Speech Enhancement by Spectral Noise Subtraction

Yann HERKLOTZ*
Imperial College London

Divyansh MANOCHA**
Imperial College London

Abstract— **The purpose of this project is to achieve real-time speech enhancement by using spectral subtraction of the estimated noise spectrum from the original signal. The noise estimation was performed by finding the minimum spectrum over a time frame, multiplying it by a small factor, and subtracting that from the original spectrum.**

I. INTRODUCTION

There are many ways in which noise reduction can be performed. The implementation that was chosen for this project can be seen below in Figure 1.

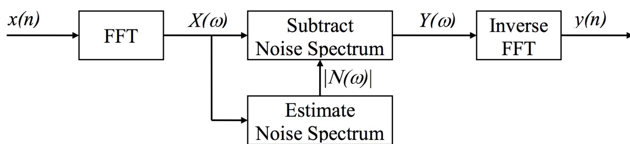

Fig. 1. A basic implementation of the noise subtraction algorithm

It can be seen in the figure that the input signal is converted to the frequency domain using the FFT (Fast Fourier Transform). The noise spectrum is then estimated, which is further explained below, and then subtracted from the original spectrum. The output can then be converted back to the time domain using the IFFT (Inverse Fourier Transform).

The resulting signal should not be noisy anymore, as it theoretically should not contain the noise spectrum anymore, however, in practice, this technique is not perfect. Spectral subtraction often leaves artifacts and specific frequencies in the resultant spectrum. When listening to the output that includes such artifacts, musical noise can be heard instead of the actual noise.

There are optimisations which were added to the algorithm that also focuses on reducing the resultant musical noise, which are described further in Section III.

* yann.herklotz15@imperial.ac.uk
** divyansh.manocha15@imperial.ac.uk

II. BASIC IMPLEMENTATION

The speech is converted into its frequency domain in order to be processed with noise removal techniques and then converted back into the time domain. The assumption here is that the noise is additive, but not necessarily white. It should also be noted that the phase spectrum of the speech will remain untouched.

As the signal is continuous and the algorithm should work in real-time, the input is split into frames. With a non-overlapping frame technique, taking the fourier transform on a frame will give rise to spectral artefacts. This would therefore not be a true representation of the frequencies of the original spectrum.

One way to avoid this is to use windowing, which reduces the amplitude of the frame to zero at the edges before FFT is performed. This, however, changes the signal in the time domain and therefore after perform IFT, the amplitude will vary unexpectedly.

Overlap add can be used so that the envelope of the overlapping windows always adds to one. This eradicates spectral artefacts due to windowing.

The square root of a Hamming window can be used as the window function, shown in equation 1.

$$\sqrt{1 - 0.85185 \cos\left(\frac{(2k+1)\pi}{N}\right)} \text{ for } k = 0 \ldots N-1 \tag{1}$$

A long frame would result in a better frequency resolution, but a worse time resolution - and vice versa for short frames. Since FFT is most efficient when the length is a power of 2, the length is chosen to be 256 in this case. Thus corresponding to 32 ms at 8 kHz sampling rate.

A. Algorithm

To develop the noise estimation, it is assumed that any speech sample does not have talking for more than 10 seconds. At each frequency the minimum power over the last 10 seconds is taken and stored in a **noise minimum buffer**.

1) Initialisation and circular buffer: The size of the buffer is defined by the FFT length and Frame increment. This is $256 + \frac{256}{4} = 320$. An input output pointer is used to traverse through this buffer. Therefore it must wrap around as follows:

```
// Wrapping around the circular buffer
if (++io_ptr >= CIRCBUF) io_ptr=0;
frame_ctr++;
```

The `process_frame(void)` function is then responsible for ensuring the rotation is performed at the right time. The processing of the frame is then done by converting the spectrum in to time domain.

```
// Initialising the fft spectrum
for (k = 0; k < FFTLEN; ++k) {
    fft_out[k] = cmplx(inframe[k], 0.0);
}
// Performing the fft
fft(FFTLEN, fft_future);
```

2) Estimating the noise spectrum: Since four frames will be used, every $\frac{10}{4} = 2.5s$, $M_i(\omega)$ must be transferred to $M_{i+1}(\omega)$. Subsequently $M_1(\omega) = |X(\omega)|$.

$$M_1(\omega) = min(|X(\omega)|, M_1(\omega)) \tag{2}$$

$$|N(\omega)| = \alpha min_{i=1..4}(M_i(\omega)) \tag{3}$$

The minimum over all M's needs to be taken, such that we obtain the minimum over 10 seconds. The implementation of this is briefly shown in listing II-A.2

```
for(k = 0; k < FFTLEN; ++k) {
    min_i = 0; // Set the minimum index
        to the first element by default
    min_val = M[0][k]; // Set the
        minimum value to the first
        component
    for(i = 1; i < NUM_M; ++i) {
        // If a smaller value is found,
            replace index
        if (M[i][k] < min_val &&
            M[i][k]!= 0) {
            min_val = M[i][k];
            min_i = i;
        }
    }
    noise[k] = M[min_i][k];
}
```

It was decided best to split this computation into its own function, for better decomposition.

3) Subtracting the noise spectrum: In the simplest case, noise spectrum would be subtracted as follows:

$$Y(\omega) = X(\omega) - N(\omega) \tag{4}$$

This can, however, be rewritten as shown in equation 5.

$$\begin{aligned} Y(\omega) &= X(\omega) \times \frac{|X(\omega)| - |N(\omega)|}{|X(\omega)|} \\ &= X(\omega) \times g(\omega) \end{aligned} \tag{5}$$

This form allows a minimum λ to be set, such that:

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

```
for (k = 0; k < FFTLEN; ++k) {
    float g; // Current output for g(w)
    // Calculate the magnitude of
        N(w)/X(w)
    mag_N_X = 1 - noise[k]/X[k];
    // Ensure min value is lambda
    g = mag_N_X > lambda ? mag_N_X :
        lambda;
    // Appropriately multiply a float by
        a complex number
    fft_out[k] = rmul(g, fft_out[k]);
}
```

A simulation in Matlab for this clearly shows the effect of using $g(\omega)$. A random signal composed of two sine waves was generated to represent human speech. Additive noise was then added to the signal, simulating the given audio samples.

