Imperial College London

Real Time Digital Signal Processing

Project Report

Andrew Zhou, CID: 00938859

Jagannaath Shiva Letchumanan, CID: 00946740

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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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

March 24, 2017

Contents

1 Introduction

This report details the methodology and techniques used to enhance a speech signal in presence of noise in real time. The algorithm used to enhance the speech signal is spectral subtraction, which operates in the frequency domain and makes the assumption that the spectrum of the input signal can be expressed as the sum of the speech spectrum and the noise spectrum. This method is used as it is a lot simpler to implement than, say a Voice Activity Detector that would estimate the noise in the absence of speech and then subtract it. The main complexities involved in this procedure are in estimating the spectrum of the noise, and subtracting the noise spectrum from the speech. A block diagram of the algorithm is shown below in Figure 1.

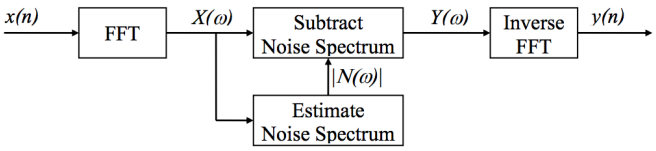

Figure 1: Spectral Subtraction block diagram

1.1 Preprocessing

As the signal can be processed with ease in the frequency domain in contrast to processing in the time domain, the first step in the preprocessing is to split it into frames and take the Fourier Transform of the data using the `fft` function provided in the skeleton code libraries. To exploit the reduced complexity of the fast Fourier Transform algorithm ($Nlog(N)$) as opposed to the N^2 complexity of the discrete Fourier Transform (DFT), the size of the frames being processed must be a power of 2.

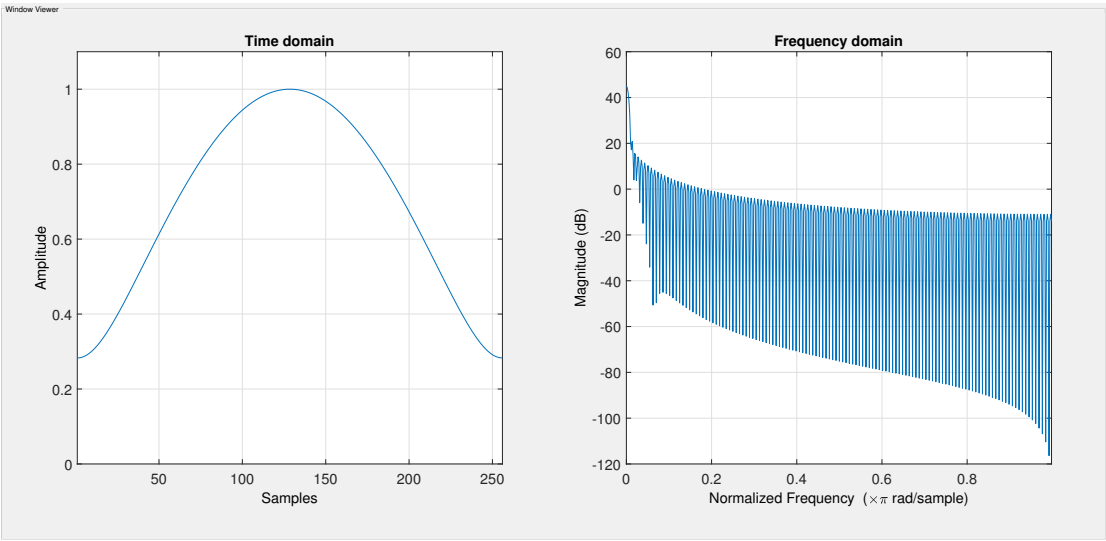

Figure 2: Square root of the Hamming window and its freqeuncy spectrum

If the frames taken do not overlap, taking the FFT will result in spectral artefacts at the frame boundaries, which occurs as a result of the rectangular windowing. This can be avoided by using a window function that doesn't have abrupt discontinuities in the time domain and hence, does not have infinite spectrum in the frequency domain, such as Hanning windows, Hamming windows, etc. In this particular case, the square root of a Hamming window is used, which can be seen in Figure 2.

The windowing, while solving the issue of spectral artefacts, modifies the data itself and hence, this would further need to be corrected, which is done by using Overlap Add processing. This entails splitting a continuous signal into discrete, overlapping frames and adding on the overlapping frames to the corresponding memory locations in the output buffer, while the remaining ones replace the oldest frames in the output buffer. Each frame overlaps its predecessor by three quarters of a frame i.e. it starts a quarter of a frame later, giving an oversampling ratio of 4. A high oversampling ratio implies higher similarity between frames, limiting the abruptness at which gains at frequency bins can change, leading to a smoother sounding output.

The sampling rate of the C6713 is 8kHz and a 256-point FFT will be performed on the input signal every 64 samples, or 8ms. This means that the output will be formed by adding a continuous stream of 256-sample frames, each of which has been multiplied by the square root of a Hamming window. This takes the form:

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

The reason that the square root of the window is taken is due to the fact that, on adding the overlapped frames after aligning them accordingly, they will now always add to 1.

The windowing is done yet again after the inverse FFT (IFFT) at the output in order to eliminate any discontinuities in the time-domain output signal that the processing may have introduced, thereby avoiding crackles in the output sound.

It is also important to note that in discrete time, the Fourier transform is evaluated at 256 equally spaced points around the unit circle, thus dividing the frequency domain signal into 256 equal bins of equal frequency resolution. The frequency resolution of each bin is $FR = F_{max}/N = 8000Hz/256 = 31.25Hz$.

As per this algorithm, 6 frames of quarter length will be undergoing processing at any point in time: one being read from the A/D converter, the immediate past four being processed and the last one being written to the D/A converter. As three frames overlap, these frames, once processed, are added on to the corresponding memory locations in the output buffer. Hence, it would suffice to use buffers of 5 quarter frame lengths (5×64) for the input and output. The inputs are written and the outputs are read using a single circular pointer (`io_ptr`), as the same location in the input buffer which is being written to will be read from, in the output buffer. In addition, another pointer (`frame_ptr`) is used to store the number of the quarter frame in the input buffer (from 1 to 5), up to which the new input must be filled.

1.2 Noise spectrum estimation

The estimation of the noise spectrum, denoted by $N(\omega)$, works under the assumption that the speaker will not speak continuously for more than 10 seconds. Over this time period, the minimum amplitude in each frequency bin of the Fourier transform is taken as the minimum noise spectrum. As this minimum will underestimate the actual noise magnitude, it will be multiplied by an oversubtraction factor α, typically in the range of 3 and 20. A higher α therefore corresponds to more noise subtraction, which could also result in the signal being attenuated giving rise to a muffled voice output.

