figure 5: Spectrographs of input phantom2 file, output from speech enhancer and a 'clean' file[6]

As we can see the speech enhancer was very effective at removing background noise that is identifiable in the noisy phantom2 file. Removing the obvious noise at ≈ 2.5kHz and almost all of the background noise at lower frequencies. Due to the low-pass filtering we do loss some high frequency information which is generally unimportant to human hearing; otherwise a very small amount of low frequency is lost resulting in a fuzzier spectrogram around those frequencies. While we can see that some (often unimportant) information was lost, the speech enhancer removes almost all the background noise, exactly as was to be designed.

The impulse response of the final program is shown in figure 7, the delay of around 48ms is larger than the expected 32 ms (from program operation) but this is mainly due to the delay due to the DSK board. Otherwise the impulse response is exactly as expected. When a sinewave of continuous frequency is placed in the program the output sinewave decreased in amplitude significantly after 10 seconds, eventually dropping to near zero.

[6] Note: the input files have been downsampled to 8000Hz sampling rate for better comparison purposes

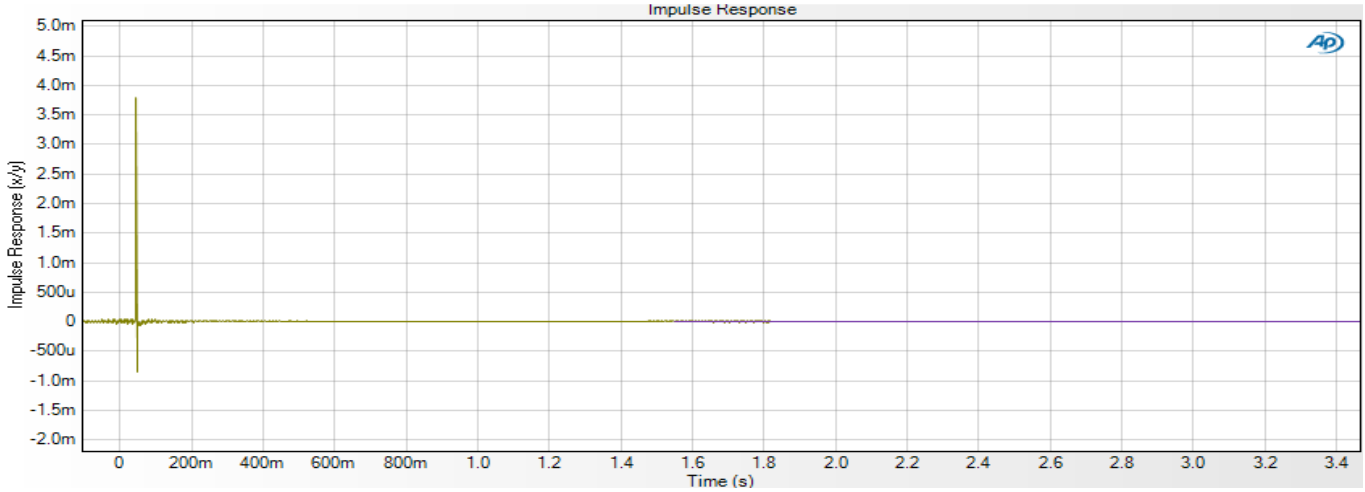

Figure 7: impulse response of DSK with speech enhancer

Future Improvements

Possible improvements would tackle the central issue of noise estimation which is the most crucial part for effective noise removal. One method would be to improve the smoothing function of the noise estimates (which enhancement 1 and 2 are concerned with) by having the k value be dynamic. This is because currently the smoothing function is always biased for lower noise values and when noise amplitude increasing the estimates lag behind. Additionally periods of speech have their frequency bins be widened up (when viewed as a time function) leading to unwanted noise estimates. Thus one improvement would to have the time constant (which determines k from $e^{-\frac{FRAMELENGTH\ [ms]}{\tau\ [ms]}}$) decrease during periods of speech or very quick increases (depending on the implementation and the type of accuracy wanted) to "forget" faster about past values which may simply be anomalies.

Works Cited

[1] M. S. &. M. Berouti, "Enhancement of Speech Corrupted by Acoustic Noise," *Proc ICASSP,* pp. pp208-211, 1979.

[2] P. Čisar and S. M. Čisar, "Optimization Methods of EWMA Statistics," 2011. [Online]. Available: http://www.uni-obuda.hu/journal/Cisar_Cisar_31.pdf. [Accessed March 2013].