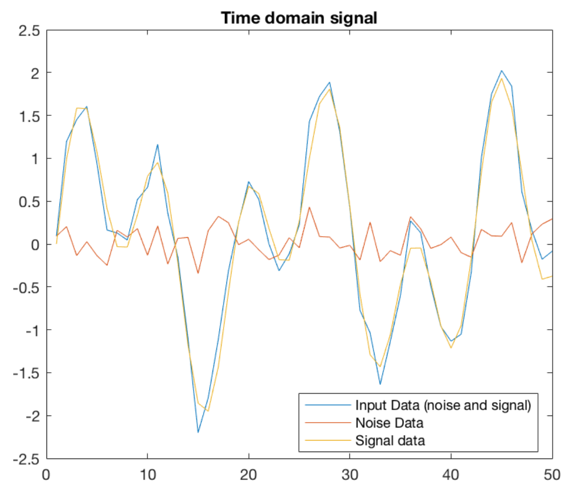

Fig. 2. A time domain representation of the input signals

$g(\omega)$ was then calculated using the frequency spectrum of the noise and original signal. This will, of course, be an ideal case in which the noise and signal are estimated to be exactly correct.

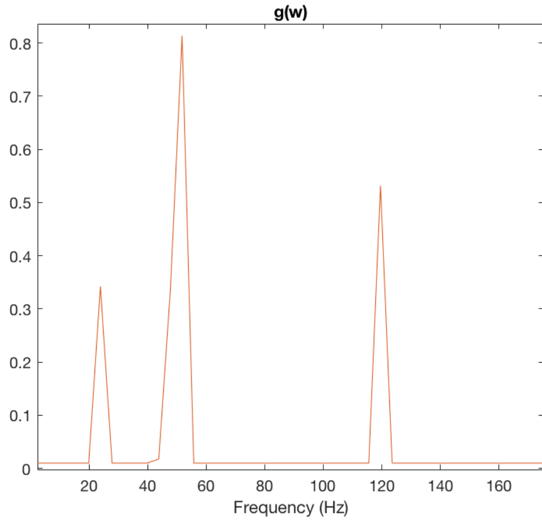

Fig. 3. A frequency domain representation of $g(\omega)$
.

$y(\omega) = x(\omega)$ and $y(\omega) = x(\omega) \times g(\omega)$ are then compared in the frequency domain.

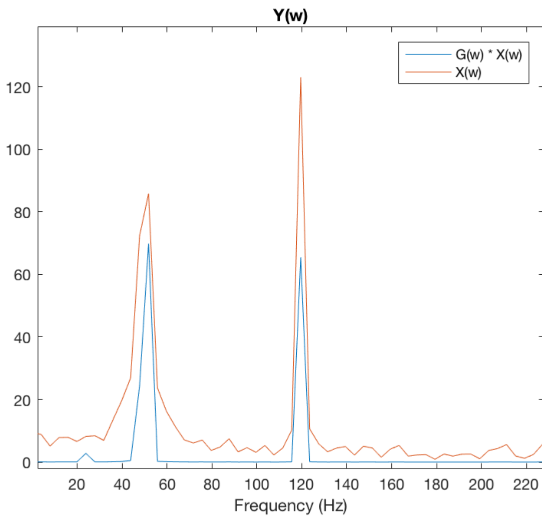

Fig. 4. A frequency domain representation of $y(\omega)$ with and without the $g(\omega)$
.

Clearly it can be observed that the noise spectral components have been reduced, whilst keeping the peaks intact. This is what is expected from this enhancement. One disadvantage of this technique also become apparent from the simulation, that the peaks are not entirely unaffected.

B. Code implementation

An estimation of the spectrum can be taken by storing the minimum magnitude spectrum. Although not accurate, it requires significantly less storage and processing. Therefore it was deemed more efficient to only store the minimum magnitude spectrum in each of the 2.5 second windows: M.

```
float *M[NUM_M];
fft_out  = (complex *) calloc(FFTLEN,
    sizeof(complex)); /* FFT Output */
power_in = (float *) calloc(FFTLEN,
    sizeof(float)); /* Output window */
lpf      = (float *) calloc(FFTLEN,
    sizeof(float)); /* Output window */
mag_in = (float *) calloc(FFTLEN,
    sizeof(float)); /* Output window */
noise = (float *) calloc(FFTLEN,
    sizeof(float)); /* Output window */
```

Before the noise is estimated, the spectrum are written to the M windows. This is defined as a `float *M[NUM_M]`, since only the magnitude spectrum need be stored.

```
unsigned int k;
for(k = 0; k < FFTLEN; ++k) {
   if(X[k] < M[m_ptr][k] || M[m_ptr][k]
      == 0) {
      M[m_ptr][k] = X[k];
   }
}
```

After subtracting the noise as mentioned in the previous section, the inverse fast fourier transform was taken to convert the spectrum back into its time domain. The real part of the result was set to the `outframe`.

```
ifft(FFTLEN, fft_out);

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

III. Enhancements

All implementations were attempted, however the final implementation only included those which had an audible effect on the audio signals. This ensured the code was also kept readable.

A. Enhancement 1: Low pass filter of $|X(\omega)|$

One main enhancement that significantly improves the noise estimation and therefore improves the noise subtraction as well, is low pass filtering the input signal's magnitude response.

Low pass filtering removes the high frequency components from the signal, which are the quick transitions in the signal. By removing those, the noise can be identified more accurately as the signal's fast random transitions will be ignored, which means a more averaged noise is estimated. As the estimated noise spectrum is a more accurate representation of the actual noise spectrum, the α value can also be reduced, as the estimated noise spectrum does not have to be amplified as much anymore.

The filter used to filter the magnitude response of the input uses the following formula to estimate the filtered spectrum.

$$P_t(w) = (1 - e^{-\frac{T}{\tau}}) \times |X(w)| + e^{-\frac{T}{\tau}} \times P_{t-1}(w) \quad (7)$$

Here the time constant is τ, T is the frame rate.