However, it is unrealistic to store all spectra over the 10 second period from a memory perspective, and so the minimum noise magnitude is taken across four independent 2.5 second intervals, including

the current one. These are named M1, M2, M3 and M4, from newest to oldest. Thus for each subsequent 2.5 second period, these values will be shuffled and the minimum noise magnitude over the new 2.5 second period will fill M1. This is implemented as in Listings 1 and 2:

```
if(counter==0)
{
  //shuffle buffers
  temp = m4;
  m4 = m3;
  m3 = m2;
  m2 = m1;
  m1 = temp;
```

Listing 1: Buffer Shuffling

```
  for(i=0;i<FFTLEN;i++)
  {
    m1[i] = FLT_MAX;
  }
```

Listing 2: Setting M1 to maximum float value so it will be overwritten on comparing to new inputs after 2.5s

Each of these correspond to the minima in each 2.5 second interval. Thus, the noise magnitude is $|N(\omega)| = \alpha min_{i=1,2,3,4}\big(M_i(\omega)\big)$, and is calculated as in Listing 3:

```
    //calculate the minimum estimated noise in each frequency bin over the
  past 10 seconds
    noisemag[i] = m1[i];
    if(noisemag[i]>m2[i])
      noisemag[i] = m2[i];
    if(noisemag[i]>m3[i])
      noisemag[i] = m3[i];
    if(noisemag[i]>m4[i])
      noisemag[i] = m4[i];
    //multiply noise magnitude by oversubtraction factor
    noisemag[i] = ALPHA*noisemag[i];
```

Listing 3: Calculating minimum noise magnitude over past 10 seconds and applying oversubtraction factor

1.3 Noise subtraction

For an input signal $X(\omega)$, the output $Y(\omega)$ is calculated by subtracting the noise $Y(\omega) = X(\omega) - N(\omega)$. To leave the phase unaltered (as the phase of noise is quite difficult to estimate), only the magnitude is subtracted, giving the output as:

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

However, as it is possible for $g(\omega)$ to become negative at some frequency bins, a precaution is taken: $g(\omega) = max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$ where λ can typically be between 0.001 and 0.1. A higher value of lambda will result in a higher noise subtraction and a lower output volume. To strike a balance of voice volume and sufficient noise cancellation, a suitable combination of λ and α must be found. The reason λ is

not set to zero is that the closer it is to zero, the more the musical noise. Hence, it is kept as a small positive constant.

1.4 Output

Once the noise has been estimated, and subtracted, the output complex array has to be converted back to real to write to the D/A Converter. As the input signal was real, its spectrum would be symmetric and hence, despite the processing, the output spectrum will still be symmetric. Hence, the output is expected to be real after the IFFT. However, due to the finite precision associated with floating point numbers used, the complex part of the output frame was, in fact, non-zero but very small. Thus, the real part of the output frame was passed to the output.

2 Original implementation

This basic implementation of spectral subtraction filtered out much of the background noise in the given audio samples. The quality of filtering however is largely dependent on the values of λ and α. Small values of α will not filter out a sufficient amount of noise, while larger values of λ result in a higher volume output as the minimum multiplicative factor $g(\omega)$ will always be larger. This would however compromise the quality of the speech as the signal itself could be attenuated, and so $\lambda = 0.01$ was chosen. This low λ value is balanced out by using a large α of 20.

Although the background noise was filtered, musical noise was introduced as a result of the introduction of λ. This is caused by isolated peaks in the spectrum, which in turn behave like sinusoids and create the effect of musical notes being played, and this is undesirable. As a result, it was required that the original implementation be improved and the enhancements introduced are documented below. These attempted to control the musical noise, as well as to further improve the reduction in the background noise level.

3 Enhancements

3.1 Low-pass Filter

The first enhancement considered was to low-pass filter the magnitude spectrum of the input signal. The noise that corrupts the input signal causes rapid fluctuations in the amplitude spectrum and hence, in order to avoid these, the magnitude spectrum itself would have to be low-pass filtered. If the input magnitude spectrum is $|X(\omega)|$ then the filtered version for frame t is:

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

where $k = e^{-T/\tau}$ is the z-plane pole for time constant τ and frame rate T (the time taken for a quarter frame). The code implementation of this is shown in Listing 4:

```
1    //low-pass filter input signal magnitude
2    p[i] = (1-lpf_const)*mag[i] + lpf_const*p[i];
3    //assign magnitude as the low-pass filtered version
4    mag[i] = p[i];
```

Listing 4: Low-pass filtering input magnitude spectrum

This is a first order IIR filter, as apparent through the z-domain formulation of the transfer function:

$$H(z) = \frac{P(z)}{X(z)} = \frac{1 - e^{\frac{-T}{\tau}}}{1 - z^{-1}e^{\frac{-T}{\tau}}}$$

As the frame rate is fixed, the position of the pole is determined by τ. Thus, it determines the cutoff frequency of the low-pass filter. It does not however affect the stability of the filter, as $e^{-T/\tau}$ is always within the unit circle, and thus the filter will always be stable. It should also be noted that this low-pass filter does not cut off the high frequency signals like in the traditional sense of a filter, it instead cuts off high frequency oscillations in the magnitude spectrum.

The cutoff frequency, ω_c, can be found by solving for $|H(\omega_0)|^2 = \frac{|H(0)|^2}{2}$, as the cut-off frequency is the half-power frequency:

$$\omega_c = cos^{-1}\left(\frac{4k - k^2 - 1}{2k}\right)$$

Using $\tau = 0.08s$, the cutoff frequency is $\omega_c = 0.1rad/s$ or $f_c = 1.98Hz$. Corresponding values for $\tau = 0.02s$ are $\omega_c = 0.405rad/s$ and $f_c = 8.07Hz$. In terms of the speech audio, both eliminated the background noise by a large amount. In order to determine the effect of τ, ω_c is plot as a function of k in Figure 3.

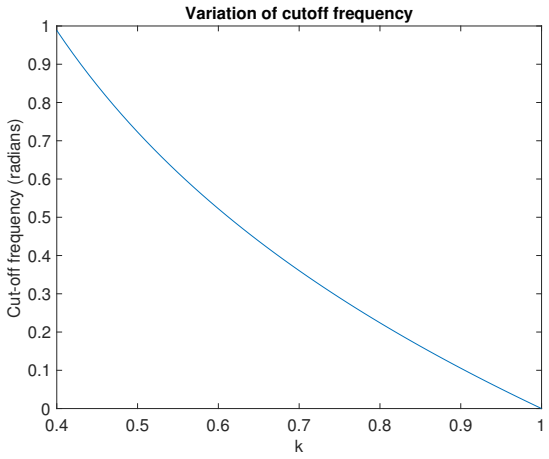

Figure 3: Cut-off frequency (ω_c variation with k)

The smaller the τ, the larger the ratio of T to τ and hence the smaller the k, which means a larger cut-off frequency for the low pass filter. As a result, for a smaller τ more frequency components (of fluctuations in the magnitude spectrum) are allowed through which results in more noise. Due to these findings, and careful inspection of the audio quality depending on τ, enhancement 1 was implemented with $\tau = 0.08s$

It was noted that the over-subtraction factor α could be decreased when this implementation was in place, as the magnitude of the noise spectrum is decreased by the filter. $\alpha = 3$ is used henceforth.

3.2 Low-pass filter in power domain

The next enhancement to be considered is to low-pass filter the magnitude spectrum of the input in the power domain as opposed to the magnitude domain. The low-pass filtered expression therefore

becomes:

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

The code is shown in Listing 5:

```
    //low-pass filter input magnitudes in power domain
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
```

Listing 5: Low-pass filtering input power

The element `p2[i]` on the right hand side is the past value of the filter output. This is reused so as to minimise the use of dynamic arrays and hence, the associated memory overhead.

Speech, in general, is a stationary signal and hence, its power spectrum will not vary much. Noise, on the other hand, is usually not stationary. As a result, the two can be separated in an easier way using the power spectrum filtered version. In addition to this, the non-linear scaling exaggerates differences in magnitude in frequency bins between the speech and noise signals, particularly at higher frequencies. A clearer separation will therefore be observed between noise magnitudes which are small, and voice which is generally large. The resulting effect on the audio is that a high pitch tone which was present in the signal filtered with enhancement 1 is eliminated. As a result, in future enhancements, enhancement 2 is used instead of enhancement 1.