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

[4] "Human speech fundamental frequencies," in *Clinical Measurement of Speech and Voice*, London, Taylor and Francis Ltd., 1987, pp. 177,188.

[5] J. Rodman, "The Effect of Bandwidth on Speech Intelligibility," Polycom, 2006. [Online]. Available: http://docs.polycom.com/global/documents/whitepapers/effect_of_bandwidth_on_speech_intelligibility_2.pdf. [Accessed 2013].

[6] L. Ding, A. Radwan, M. S. El-Hennawey and R. A. Goubran, "Measurement of the Effects of Temporal Clipping on Speech Quality," *IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT,* vol. 55, no. 4, pp. 1197-1203, August 2006.

Appendix

Code snippets

```
clc;
a = ones(256,1);
%choose what factor to set the values to
%and where this should start
lp_ampli = 1.5;
lp_freq = 10;
hp_ampli = 1.2;
hp_freq = 30;
for i=0:(lp_freq-1)
    a(i+1) = (lp_freq*lp_ampli-(lp_ampli-1)*i)/lp_freq;
    a(256-i) = (lp_freq*lp_ampli-(lp_ampli-1)*i)/lp_freq;
end
a_eq = (hp_ampli-1)/(128-hp_freq);
b_eq = 1 - a_eq*hp_freq;
for i=hp_freq:128
    a(i+1) = a_eq*i+b_eq;
    a(257-i) = a_eq*i+b_eq;
end
plot(1:256,a)
axis tight
title('Alpha exaggeration coefficients')
xlabel('Index')
ylabel('Amplification factor')
formatSpec = '%1.16f,'; %set format to high precision to avoid rounding
issues
fid=fopen('project_pt1\RTDSP\lpfcoef.txt','w'); %open file
% define order of filter+1 here
% simpler and more intuitive way than

%define a and b coefficient arrays
fprintf(fid,'float lpfcoef[] = {');
fprintf(fid,formatSpec,a(1:length(a)-1));
fprintf(fid,'%1.16f};\n',a(length(a)));
fclose(fid); %close file
```

Code Snippet 1 Matlab code to generate alpha exaggeration coefficients

1. Calculate Noise Power
2. Calculate Input Power
3. Set SNR = 10log(Input_Power/Noise_Power)
4. Apply low pass filter on SNR:
   SNR_lpf = 0.1*SNR+0.9*SNR_lpfprev
5. Create minimum and maximum buffers
6. From these buffers calculate the SNR range (max – min)
7. From this range determine whether or not it is sensible to activate voice activity detection (values under 3 are not recommended to have VAD on)
8. Calculate the threshold: SNR_threshold = SNR_min + 0.2*SNR_range
   a 20% range is an experimental value found to work but any other appropriate value could be used
9. If (SNR_lpf_current < SNR_Threshold) then decrease output signal by a constant as the algorithm has decided that no speech is occurring.

Code Snippet 2 Pseudo-code to create a simple VAD which removes noise when no speech is occurring.

Final Code

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

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

PROJECT: Frame Processing

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

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

*************************************************************************************
By Danny Harvey: 21 July 2006
Updated for use on CCS v4 Sept 2010
*************************************************************************************/
/*
*     You should modify the code so that a speech enhancement project is built
*  on top of this template.
*/
/***************************** Pre-processor statements ****************************/
//optimisations  - defined as preprocessors to keep code efficient
//however means optimisations cannot be switched on and off at runtime
#define optim1          0
#define optim2          1
#define optim3          0
#define optim4a   0
#define optim4b   0
#define optim4c   1//
#define optim4d   0
#define optim5          0  //bad
#define optim6          0 //removes some bg noise better
#define optim6a   1 //calculates SNR and dynamically changes alpha
#define optim8 0
//  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"
#include "lpfcoef.txt"
//include coefs

// 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/(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 */
//added defines
#define K 0.95//(exp(-(TFRAME/0.04)))