```
#define TFRAME FRAMEINC/FSAMP
```

Where $P_{t-1}(w)$ is the previously estimated low-pass filtered magnitude response.

Taking the Z-transform, we derive the following transfer function in an IIR form:

$$P_f(Z) = (1-e^{-\frac{T}{\tau}}) \times |X_f(Z)| + e^{-\frac{T}{\tau}} \times P_f(Z)Z^{-1} \quad (8)$$

$$H(Z) = \frac{P_f(Z)}{X_f(Z)} = \frac{1 - \exp^{-\frac{T}{\tau}}}{1 - \exp^{-\frac{T}{\tau}} Z^{-1}} \quad (9)$$

```
low_pass_filter(X, lpf);
```

The low pass filter from the equation above can be directly implemented as follows:

```
int w; // Current frequency bin
// Loop through the spectrum to perform
    the low passfilter operation
for (w = 0; w < FFTLEN; ++w) {
    current[w] = (1-K)*current[w] +
        K*next[w];
    // Performs low pass filtering and
        updates the next spectrum
        accordingly
    next[w] = current[w];
}
```

The most appropriate time constant for the speech signal was decided to be around 40-50. To choose the appropriate value for the time constant, certain trade offs had to be made. These are discussed in section IV-A.

B. Enhancement 2: Power Domain

Algorithms

Using the power domain avoids the use of `cabs` for FFT calculations. This is more efficient because it avoids square rooting `FFTLEN` times for each frame process. A slow computation could lead to delays, which could in-turn be heard as echo.

Implementation in the power domain now means that the low pass filter operation is actually:

$$P_t^2(w) = (1-e^{-\frac{T}{\tau}}) \times |X(w)|^2 + e^{-\frac{T}{\tau}} \times P_{t-1}^2(w) \quad (10)$$

Implementation

To implement this enhancement, the `cabs` was simply replaced by the $real^2 + imaginary^2$. This is shown in the code below:

```
// calculate the power spectrum
for (k = 0; k < FFTLEN; ++k) {
    power_in[k] = fft_out[k].r *
        fft_out[k].r + fft_out[k].i *
        fft_out[k].i;
}
```

C. Enhancement 3: Low pass filter noise estimation

Algorithms

The noise estimate $|N(\omega)|$ can be low pass filtered to avoid abrupt discontinuities when the minimisation buffers rotate. This will only be noticeable if the noise level varies greatly.

Implementation

The low pass filter implementation from section III-A is generic on purpose. Therefore it can be easily used with noise as follows.

```
low_pass_filter(noise, next_noise);
```

D. Enhancement 4 and 5: modifying of $g(\omega)$

Algorithms

The minimum value of $g(\omega)$ is currently set to a constant: λ. This value is arbitrarily chosen, between 0.01 and 0.1. Theoretically, it may be possible to

achieve a better approximation by making the minimum value dynamic. By depending on the magnitudes of the signal l(in the magnitude or power domain) and noise, a better approximation for the frequency dependent gain factor can be obtained.

Implementation

The following settings for $g(\omega)$ were attempted: $max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$

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

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

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

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

As in figure 3, the same signal spectrum was also used to derive the value of $g(\omega)$. The result of this is shown in figure 6.

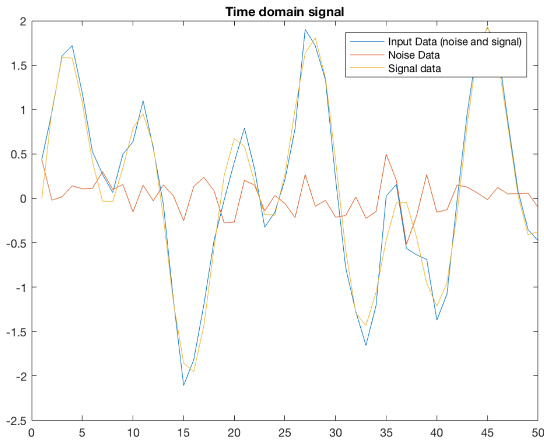

Fig. 5. A time domain representation of the input signals

$g(\omega)$ was then calculated using the frequency spectrum of the noise and original signal. This will, of course, be an ideal case in which the noise and signal are estimated to be exactly correct.

$y(\omega) = x(\omega)$ and $y(\omega) = x(\omega) \times g(\omega)$ are then compared in the frequency domain.

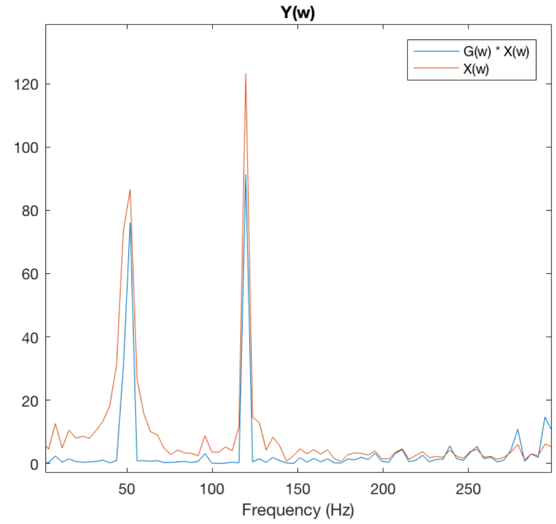

Fig. 6. A frequency domain representation of $y(\omega)$ with and without the $g(\omega)$.

Comparing this to the original $g(\omega) = max(\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|})$ in figure 4, it can be observed that the noise was reduced however more of the smaller spectral components were kept. This is expected as the minimum is now dependent on the signal itself.

It was found that changing the values of $g(\omega)$ did not make any audible change to the speech signals. The original function was therefore restored as: $max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$

The calculation for the function $g(\omega)$ value was conducted in the power domain.

$$max(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}}) \quad (11)$$