3.3 Low-pass filter noise estimate

Abrupt discontinuities can be observed during times when the buffers rotate. The discontinuities are heard as crackles in the output waveform, and in the frequency domain noise magnitude spectrum, are seen as high frequency fluctuations. These can be avoided by low-pass filtering the noise estimate $|N(\omega)|$. Letting the filtered noise estimate be $M(\omega)$, it can be be constructed as follows:

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

where $k = e^{\frac{-T}{\tau_n}}$. The code for this is shown in Listing 6

```
      //low-pass filter noise magnitude
      noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
```

Listing 6: Low-pass filtering noise magnitude

This is the same first-order IIR filter used in enhancement one with the same dependence of the cut-off frequency on τ. The filters in the previous enhancements do not just target the noise and even remove some of the signal. Adding in this noise filter would target the filtering operation more towards the noise. In addition, the overall implementation responds faster to changes in the signal and the moments in time when the speaker does not speak are a lot more filtered than in the previous enhancements. Ultimately, this enhancement was implemented in the final algorithm.

The noise filtering was also implemented in the power domain, but this approach was abandoned as the outputs were slightly distorted. Moreover, the best value of τ_n was found to be 0.08s yet again.

3.4 Different gain factors

Initially the gain factor $g(\omega) = max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$ with $\lambda = 0.01$ was used to prevent the multiplicative factor from becoming negative. Other options include scaling λ and using the low-pass filtered noise estimate $P(\omega)$ instead of $X(\omega)$. The code for all gain factors tried is shown below in Listing 7:

```
    //calculate the noise to signal ratio
    noiseratio = noisemag[i]/mag[i];
    //different options for g
    switch(option)
    {
      case 1: g[i] = max(LAMBDA,1-noiseratio);
          break;
      case 2: g[i] = max(LAMBDA*noiseratio,1-noiseratio);
          break;
      case 3: g[i] = max(LAMBDA*p2[i]/mag[i],1-noiseratio);
          break;
      case 4: g[i] = max(LAMBDA*noisemag[i]/p2[i],1-noisemag[i]/p2[i]);
          break;
      case 5: g[i] = max(LAMBDA,1-noisemag[i]/p2[i]);
          break;
      default:g[i] = max(LAMBDA,1-noiseratio);
    }
```

Listing 7: Switch case for different options of $g(\omega)$

The switch case allows for g to be altered in real time, allowing for effective comparison between the options. Options 1 (same as enhancement 3) and 2, had audible (but low amplitude) static noise in the background, whereas option 4 had a much clearer static noise. Options 3 and 5 filtered out almost all the background noise, the latter of which, muffled the speech and introduced a deep rumbling in certain tracks. The best gain factor was therefore found to be option 3: $g(\omega) = max(\lambda\frac{|P(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|})$. This was attributed to the fact that as λ introduced the musical noise (explained in section 1.3), scaling it by the ratio of the power spectrum to the magnitude spectrum would help reduce the musical noise as well as background noise in comparison to the other options.

3.5 Oversubtraction factor in power domain

This implementation considers evaluating $g(\omega)$ in the power domain. Option 1 from above would therefore be expressed as: $g(\omega) = max\left(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}}\right)$. An example of this is shown in Listing 8:

```
      case 6: g[i] = max(LAMBDA,sqrt(1-noiseratio*noiseratio));
          break;
```

Listing 8: Calculating oversubtraction factor in power domain

This was similarly applied to all other options tested in the previous section. The resulting effect is that the static background noise that wasn't audible as much in the previous cases, was a lot clearer now and hence, this enhancement was not implemented in the final algorithm.

3.6 Frequency varying noise overestimation

So far the oversubtraction factor (α) has been kept constant. Observations have been made that small α results in high noise, and large α decreases the quality of the speech signal along with the noise.

As a result, a balance was struck, but this is not ideal. One enhancement that can be made is to deliberately overestimate the noise level at frequency bins that have a poor signal to noise ratio (SNR) i.e. increase the oversubtraction factor if the SNR for the bin is low (dominated by noise rather than the signal). As the amplitude of the magnitude and noise signals are being used, the exact SNR using the estimated noise can be calculated by:

$$SNR_{dB} = 20log_{10}\left(\frac{mag[i]}{noisemag[i]}\right)$$

Originally, this was implemented using an algorithm from Berouti et al. [1], based on linearly varying α with SNR for $-5 \leq SNR \leq 20$. Outside this range, α was fixed as constant as can be seen in Listing 9, where `upper_bound` is 20dB, `ALPHA` is 4 and `s` is 6.67.

```
    //caluculate SNR
    SNR = 20*log(mag[i]/noisemag[i]);
    //vary oversubtraction factor depending on SNR
    if((SNR<upper_bound)&&(SNR>-5))
      alpha_cur = ALPHA - SNR/s;
    else if((SNR<0)&&(SNR>-5))
      alpha_cur = ALPHA*gain*gain;
    else if(SNR<-5)
      alpha_cur = ALPHA + 5/s;
```

Listing 9: Linearly varying oversubtraction factor as suggested by Berouti [1]

Another similar algorithm was attempted in which, the SNR is not calculated in logarithms and rather as just a ratio of the magnitude of the input magnitude spectrum to that of the estimate noise. In addition, this algorithm only varied α within some range of frequency bins while keeping it constant outside. This is based on the principle that the bins with the lowest SNR are typically those that do not contain any of the desired signal i.e. human voice, which normally ranges from 85Hz to 255Hz [2]. However, in order to allow for more leeway, the limit on the voice was set as between 0Hz and 1kHz. As mentioned in section 1.1, each frequency bin has a frequency resolution of 31.25Hz. Hence, these limits correspond to bin 0 and bin 30 - 1kHz corresponds to bin 32 but the limit was slightly reduced based on experimentation. In addition, as the magnitude spectrum is symmetric (real input), this would require similar processing on bins 226 to 256 as well. This is implemented in Listing 10:

```
      noisemag[i] = m4[i];
    //low-pass filter noise magnitude
    noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
    if((i<30)||(i>FFTLEN-30))
    {
      //calculate SNR
      SNR = (mag[i]/noisemag[i]);
      //vary oversampling factor depending on SNR
      if(SNR>threshold)
        noisemag[i] = ALPHA*noisemag[i];
      else
        noisemag[i] = ALPHA*gain*noisemag[i];
    }
    else
    {
      //penalise high frequency bins
      noisemag[i] = ALPHA*gain*gain*noisemag[i];
    }
    //calculate g
```

```
20      g[i] = max(LAMBDA*p2[i]/mag[i],1-noisemag[i]/mag[i]);
```

Listing 10: Varying oversubtraction factor with SNR

In the above Listing, `gain` is 10 while `ALPHA` could be reduced to as low as 1. Application of this enhancement vastly reduced the presence of musical noise, as the frequency bins that are associated with the isolated peaks have low SNR and will subsequently be penalised by a large oversubtraction factor. This was due to the fact that, originally, the noise minimum buffer was set once every 2.5s due to the computational complexity of the `log` function for the SNR, whereas now, it is set every cycle and hence, the noise values are very recent. A further reduction in λ to 0.005 helped in improving speech clarity. A major trade-off was made at this point - cancelling noise out almost completely for a small sacrifice in signal clarity, i.e. the voice becomes more muffled.