/***************************** 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 */
//---------------------------------------------------------------------------
//################################ - Added Variables - ########################
float X[FFTLEN];                        /* Temporary variable storing |X(w)| */
complex C[FFTLEN];                      /* Temporary variable storing X(w) - ie the complex
version */
float *M1;                              /* Buffers used to store past noise estimates */
float *M2;                              /* Buffers used to store past noise estimates */
float *M3;                              /* Buffers used to store past noise estimates */
float *M4;                              /* Buffers used to store past noise estimates */
float N[FFTLEN];                        /* Noise estimate */
float alpha = 4;                        /* alpha coefficient used in noise overestimation */
float lambda = 0.02;                    /* lambda coefficient used in noise overestimation */
int f_count = 0;                        /* frame count */
int f_count_loop = 312;                 /* choose how often to loop back */
#if (((optim1)||(optim2))==1)
float Ptm1[FFTLEN];
#endif
#if ((optim3)==1)
float Ntm1[FFTLEN];                     /* Stores past spectral estimates of noise*/
#endif
#if (optim6a ==1)
float S_Power = 0;                      /* Signal Power estimate in able to calculate SNR */
float N_Power = 0;                      /* Noise Power estimate in able to calculate SNR */
float SNR = 1;                          /* Calclated/estimated SNR*/
float SNR_tm1 = 1;                      /* Estimated SNR Trail */
float SNR_trail_coef = 0.99;
float SNR_Threshold = 0;                /* Calclated/estimated SNR*/
float Threshold_coef = 0.22;            /* Calclated/estimated SNR*/
float SNR_max[4] = {0,0,0,0};           /* Buffer of max SNR */
float SNR_min[4] = {0,0,0,0};           /* Buffer of min SNR */
float VAD_coef = 0.1;                   /* Voice activity detection factor */
float snr_inter_min = 0;                /* Stores current SNR global min */
float snr_inter_max = 0;                /* Stores current SNR global max */
float snr_inter_range = 0;              /* Store current SNR range */
int   VAD_on = 1;                       /* Allows the code to turn VAD off */
int   snr_index = 0;                    /* index for snr min max buffers */
int   snr_count = 0;                    /* index for snr min max buffers */
#endif
#if optim8 == 1
int set = 0;
float residue=0.5;
complex D[FFTLEN];
#endif
#if optim5 == 1
float final;                            /* Temporary variable need for optim 5*/
#endif

//---------------------------------------------------------------------------
/****************************** Function prototypes ***************************/
void init_hardware(void);    /* Initialise codec */
```

```
void init_HWI(void);            /* Initialise hardware interrupts */
void ISR_AIC(void);             /* Interrupt service routine for codec */
void process_frame(void);       /* Frame processing routine */
//################################################################################
//-------------------------| Added Prototypes |--------------------------
float max(float a, float b);    /* Maximum finding function for floats*/
float min(float a, float b);    /* Minimum finding function for floats*/
//################################################################################
//--------------------| End of Added Prototypes |-------------------------

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//_____________________________\\Custom Functions//_____________________________

// |Max| Finds max for two floats
float max(float a, float b){
      if (a>b)
      {
            return a;
      }else
      {
            return b;
      }
}
// |Min| Finds min for two floats
float min(float a, float b){
      if (a<b)
      {
            return a;
      }else
      {
            return b;
      }
}
// |Min| Finds min for two complex
complex minC(complex a, complex b){
      if(cabs(a)<cabs(b))
      {
            return a;
      }
      else
      {
            return b;
      }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//_____________________________\\End of Custom Functions//_____________________

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

      int k; // used in various for loops

      /*  Initialise and zero fill arrays */

      inbuffer    = (float *) calloc(CIRCBUF, sizeof(float));     /* Input array */
      outbuffer   = (float *) calloc(CIRCBUF, sizeof(float));     /* Output array */
      inframe           = (float *) calloc(FFTLEN, sizeof(float));      /* Array for processing*/
      outframe    = (float *) calloc(FFTLEN, sizeof(float));      /* Array for processing*/
      inwin       = (float *) calloc(FFTLEN, sizeof(float));      /* Input window */
      outwin            = (float *) calloc(FFTLEN, sizeof(float));      /* Output window */
      M1                = (float *) calloc(FFTLEN, sizeof(float));      /* Used for storage of
past noise spectra */
      M2                = (float *) calloc(FFTLEN, sizeof(float));      /* Used for storage of
past noise spectra */
      M3                = (float *) calloc(FFTLEN, sizeof(float));      /* Used for storage of
past noise spectra */
      M4                = (float *) calloc(FFTLEN, sizeof(float));      /* Used for storage of
past noise spectra */


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

      /* Initialise hardware interrupts */
      init_HWI();