```
for (k = 0; k < FFTLEN; ++k) {
   float g;
   // Calculating |N(w)|/|X(w)|
   mag_N_X = sqrt(1 -
       noise[k]/power_in[k]);
   // Setting the maximum of the two
   g = mag_N_X > lambda ? mag_N_X :
       lambda;
   // Outputting the value
   fft_out[k] = rmul(g, fft_out[k]);
}
```

Fundamentally this is implementing a zero-phase filter, which has a real and even frequency response. This is a special case of a linear phase filter with a zero phase slope.

E. Enhancement 6: Over-estimation

Musical noise is a phenomena in which isolated peaks are left subsequent to spectral subtraction. These isolated components form a large difference in magnitude at different frequency bins, which sound like musical noise.

To attempt to attenuate these, overestimation can be used to remove the musical noise as well as the originally detected noise. This is done by increasing the value of α for frequency bins that have a very low SNR. The SNR was estimated by dividing the original signal by the estimated noise, and if that result was lower than a specific threshold, the alpha value at that point was increased. By doing this, there is a greater chance that musical noise will not be left behind after the spectral subtraction, and the effect of musical noise will be reduced.

Usually the threshold would be chosen by experimentation, however, this becomes cumbersome as more thresholds are added to make the over-estimation more specific, and increasing or decreasing α for different SNR bands. Instead it was decided to use the average of the SNR over the current frame and use that to normalize the SNR. Using the normalized SNR, it can then be multiplied by the number of different α values that should be used. Once rounded, this gives an index to the array of α's, and can then be multiplied by the according α. As the lower SNR frequency bins are the most likely to be noise, they should be the ones that are multiplied by the larger α, whereas frequency bins with a large SNR value are most likely part of the actual signal, and should be multiplied by a very low α.

The algorithm can be shown mathematically as follows.

$$
\begin{aligned}
\text{SNR}_i &= \frac{|X(\omega_i)|^2}{|N(\omega_i)|^2} \\
\text{SNR}_{avg} &= \text{average}_i(\text{SNR}_i) \\
\text{noise}_i &\leftarrow \alpha \left[\frac{\text{SNR}_i}{2 \times \text{SNR}_{avg}} \right] \times \text{noise}_i
\end{aligned} \tag{12}
$$

Implementation

The speech samples did not vary a lot in frequency, and therefore it was deemed unnecessary to have more than four values for α, as this was enough to separate the speech from the noise, and multiply the noise by higher α values.

The C code that implements the algorithm described above can be seen below.

```
int i;
float sum;

// Calcualte |signal^2/noise^2| for all
   k
for (i = 0; i < FFTLEN; ++i) {
   SNR[i] = power_in[i] / noise[i];
   sum += SNR[i];
}

// Calculate average
sum /= FFTLEN;

// Use SNRs to divide
for (i = 0; i < FFTLEN; ++i) {
   // Normalising
   SNR[i] /= 2*sum;
   SNR[i] = SNR[i] > 1 ? 1 : SNR[i];
   noise[i] *= alpha[(int)(SNR[i] *
       (NUM_ALPHA-1))];
}
```

The highest SNR was achieved with the values: `float alpha[NUM_ALPHA] = {50, 40, 30, 10}`. This means that when the noise is in a low SNR bin, that it will be multiplied with α value 50, whereas if it is in a high SNR bin, it will be multiplied by α = 10.

It was found that choosing between different α values is a trade off for different speech signals. The average SNR was calculated over the duration of the signal by keeping a global counter, for the SNR and the number of frames processed.

```
snr_val = total_snr / counter;
```

Note the alpha values are squared

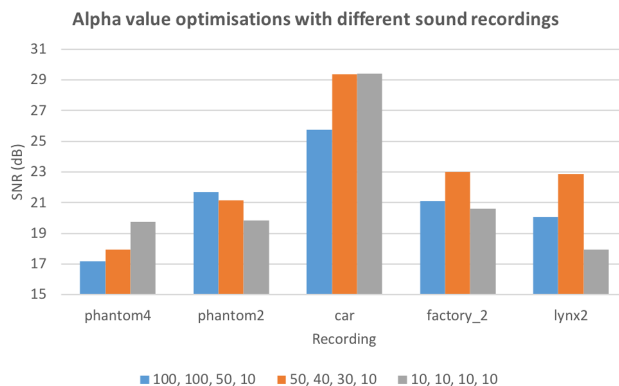

Fig. 7. Average SNR values for optimising α values

Experimental techniques showed that lower alpha values were more suitable for phantom 4. It is true that considering solely the SNR values, the chosen values for alpha may not be optimal. However other factors also needed to be considered when judging the performance of different alpha values, such as crackling. One reason for this may be that filtering increases the time domain amplitude for certain samples, and if this exceeds the available range then it clips or wraps around.

F. Enhancement 7: Frame length

Reducing the frame length from the initial value of 256 to 128 and 64, made the voice of the sound clip sound more distorted and rougher. A larger frame length of 512 samples was also implemented, however, that made the voice in the sound clip sound slurred. The chosen values that were tested had to be a power of two as the FFT algorithm will be applied to it, and it requires the number of samples to be a power of two.

We ended up using the original frame time of 256 samples, as the other options that were tested reduced the quality of the output signal. This can be explained by the fact that experimentally [1], it was found that a frame length of 25-64 ms resulted in the best reconstruction of a voice signal when only using the magnitude response. As the magnitude spectrum of the speech signal was used to estimate and subtract the noise from the original signal, a more optimal frame length to estimate this will give a better result.

Contrary to what the project description mentioned, when decreasing the frame length, the musical noise did not seem to increase. This can be explained by the fact that the musical noise was already very low due to the other enhancements.

The framer length is implemented using a `#define` and assigned to the symbol `FFTLEN`, which can be seen below. By changing the desired length (keeping it a power of 2), different frame lengths can be tested.