3.7 Modified Frame Lengths

Experimentation was done on changing the size of the variable `FFTLEN`, which controls the frame length as it determines the number of points around the unit circle that the Fourier transform will be evaluated at, along with the size of data frames being used. Halving `FFTLEN` from 256 to 128 decreases the number of points around the unit circle that the Fourier transform will be evaluated over, and will thus decrease the resolution of each frequency bin. This decreases the performance of the noise subtraction.

Doubling `FFTLEN` to 512 takes double the resolution in terms of frequency bins evaluated, but as a result introduces more possibilities for isolated peaks which create musical noise. In addition, it could not be implemented as it was too slow and used too many computational resources that `cpufrac` was overloaded and appeared as 10% (which is not a possible value). As a result, `FFTLEN` was kept at 256 as a compromise.

3.8 Residual Noise Reduction

As stated before, musical noise is caused by seemingly uncorrelated isolated peaks. Residual noise reduction reduces the effect of this by replacing output frequency bins that correspond to a large spike in noise, with the corresponding frequency bin of adjacent frames. Due to the frame-to-frame randomness of noise, it can be assumed that the adjacent frames do not contain an isolated peak at the same frequency bin.

This is implemented in practice by replacing the current output $Y_t(\omega)$ by the minimum out of three adjacent frames ($Y_{t-1}(\omega)$, $Y_t(\omega)$ and $Y_{t+1}(\omega)$) if $\frac{|N(\omega)|}{|X(\omega)|}$ exceeds a certain threshold [3]. This cancels out noise without affecting speech because of the following reason: A large $\frac{|N(\omega)|}{|X(\omega)|}$ means that the noise is large in comparison to the speech, implying that the current frame corresponds to a time when there is no speech. The main signal at the time will therefore be noise, which is random, and will therefore be suppressed by the aforementioned algorithm.

Three buffers must therefore be initialised to contain the output of three adjacent frames: `out_prev`, which represents the previous output; `out_cur` which represents the current output; and `out_next`, which represents future output. The current input will be processed and assigned to `out_next`. Note that this introduces a one frame delay as out_cur is the default output. Implementation of this is shown in Figure 11.

```
1       //assign current output
```

```
    outframe[i]=out_cur[i];
    //if ratio exceeded, assign output as minimum between two adjacent frames
    if(ratio_prev[i]>=threshold)
    {
      if(cabs(outframe[i])>cabs(out_prev[i]))
      {
        outframe[i] = out_prev[i];
      }
      if(cabs(outframe[i])>cabs(out_next[i]))
      {
        outframe[i] = out_next[i];
      }
    }
    //store noise ratio
    ratio_prev[i] = noiseratio;
```

Listing 11: Residual noise reduction

The performance of this enhancement is dependent on the threshold. If the threshold is too high, $\frac{|N(\omega)|}{|X(\omega)|}$ will never exceed the threshold, and the output will be the same as not implementing the enhancement at all. No improvement to the output will be observed, and the complexity of the algorithm is increased. When the threshold was lowered, it was found that the speech would sound chopped up, especially for the *phantom* tracks. This is due to the noise estimation technique misinterpreting speech as noise, meaning that $\frac{|N(\omega)|}{|X(\omega)|}$ was high during speech and would exceed the threshold. No threshold value would provide a desirable result and as such, this enhancement is not implemented.

3.9 Reduced noise estimation period

By taking the noise estimate over a shorter period than 2.5 seconds, the system can respond more quickly to sudden changes in noise level. The benefits of this are particularly pronounced when switching between tracks and the level of noise changes. Shortening the estimation period can be implemented by reducing the time between buffer rotations and is done as in Listing 12:

```
  //each frame is 8ms. framelen = 312 for 2.5 seconds per rotation: 0.008*312 =
    2.5 to 1 decimal point.
  if(counter == framelen)
    counter = 0;
```

Listing 12: Modified estimation period

However, if the system is made to react too quickly, variations in the speech signal can be read as noise causing the output speech to be distorted. Hence, this was not implemented in the final algorithm.

4 Final Algorithm

The final algorithm implements:

- Enhancement 2 - low-pass filter magnitude spectrum in power domain
- Enhancement 3 - low-pass filter the estimated noise magnitude spectrum
- Enhancement 4(3) - use gain factor $g(\omega) = max(\lambda\frac{|P(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|})$

- Enhancement 6 - Varying oversubtraction factor α depending on SNR of bin

Enhancements 2 and 3 eliminate the majority of the background noise, but leave isolated peaks which result in musical noise. This distortion is then eliminated by enhancements 4(3) and 6.

Computational complexity was considered for the final algorithm. Instead of allowing for real-time switching between options such as the switch case for $g(w)$ in section 3.4, the optimum value was hard-coded in for the final implementation. In addition, `g` was made a single variable rather than an array and other buffers were reused or deleted (if not needed) to save memory.

The final algorithm (Appendix B) was most successful in filtering the lynx2 track. This was confirmed by observing the spectrograms of the filtered and unfiltered signals, shown in Figure 5. The spectrogram of the clean signal is provided as reference in Figure 4. These spectrograms were plot using the VoiceBox add-on in MATLAB, designed by Mr. Mike Brookes [4].

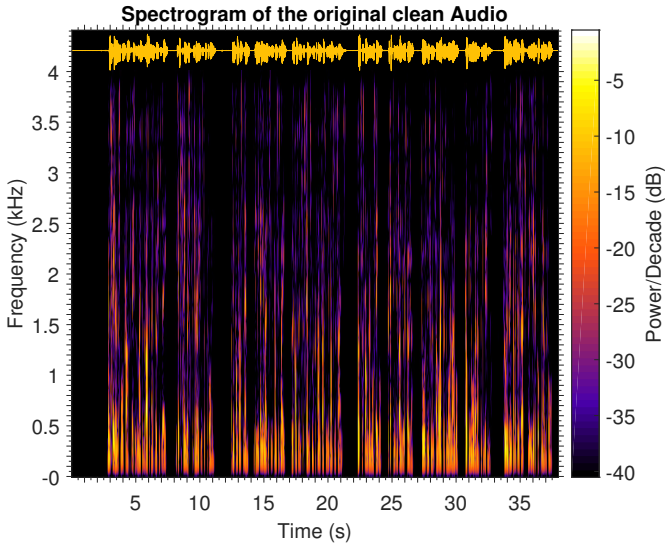

Figure 4: Clean speech signal spectrogram.