      /* Initialise 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];
		//init custom variables here
		X[k] = 0;
		M1[k] = 0;
		M2[k] = 0;
		M3[k] = 0;
		M4[k] = 0;
		#if (((optim1)||(optim2))==1)
		//Setup variable to allow a moving average of frequency bins
		Ptm1[k] = 0;
		#endif
		#if ((optim3)==1)
		//set all noise trail estiamtes to 0
		Ntm1[k] = 0;
		#endif
		#if (optim6 == 0)
		//if optim6 not enable set all the coefficients to one such that they do not have any
effect
		lpfcoef[k] = 1;
		#endif
		//end of custom added variables
	}
	ingain=INGAIN;
	outgain=OUTGAIN;


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

/******************************** init_hardware() ******************************/
void init_hardware()
{
	// Initialise 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 *Mptr;

	/* 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 Code Here||||||||||||||||||||||||||||||||||||||||||||
	//copy values to complex variable to do fft
	for (k=0;k<FFTLEN;k++)
	{
		C[k] = cmplx(inframe[k],0);/* copy input to complex */
	}
	//do the fft
	fft(FFTLEN,C);
	#if optim6a == 1
	//increase SNR buffer count
	snr_count++;
	//reset input power variables to 0
	S_Power = 0;
	//every defined number of frames reset the noise power estimate to be recalculated
	if (snr_count >= f_count_loop/3)
	{
		N_Power = 0;
	}
	#endif
	//now start processing on those values
	for (k=0;k<FFTLEN/2;k++)
	{
		//init X(w)
		X[k] = cabs(C[k]);
		#if optim6a
		//calculate power
		S_Power += X[k]*X[k];
		#endif
		#if optim1 == 1
		X[k] = X[k]*(1-K)+K*Ptm1[k];
		//Ptm1[k] = X[k];
		#elif optim2 == 1
		X[k] = (X[k]*X[k])*(1-K)+K*(Ptm1[k]*Ptm1[k]);
		X[k] = sqrt(X[k]);
		//Ptm1[k] = X[k];
		#endif
		//M1(w) = min(|X(w)|,M1(w))
		if (M1[k]>X[k]){
			M1[k] = X[k];
		}
		//N(w) = alpha*min(M1,M2,M3,M4)
		#if optim3 == 1
		//N(w) = alpha*min(M1,M2,M3,M4)
		N[k] = (1-K)*alpha*lpfcoef[k]*min(min(M1[k],M2[k]),min(M3[k],M4[k]))+K*Ntm1[k];
		Ntm1[k] = N[k];
		#else
		//if optim6 is active lpfcoef increase N for small K otherwise
		//lpfcoef = 1 for all k
		//N[k] chosen from the 4 frames
		N[k] = alpha*lpfcoef[k]*min(min(M1[k],M2[k]),min(M3[k],M4[k]));
		#endif
		#if (optim6a == 1)
		//VAD Code here
		if (snr_count >= f_count_loop/3)
		{
			N_Power += N[k]*N[k]/alpha;
```