```
#define FFTLEN 256
```

G. Enhancement 8: Residual noise reduction

This enhancement was based on [2], in which a similar noise reduction method is used. Theoretically, and according to [2], using residual noise reduction helps reduce the noise that is left after removing the mean, which is the **musical noise**, as described above.

This was implemented as shown below, where `fft_out` is the current output, however, it has one frame delay, as the `fft_future` was the output that was calculated for the current input frame.

```
for (k = 0; k < FFTLEN; ++k) {
    // calculate the future output with
       the noise
    float g;
    mag_N_X = sqrt(1 -
       noise[k]/power_in[k]);
    g = mag_N_X > lambda ? mag_N_X :
       lambda;
    fft_future[k] = rmul(g,
       fft_future[k]);

    // output the fft_out calculated in
       the previous sample, except if
       the SNR is smaller than a
       threshold
    if(power_in_prev[k]/noise_prev[k] <
       threshold) {
       // calculate the minimum of all
          three signals
       fft_out[k] = min(fft_prev[k],
          min(fft_out[k],
          fft_future[k]));
    }
}
```

This enhancement, however, did not seem to improve the quality of the sound output, as the over-estimation seem to already have gotten rid of most of the musical noise that was left over after subtracting the mean of the noise.

H. Enhancement 9: Shorter period

As mentioned in the introduction, the noise estimation is based on finding the minimum over a fixed amount of time, which will likely be a good estimation of the noise. Instead of waiting 10 seconds to get this minimum though, it is much more efficient to calculate it at 4 different times using a period of $1/4 \times 10$. This way, the minimum over these four $M_i(\omega)$ can be found, and the noise can be estimated much earlier. However, when starting the program without initially playing a sound file, the noise is estimated to be λ, as a consequence of Equation 13, as the estimated noise will be negligible. When starting the noisy sound file, the algorithm then took 7.5 seconds to start estimating the actual noise. The reason for this is that for the 7.5 seconds, the minimum for the estimated noise will be lower than λ, and therefore λ will be used for this time, and the algorithm has to wait until the buffer containing the estimated noise of 0 is reset until it can

start estimating the real noise.

$$g(\omega) = max(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}}) \quad (13)$$

To avoid this problem, a shorter period can be used when taking the minimum value of each frequency bin over time, this, however, comes at the cost of signal quality, as the shorter the period for the minimum estimation, the more musical noise was left in the signal. The optimal value that was found for the period, was 1 seconds instead of 2.5 seconds, which was a good compromise between signal quality and speed.

Another enhancement that was performed with respect to the minimum, was reduce the amount of buffers of the minimum $M(\omega)$ that are stored. The initial algorithm was storing four past minimums, however, we decided to reduce this to storing two past minimums instead, as that doubled the noise estimation rate and made it much more reactive.

The number of passed minimums and the period were changed by the following definitions in C.

```
// Sets the update period
#define FRAME_TIME 1
// Calculate the number that has to be
   counted to using a counter in the
   interrupt
#define MAX_COUNT (FRAME_TIME * FSAMP)
// defines how many minimums are stored
#define NUM_M 2
```

IV. Optimisations

A. Parameters: Time constant

Consider again the low pass filter:

$$P_t(w) = (1 - e^{-\frac{T}{\tau}}) \times |X(w)| + e^{-\frac{T}{\tau}} \times P_{t-1}(w) \quad (14)$$

Which gives the transfer function for the IIR filter, as derived earlier:

$$H(Z) = \frac{P_f(Z)}{X_f(Z)} = \frac{1 - e^{-\frac{T}{\tau}}}{1 - e^{-\frac{T}{\tau}} Z^{-1}} \quad (15)$$

In order to better observe the effects of the low pass filtering, a speech signal was modeled along with an additive noise signal. Matlab's filter function was used to model the low pass filter. The code for this is shown below:

```
k = exp(-T/tau);
x = signal + noise;
b = 1-k;
a = [1 -k];
y = filter(b, a, x, [], 2);
```

The simulations for three different values of time constants are shown below:

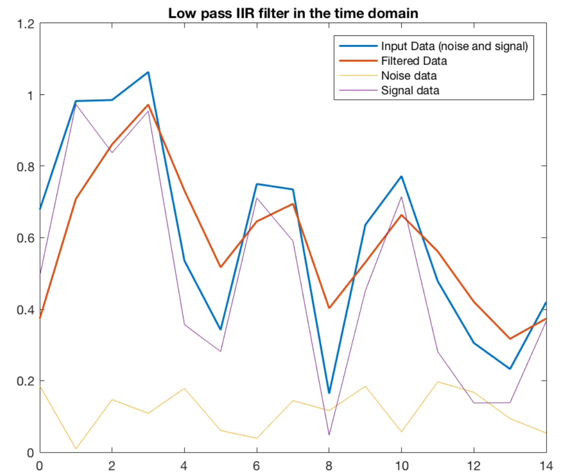

Fig. 8. Low pass IIR filter for a time constant 10 ms

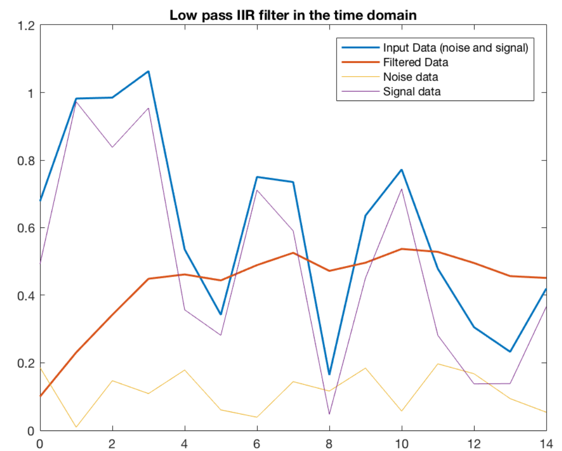

Fig. 9. Low pass IIR filter for a time constant 50 ms

Both the signals were simulated in the time domain, enabling better comparison of the effects on the signal that will be heard. It can be observed that the lower the time constant, the closer the signal is to the input signal. Therefore at a time constant of 10 ms, a lot of noise is expected as less is filtered out. At a time constant of 50 ms, peaks are noticeably filtered out and the signal is therefore expected to have less noise but the original signal data's characteristics may be suppressed.

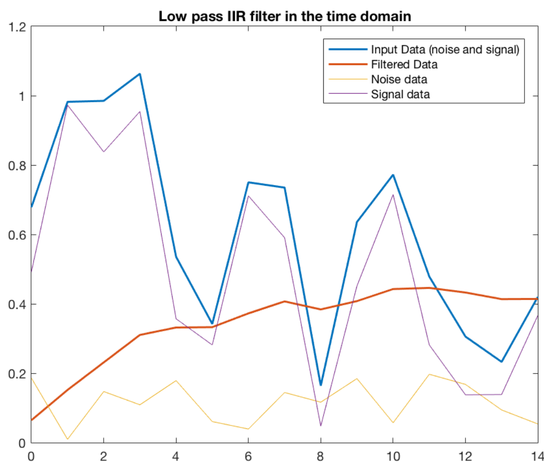

Fig. 10. Low pass IIR filter for a time constant 80ms

Further suppression is observed at larger values of the time constant. Therefore a trade off needed to be made, between the suppression of the signal and the filtering out of the noise. Practically this was found to be ideal at 40 ms. This is therefore what will be used in the final algorithm.

V. Final Algorithm

The final algorithm only included some of the optimisations discussed above, as not all of them improved the noise reduction, and some even decreased its performance. Firstly, the low pass filter in the power domain was left in, because it dramatically reduced the musical noise and the alpha values that had to be used. The low pass filter in the power domain was used, because it made the calculations in the whole frame processing loop more efficient, as there only had to be one `sqrt` function at the end, when the processed frame is output. This low-pass filter was also applied to the noise, so that sudden changes and peaks in the noise did not affect it significantly and it would be more resistant. As we already had the signal in the power domain, $g(\omega)$ could also be estimated in the power domain efficiently by using

$$g(\omega) = max(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}} \quad (16)$$

Overestimation also improved the quality of the output, therefore it was included in the algorithm as well. Finally, the number of minimums that were stored, as well as the time used to estimate the minimum, were reduced to 2 and 1. This degraded the quality of the noise reduction a bit, as the noise was not estimated over a large window anymore, however, it improved the responsiveness of the reduction by quite a lot.

Parameter	Optimal values
λ:	0.05
Time:	$40e-3$
Alpha Values:	$50, 40, 30, 10$

These parameters, even though they performed well on all signals, did perform better on some compared to others. By trying to optimise the parameters on `phantom4.wav`, by dampening the noise and musical noise as much as possible, some of the voice ended up being distorted as well. This meant that, even though it sounded very good on `phantom4.wav`, it sounded a bit more distorted on a simpler noisy signal such as `car1.wav`. Another compromise that was made, is that as we chose a shorter period for the minimum estimation, signals that had varying noise suffered a bit more with the shorter period, as it was not possible to calculate a good average for the noise.

In Figure 11, the spectrogram of the original noisy signal for the `phantom4.wav` file can be seen. The patches in yellow specify the parts in the frequency that have high power, which is measured over time. It can be observed that most of the spectrogram is yellow, which indicates that the signal is very noisy.

Original Noisy Phantom4 Spectrogram

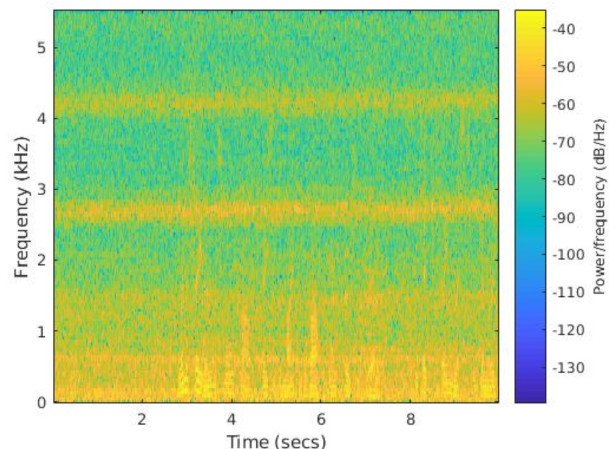

Fig. 11. phantom4.wav original noisy signal spectrogram

After passing it through the noise reduction, the output in Figure 12 can be observed. For the first second, the algorithm does not have the correct minimum stored, and is subtracting 0 from the signal, leaving

the original signal intact. However, after it has found an estimate for the noise, it subtracts it and reduces the noise by quite a lot. Finally, the clean signal can be seen in Figure 13. By comparing the noise reduced output to the clean signal, it can be seen that the parts where the voice is are definitely identified, and the noise reduced.

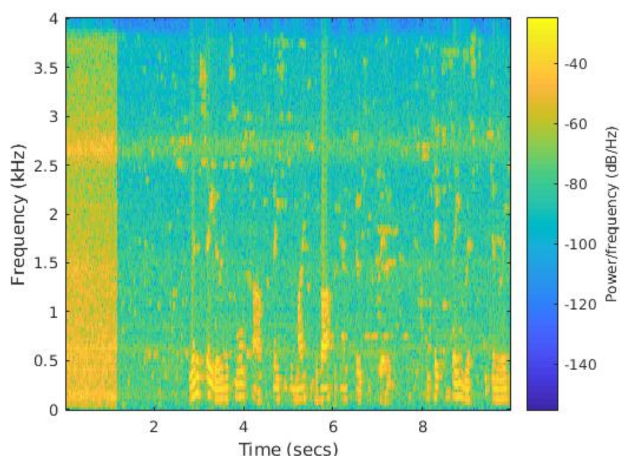

Fig. 12. phantom4.wav spectrogram after it was noise reduced using our optimised parameters

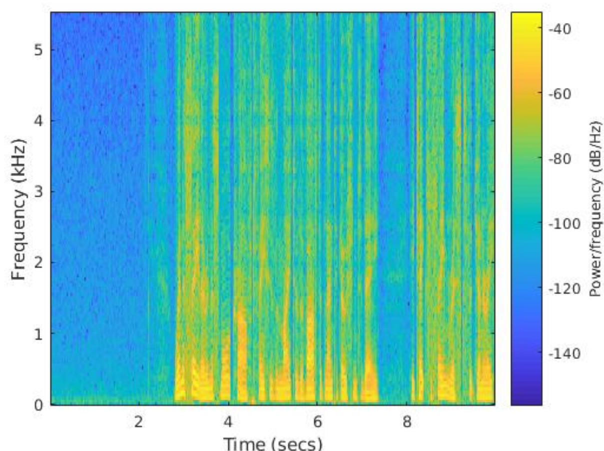

Fig. 13. Spectrogram of clean signal without any noise

REFERENCES

[1] Vahid Montazeri, Soudeh A. Khoubrouy, Issa M. S. Panahi, "A perceptually motivated estimator for speech enhancement", Image and Signal Processing and Analysis (ISPA) 2013 8th International Symposium on, pp. 366-370, 2013.

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

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

[4] 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

[5] 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

Appendix

A. Low Pass filter Simulation Matlab Code