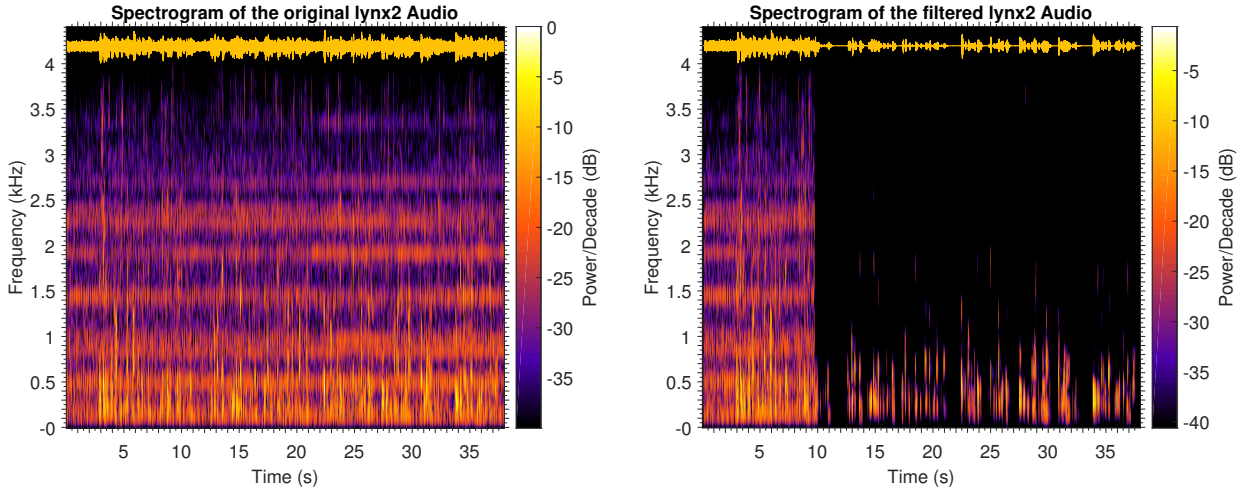

Figure 5: Left: Spectrogram of the unfiltered lynx2 track. Right: Spectrogram of the filtered lynx2 track.

By comparing Figure 4 and Figure 5 Left, it is clear that there is a lot of stationary noise centred at around 2.3kHz, 1.9kHz, 1.5kHz, 1kHz and 0.5kHz. The algorithm completely filters this out, as seen in Figure 5, Right, after the 10 second training time. The speech signal found in lower frequencies is mostly intact and the very low frequency noise was removed. This fact was verified by the clear speech observed in the output audio.

The algorithm was most challenged by the *phantom4* track, whose spectrogram is shown below in Figure 6 (while keeping in mind the 10 second training time):

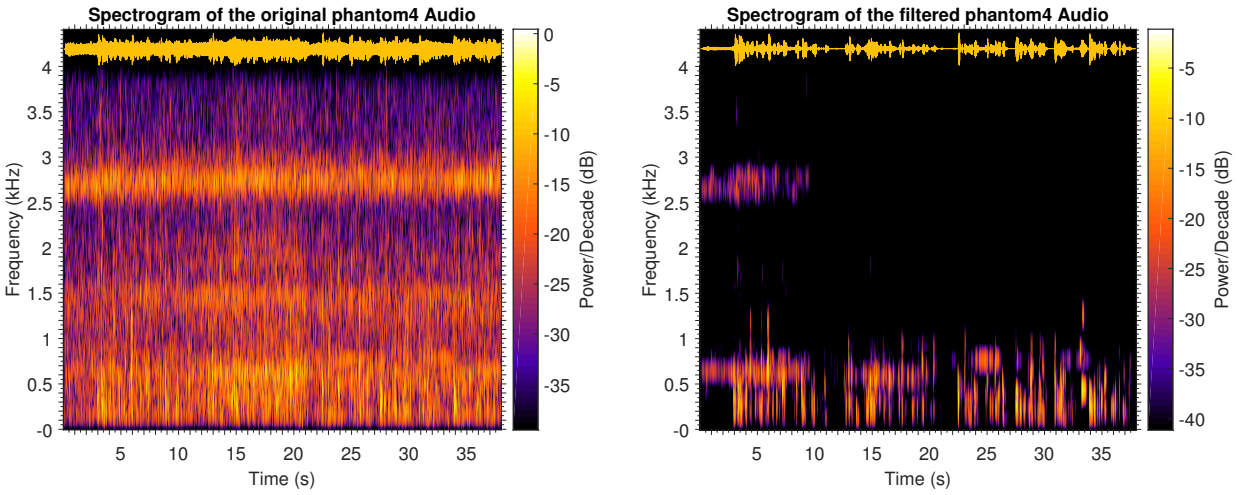

Figure 6: Left: Spectrogram of the unfiltered phantom4 track. Right: Spectrogram of the filtered phantom4 track.

It is clear that the stationary noise at 2.7kHz has been mostly eliminated, but the non-stationary noise at 500Hz is still present. The speech signal itself, found at lower frequencies, has also been slightly distorted. This is because the algorithm erroneously recognises it as noise, as they often appear in the same frequency bins and are of similar magnitudes, for this particular signal. The resulting output audio, although stripped of the majority of the noise, has a muffled speech; a tradeoff which had to be made to extensively filter out the noise.

Spectrograms of the other unfiltered and filtered tracks (by the final algorithm) can be found in the Appendix A.

5 Future work and Conclusions

The original spectral subtraction algorithm did not eliminate a sufficient amount of noise for the speech signal to be absolutely coherent. As a result, several enhancements were explored in order to further improve the output signal. Various trade offs had to be made while choosing between the explored enhancements. Sacrifies such as slight reduction in speech quality were made in favour of major reduction in musical noise. Optimum values for variables such as the oversubtration factor α were identified as well as implementing these major enhancements.

In future, a Voice Activity Detector can be implemented to improve the accuracy of the background noise estimation.

References

[1] H.Schwartz M. Berouti and J. Makhoul. *Enhancement of speech corrupted by acoustic noise.* Bolt Beranek and Newman Inc., 1979.

[2] I.R. Titze. *Principles of Voice Production.* Prentice Hall (currently published by NCVS.org), ISBN 978-0-13-717893-3, 1994.

[3] S.F. Boll. *Suppression of Acoustic Noise in Speech using Spectral Subtraction.* IEEE Transactions on Acoustics, Speech, and Signal Processing, 1979.

[4] M. Brookes. *VOICEBOX: Speech Processing Tool for MATLAB.* Imperial College, London, 1997.

Appendices

A Spectrograms for the original signals and the final algorithm's filtered outputs