```
                //calculate power
            }
            #endif
            //Y(w) variations
            #if optim4a == 1
            C[k] = rmul(max(lambda*(N[k]/X[k]),1-(N[k]/X[k])),C[k]);
            #elif optim4b == 1
            C[k] = rmul(max(lambda*(Ptm1[k]/X[k]),1-(N[k]/X[k])),C[k]);
            #elif optim4c == 1
            C[k] = rmul(max(lambda*(N[k]/Ptm1[k]),1-(N[k]/Ptm1[k])),C[k]);
            #elif optim4d == 1
            C[k] = rmul(max(lambda,1-(N[k]/Ptm1[k])),C[k]);
            #elif optim5 == 1
            final =sqrt(1-((N[k]*N[k])/(X[k]*X[k])));
            C[k] = rmul(max(lambda,final),C[k]);
            #else
            //Y(w)  = X(w)*max(lambda,g(w))
            C[k] = rmul(max(lambda,1-(N[k]/X[k])),C[k]);
            #endif
            #if optim8 == 1
              if( set == 1)
              {//if N[k-1]/X[k-1] > residue for C[k-1] then take min of adjancent frames
                  D[k] = minC(minC(C[k-1],C[k-2]),C[k]);
                  //output variable D[k]
                        set = 0;
                        //then check N[k]/X[k] for current future frame
                  if( (N[k]/X[k]) > residue )
                  {//set=1 so take min of adjacent frames in next loop
                        set = 1;
                  }
              }
              else if( (N[k]/X[k]) > residue )
              {// check N[k]/X[k] for current future frame
                  set = 1;
                  //if > residue set = 1 for next loop
                  D[k] = C[k-1];
                  //set output that is IFFTed
              }
              else
              {// else set remains 0 and output is delayed C[k-1]
                  D[k] = C[k-1];
              }
          #endif
          #if ((optim1)||(optim2) == 1)
              //set past value to new one
              Ptm1[k] = X[k];
          #endif
      }
      #if (optim6a == 1)
      //calculate SNR
      SNR = 10*log10(S_Power/N_Power);
      SNR_tm1 = SNR_tm1*SNR_trail_coef + (1-SNR_trail_coef)*SNR;
      //Keep SNR estimates within a range (to avoid overflow issues)
      if ((SNR_tm1 <= -20) || (SNR_tm1 >= 10)){
            SNR_tm1 = 0;
      }
      //claculate min and max SNR
      SNR_min[snr_index] = min(SNR_tm1,SNR_min[snr_index]);
      SNR_max[snr_index] = max(SNR_tm1,SNR_max[snr_index]);
      //Process noise if speech not detected
      if (VAD_on&(SNR_tm1 <= SNR_Threshold)){
            for (k=0;k<FFTLEN/2;k++){
                  C[k]= rmul(VAD_coef,C[k]);//cmplx(0,0);
            }
      }
      if (snr_count >= f_count_loop/3){
        snr_count = 0;
        //update Threshold
        //SNR_Threshold =
min(min(SNR_min[0],SNR_min[1]),min(SNR_min[2],SNR_min[3]))+Threshold_coef*(max(max(SNR_max[0],SNR_
max[1]),max(SNR_max[2],SNR_max[3]))-min(min(SNR_min[0],SNR_min[1]),min(SNR_min[2],SNR_min[3])));
        snr_inter_min = min(min(SNR_min[0],SNR_min[1]),min(SNR_min[2],SNR_min[3]));
        snr_inter_max = max(max(SNR_max[0],SNR_max[1]),max(SNR_max[2],SNR_max[3]));
        //snr_inter_min = (3*SNR_min[0]+2*SNR_min[1]+2*SNR_min[2]+SNR_min[3])/8;
        //snr_inter_max = (3*SNR_max[0]+2*SNR_max[1]+2*SNR_max[2]+SNR_max[3])/8;
        snr_inter_range = snr_inter_max - snr_inter_min;
        //If power opf nosie under 1 we can assumed noise has not been correctly detected so far
and VAD must be turned off
        if (N_Power <= 1)
```

```
        {
             VAD_on = 0;
             Threshold_coef = .05;
        }else
        {
              if (snr_inter_range <= 3)
              {
                  VAD_on = 0;
                  Threshold_coef = 0;
              }else
              {
                  //turn VAD on and decrease any non-speech inputs by 0.2
                  VAD_on = 1;
                  Threshold_coef = 0.2;
              }
        }
        SNR_Threshold = snr_inter_min+Threshold_coef*snr_inter_range;
        // shift SNR minmax buffers
        snr_index++;
        if (snr_index >= 4){
             snr_index = 0;
        }
        SNR_min[snr_index] = SNR_tm1;
        SNR_max[snr_index] = SNR_tm1;
      }
      #endif
      //shift every arbitrary number of seconds
      if (f_count >= f_count_loop){
            f_count = 0;
            //shift
            Mptr = M4;
            M4 = M3;
            M3 = M2;
            M2 = M1;
            M1 = Mptr;
            //M1(w) = |X(w)|
            for (k=0;k<FFTLEN/2;k++){
                 M1[k] = X[k];
            }
      }
     //increase frame count
      f_count++;
      //perform ifft to get back into time domain
     for (k=FFTLEN/2+1;k<FFTLEN;k++){
            //C[k]=cmplx(0,0);
            C[k]=cmplx(C[FFTLEN-k].r,-1*C[FFTLEN-k].i);
      }


    #if optim8 == 1
    ifft(FFTLEN,D);
    #else
    ifft(FFTLEN,C);
    #endif
      //write to output
      for (k=0;k<FFTLEN;k++){
            #if optim8 == 1
            outframe[k] = D[k].r;
            #else
            outframe[k] = C[k].r;
            #endif
      }


//#############################################################################################
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//|||||||||||||||||||||||||||||||||| End of Section ||||||||||||||||||||||||||||||||||
      /*****************************************************************************/

      /* 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;
}

/*******************************************************************************/
```