```
T = 0.008;
tau = 10e-03;
k = exp(-T/tau);
lambda = 0.01;

% Randomly generated signal
%signal = rand(2, 15);
signal = [0.4923 0.9727 0.8378 0.9542 0.3569 0.2815  0.7111  0.5906  0.0476  0.4513
    0.7150  0.2815 0.1378 0.1386 0.3662 ;
        0.6947  0.3278  0.7391  0.0319  0.6627  0.2304  0.6246  0.6604  0.3488  0.2409
           0.8562  0.7311 0.8367 0.5882 0.8068 ];
%noise = 0.2*rand(2,15);
noise = [0.1860 0.0095 0.1472 0.1090 0.1787 0.0607  0.0391  0.1444  0.1165  0.1845
    0.0572  0.1970 0.1678 0.0941 0.0538;
       0.0798  0.0685  0.1589  0.1372  0.0110  0.0092  0.1440  0.1756  0.0141  0.1601
          0.1087  0.1431 0.0867 0.1121 0.1498];
x = signal + noise;
b = 1-k;
a = [1 -k];

y = filter(b, a, x, [], 2);

t = 0:length(x)-1;

plot(t, x(1, :),'LineWidth',1.5)
hold on
plot(t,y(1,:),'LineWidth',1.5)
hold on
plot(t, noise(1,:),'LineWidth',0.2)
hold on
plot(t, signal(1,:),'LineWidth',0.2)
legend('Input Data (noise and signal)','Filtered Data', 'Noise data', 'Signal data')
title('Low pass IIR filter in the time domain')
```

B. Zero phase filter Simulation Matlab Code

```
lamba = 0.01;
alpha = 20;
t = 0:.001:.25;
x = sin(2*pi*50*t) + sin(2*pi*120*t);
signal = x;% + 2*randn(size(t));
noise = 0.2*randn(size(t));
y = signal + noise;

figure;
plot(y(1:50))
hold on
plot(noise(1:50))
hold on
plot(signal(1:50))
legend('Input Data (noise and signal)','Noise Data', 'Signal data')
title('Time domain signal')
```

```
figure;
SIGNAL = fft(signal,251);
NOISE = fft(noise,251);
G = max(lambda, 1 - abs(alpha*NOISE)./abs(SIGNAL));
%G = max(lambda*abs(alpha*NOISE)./abs(SIGNAL), 1 - abs(alpha*NOISE)./abs(SIGNAL));
randn(size(t))
Pyy = G.*conj(G)/251;
f = 1000/251*(0:127);
plot(f,G(1:128))
hold on
plot(f,G(1:128))
title('g(w)')
xlabel('Frequency (Hz)')


figure;
Y_fft = fft(y,251);
Y = Y_fft .* G;
randn(size(t))
f = 1000/251*(0:127);
plot(f,abs(Y(1:128)))
hold on
plot(f,abs(Y_fft(1:128)))
title('Y(w)')
xlabel('Frequency (Hz)')
legend('G(w) * X(w)','X(w)')
```

C. *Spectrogram*

```
%%
% Creates the spectrogram for the audio files

[song, fs] = audioread('../audio/best_case/phantom4.wav');
song = song(1:fs*10);
figure
spectrogram(song, 256, [], [], fs, 'yaxis');

%%
[song, fs] = audioread('../audio/original/car1.wav');
song = song(1:fs*10);
figure
spectrogram(song, 256, [], [], fs, 'yaxis');

%%
[song, fs] = audioread('../audio/original/clean.wav');
song = song(1:fs*10);
figure
spectrogram(song, 256, [], [], fs, 'yaxis');
```

D. *Enhance.c*

```
// 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 FRAME_TIME 2
#define MAX_COUNT (FRAME_TIME * FSAMP)
#define MAX_FLOAT 3.4E+38
#define OUTGAIN 16000.0     /* Output gain for DAC */
#define INGAIN (1.0/16000.0)  /* Input gain for ADC */
#define NUM_M 2
#define NUM_ALPHA 4
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP /* time between calculation of each frame */