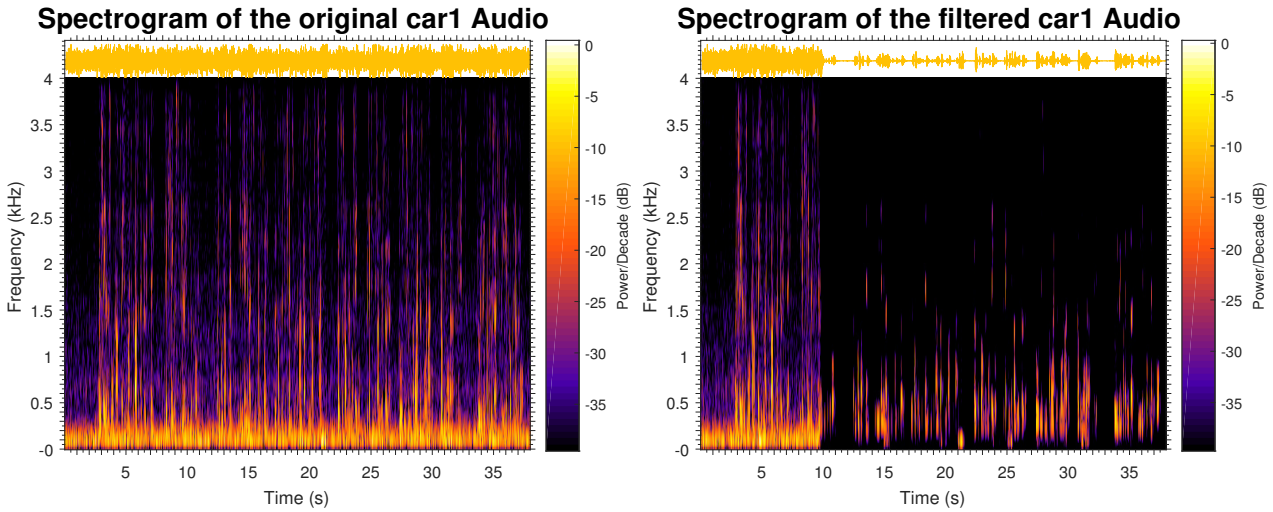

Figure 7: Spectrogram of the original and filtered car1 signal

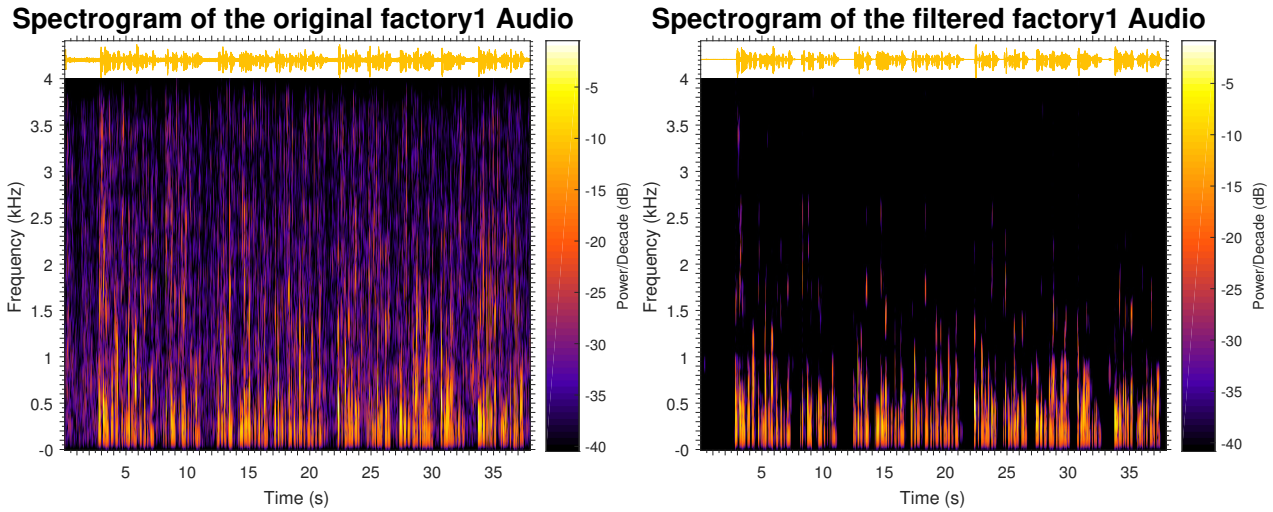

Figure 8: Spectrogram of the original and filtered factory1 signal

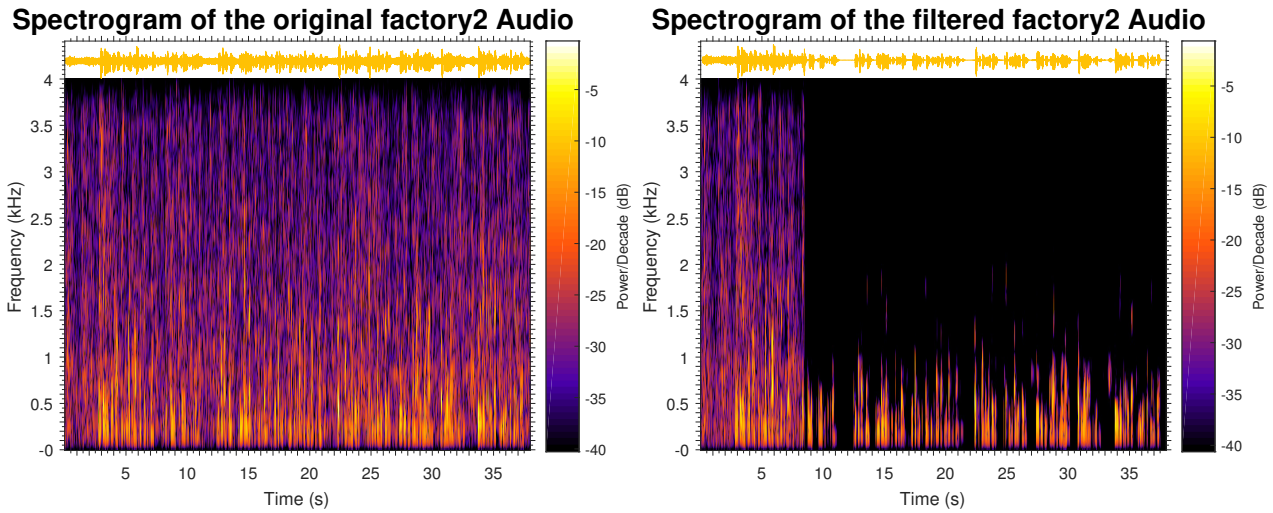

Figure 9: Spectrogram of the original and filtered factory2 signal

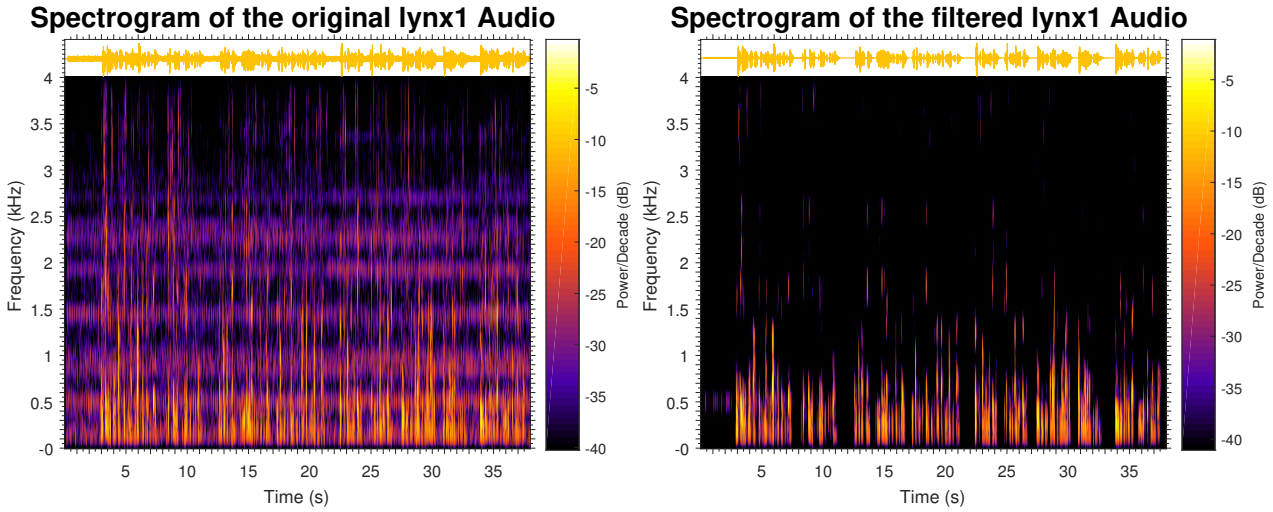

Figure 10: Spectrogram of the original and filtered lynx1 signal

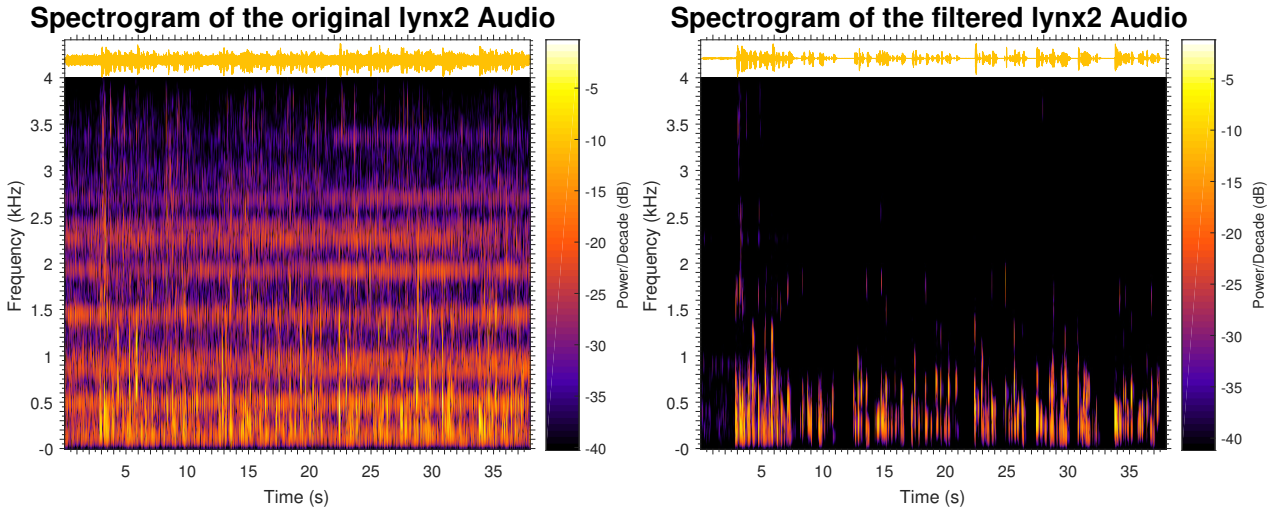

Figure 11: Spectrogram of the original and filtered lynx2 signal

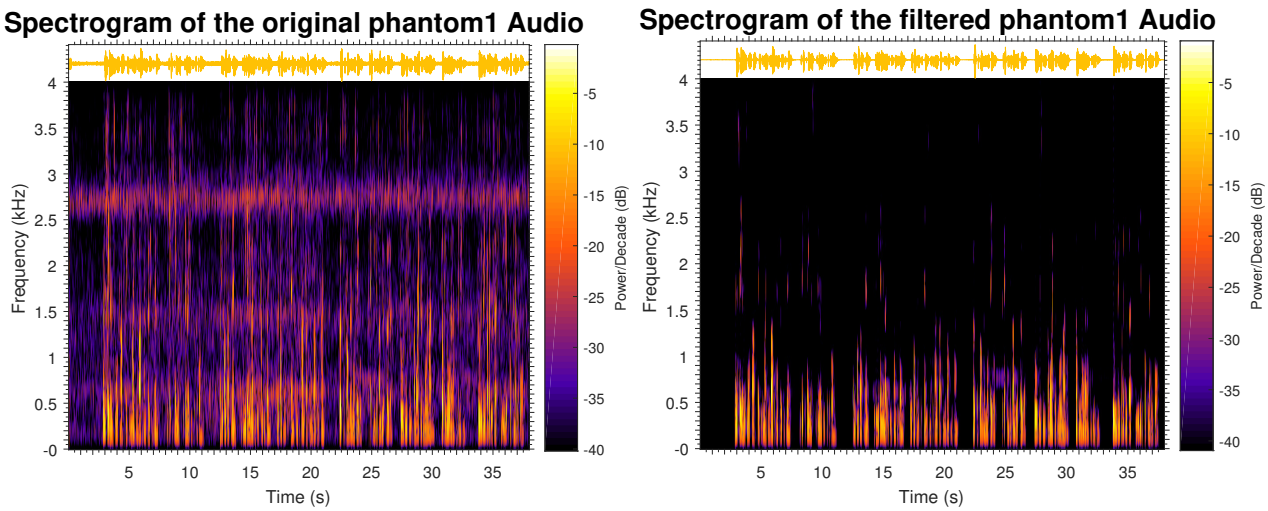

Figure 12: Spectrogram of the original and filtered phantom1 signal

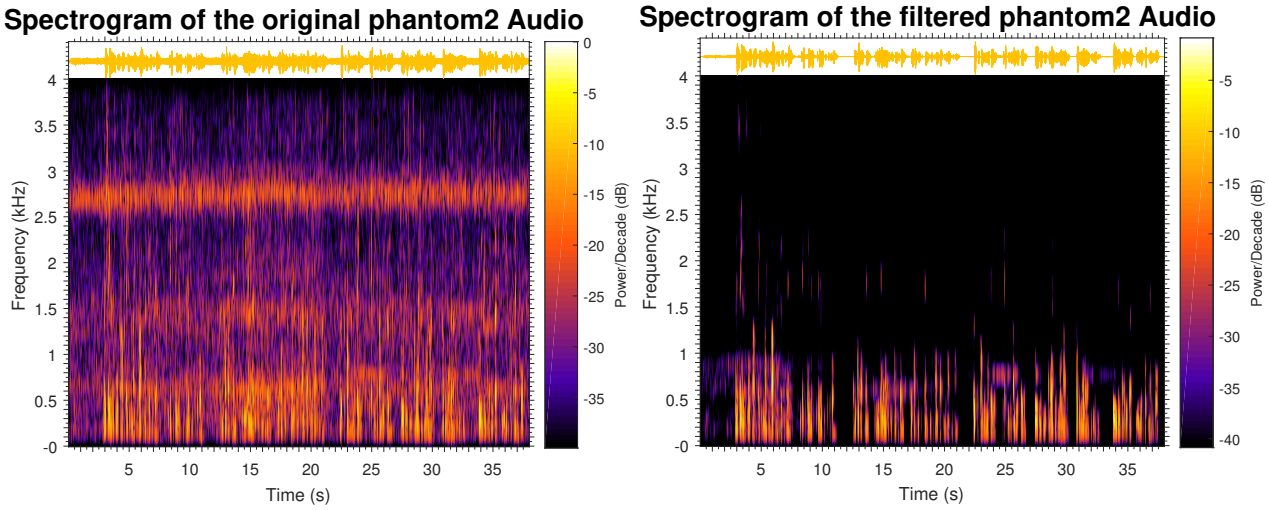

Figure 13: Spectrogram of the original and filtered phantom2 signal

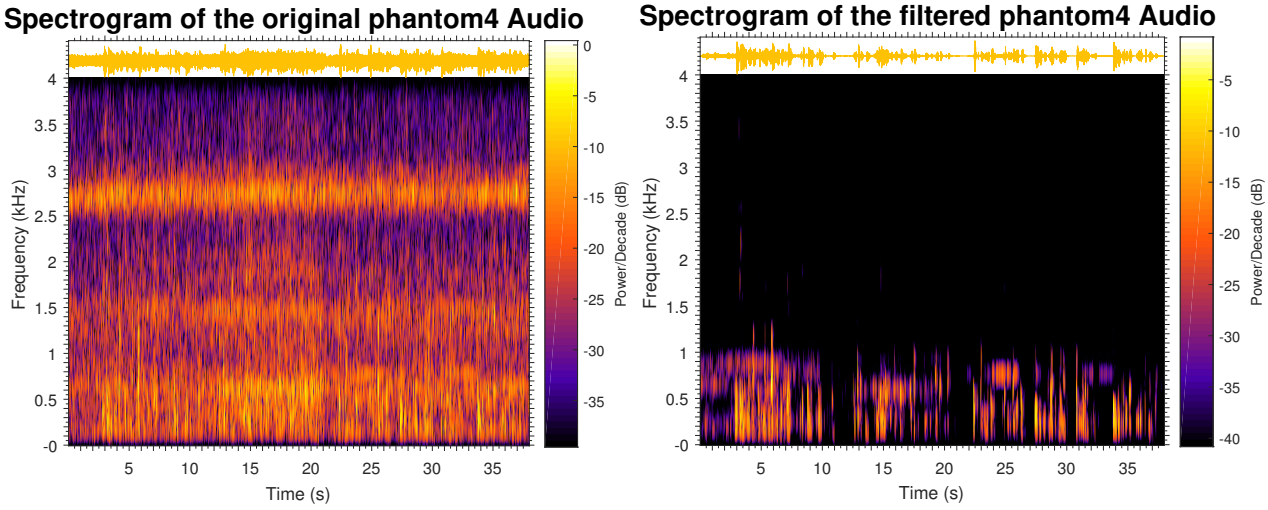

Figure 14: Spectrogram of the original and filtered phantom4 signal

B Final Algorithm

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

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

                        PROJECT: Frame Processing

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

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

 *************************************************************
                       By Danny Harvey: 21 July 2006
                  Updated for use on CCS v4 Sept 2010
 *************************************************************/
/*
 *  You should modify the code so that a speech enhancement project is built
 *  on top of this template.
 */
/***************************** Pre-processor statements
    ******************************/
//  library required when using calloc
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

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

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

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

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

#define WINCONST 0.85185      /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0    /* sample frequency, ensure this matches Config for AIC
    */
#define FFTLEN 256          /* fft length = frame length 256/8000 = 32 ms*/
#define NFREQ (1+FFTLEN/2)      /* number of frequency bins from a real FFT */
#define OVERSAMP 4          /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP)  /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC) /* length of I/O buffers */

#define OUTGAIN 16000.0       /* Output gain for DAC */
```

```
54 #define INGAIN  (1.0/16000.0)   /* Input gain for ADC  */
55 // PI defined here for use in your code
56 #define PI 3.141592653589793
57 #define TFRAME FRAMEINC/FSAMP       /* time between calculation of each frame */
58
59 /****************************** Global declarations
       *******************************/
60
61 /* Audio port configuration settings: these values set registers in the AIC23
       audio