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

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

```
   0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ-ensure matches FSAMP */\
   0x0001 /* 9 DIGACT Digital interface activation On          */\
          /*********************************************************************/
};

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

float *inbuffer, *outbuffer; /* Input/output circular buffers */
float *inframe, *outframe; /* Input and output frames */
float *inwin, *outwin;     /* Input and output windows */
float ingain, outgain;     /* ADC and DAC gains */
float cpufrac;             /* Fraction of CPU time used */
complex *fft_out;             /* FFT output */
float *noise;
float *power_in;
float *mag_in;
float* p_w;
float* prev_noise;
float* SNR;
volatile int io_ptr=0;     /* Input/ouput pointer for circular buffers */
volatile int frame_ptr=0;  /* Frame pointer */
volatile int frame_ctr = 0;
volatile int m_ptr = 0;
float snr_val = 0;
float total_snr = 0;
float lambda = 0.05;
float alpha[NUM_ALPHA] = {50, 40, 30, 10};
float avg = 0;
float sum = 0;
float *M[NUM_M];
float mag_N_X;
float K;
float time_constant = 40e-3;  /* Time constant in ms */
int started = 0;
 /******************************** 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 write_spectrum(void);
void get_noise(void);
void low_pass_filter(float* current, float* next);
void overestimation(void);
/********************************** Main routine
   ************************************/
void main()
{

   int k; // used in various for loops
   int counter = 1;
/* 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 */
    fft_out  = (complex *) calloc(FFTLEN, sizeof(complex)); /* FFT Output */
    power_in = (float *) calloc(FFTLEN, sizeof(float)); /* Output window */
    p_w      = (float *) calloc(FFTLEN, sizeof(float)); /* Output window */
    mag_in   = (float *) calloc(FFTLEN, sizeof(float)); /* Output window */
    noise    = (float *) calloc(FFTLEN, sizeof(float)); /* Output window */
    prev_noise = (float *) calloc(FFTLEN, sizeof(float)); /* Output window */
    SNR      = (float *) calloc(FFTLEN, sizeof(float)); /* Output window */
    for(k = 0; k < FFTLEN; ++k) {
       SNR[k] = 0;
    }
    /* initialize board and the audio port */
    init_hardware();

    /* initialize hardware interrupts */
    init_HWI();

 /* initialize algorithm constants */

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

    for (k = 0; k < NUM_M; ++k) {
       M[k] = (float *) calloc(FFTLEN, sizeof(float));
    }

    K = exp(-TFRAME/time_constant);
    /* main loop, wait for interrupt */
    while(1) {
       process_frame();
       counter++;
       snr_val = total_snr / counter;
    }
 }

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

}

// Spectrum calculations for the new values
void write_spectrum(void) {
   unsigned int k;
   for(k = 0; k < FFTLEN; ++k) {
      if(power_in[k] < M[m_ptr][k] || M[m_ptr][k] == 0) {
         M[m_ptr][k] = power_in[k];
      }
   }
}

// Noise estimataion
void get_noise(void) {
   int k, i, min_i;
   float min_val;

   for(k = 0; k < FFTLEN; ++k) {
      min_i = 0;
      min_val = M[0][k];
      for(i = 1; i < NUM_M; ++i) {
         if (M[i][k] < min_val && M[i][k]!= 0) {
            min_val = M[i][k];
            min_i = i;
         }
      }
      noise[k] = M[min_i][k];
   }

   overestimation();
```

```
}


void overestimation(void) {
   int i;
   sum = 0;
   // Calcualte |signal^2/noise^2| for all k
   for (i = 0; i < FFTLEN; ++i) {
      if(noise[i] != 0) {
         SNR[i] = power_in[i] / noise[i];
         sum += SNR[i];
      }
   }

   // Calculate average
   sum /= FFTLEN;
   avg = sum;
   total_snr += sum;
   // Use SNRs to divide
   for (i = 0; i < FFTLEN; ++i) {
      // Normalising
      SNR[i] /= 2*sum;
      SNR[i] = SNR[i] > 1 ? 1 : SNR[i];
      noise[i] *= alpha[(int)(SNR[i] * (NUM_ALPHA-1))];
   }
}

// Low pass filter X(w)
void low_pass_filter(float* current, float* next) {
   int w;
   for (w = 0; w < FFTLEN; ++w) {
      current[w] = (1-K)*current[w] + K*next[w];
      next[w] = current[w];
   }
}

/********************************* process_frame()
    ***********************************/
void process_frame(void)
{
   int k, m;
   int io_ptr0;
   /* work out fraction of available CPU time used by algorithm */
   cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;

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

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

   /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where
      the
```

```
data should be read (inbuffer) and saved (outbuffer) for the purpose of
   processing */
io_ptr0=frame_ptr * FRAMEINC;

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

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

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

// Initialise the array fft_out for FFT
for (k = 0; k < FFTLEN; ++k) {
   fft_out[k] = cmplx(inframe[k], 0.0);
}

// Perform the FFT
fft(FFTLEN, fft_out);

// calculate the power spectrum
for (k = 0; k < FFTLEN; ++k) {
   power_in[k] = fft_out[k].r * fft_out[k].r + fft_out[k].i * fft_out[k].i;
}

low_pass_filter(power_in, p_w);
low_pass_filter(noise, prev_noise);

// Get average of fft_out and write to Spectrum
write_spectrum();

// Set the noise
get_noise();

if(frame_ctr > MAX_COUNT-1) {
   int i;
   frame_ctr = 0;
   if(++m_ptr == NUM_M) m_ptr = 0;
   for(i = 0; i < FFTLEN; ++i) {
      M[m_ptr][i] = power_in[i];
   }
}

// max(lambda, |N(w)/g(w)|
for (k = 0; k < FFTLEN; ++k) {
   float g;
   mag_N_X = sqrt(1 - noise[k]/power_in[k]);
   g = mag_N_X > lambda ? mag_N_X : lambda;
   fft_out[k] = rmul(g, fft_out[k]);
}
```

```
    // Back into time domain
    ifft(FFTLEN, fft_out);

    for (k = 0; k < FFTLEN; ++k) {
        outframe[k] = fft_out[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;
    frame_ctr++;
    started = 1;
}

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