62    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
63 DSK6713_AIC23_Config Config = { \
64       /*************************************************************************/
65       /*   REGISTER              FUNCTION            SETTINGS         */
66       /*************************************************************************/\
67     0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB
          */\
68     0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB
          */\
69     0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB
          */\
70     0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB
          */\
71     0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost
       20dB*/\
72     0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off
          */\
73     0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on
          */\
74     0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit
          */\
75     0x008d,  /* 8 SAMPLERATE Sample rate control        8 KHZ-ensure matches
       FSAMP */\
76     0x0001   /* 9 DIGACT     Digital interface activation    On
          */\
77       /*************************************************************************/
78 };
79
80 // Codec handle:- a variable used to identify audio interface
81 DSK6713_AIC23_CodecHandle H_Codec;
82
83 float *inbuffer, *outbuffer;      /* Input/output circular buffers */
84 complex *inframe, *outframe;          /* Input and output frames */
85 float *inwin, *outwin;              /* Input and output windows */
86 float ingain, outgain;        /* ADC and DAC gains */
87 float cpufrac;             /* Fraction of CPU time used */
88 volatile int io_ptr=0;              /* Input/ouput pointer for circular buffers
       */
89 volatile int frame_ptr=0;           /* Frame pointer */
90 volatile int enable = 1;
91 volatile int counter = 0;
92 float *m1,*m2,*m3,*m4,*mag,*noisemag,*temp,*p2,*noisemag_prev;
93 float ALPHA=1;
94 float LAMBDA=0.005;
95 float TAU = 0.08;
96 float lpf_const,threshold=5,gain=3;
97 float SNR=0,g;
98  /****************************** Function prototypes
```

```
   ******************************/
void init_hardware(void);     /* Initialize codec */
void init_HWI(void);            /* Initialize hardware interrupts */
void ISR_AIC(void);             /* Interrupt service routine for codec */
void process_frame(void);       /* Frame processing routine */
float max(float x, float y);
//float magnitude(complex x);
void enhance();
/********************************* Main routine
   ***********************************/
void main()
{

    int k; // used in various for loops
    lpf_const = exp(-TFRAME/TAU);
/*  Initialize and zero fill arrays */

  inbuffer  = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
    outbuffer = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
  inframe   = (complex *) calloc(FFTLEN, sizeof(complex));  /* Array for
   processing*/
    outframe  = (complex *) calloc(FFTLEN, sizeof(complex));  /* 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));
  m2       = (float *) calloc(FFTLEN, sizeof(float));
  m3       = (float *) calloc(FFTLEN, sizeof(float));
  m4       = (float *) calloc(FFTLEN, sizeof(float));
  mag      = (float *) calloc(FFTLEN, sizeof(float));
  noisemag  = (float *) calloc(FFTLEN, sizeof(float));
  temp     = (float *) calloc(FFTLEN, sizeof(float));
  p2       = (float *) calloc(FFTLEN, sizeof(float));
  noisemag_prev = (float *) calloc(FFTLEN, sizeof(float));

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

    /* initialize hardware interrupts */
    init_HWI();

/* initialize algorithm constants */

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


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

/********************************* init_hardware()
   *********************************/
```

```
152 void init_hardware()
153 {
154     // Initialize the board support library, must be called first
155     DSK6713_init();
156 
157     // Start the AIC23 codec using the settings defined above in config
158     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
159 
160   /* Function below sets the number of bits in word used by MSBSP (serial port)
       for
161   receives from AIC23 (audio port). We are using a 32 bit packet containing two
162   16 bit numbers hence 32BIT is set for  receive */
163   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
164 
165   /* Configures interrupt to activate on each consecutive available 32 bits
166   from Audio port hence an interrupt is generated for each L & R sample pair */
167   MCBSP_FSETS(SPCR1, RINTM, FRM);
168 
169   /* These commands do the same thing as above but applied to data transfers to
       the
170   audio port */
171   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
172   MCBSP_FSETS(SPCR1, XINTM, FRM);
173 
174 
175 }
176 /********************************** init_HWI()
       **************************************/
177 void init_HWI(void)
178 {
179   IRQ_globalDisable();      // Globally disables interrupts
180   IRQ_nmiEnable();        // Enables the NMI interrupt (used by the debugger)
181   IRQ_map(IRQ_EVT_RINT1,4);   // Maps an event to a physical interrupt
182   IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
183   IRQ_globalEnable();       // Globally enables interrupts
184 
185 }
186 
187 /******************************** process_frame()
       **********************************/
188 void process_frame(void)
189 {
190   int m, i;
191   int io_ptr0;
192   /* work out fraction of available CPU time used by algorithm */
193   cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;
194 
195   /* wait until io_ptr is at the start of the current frame */
196   while((io_ptr/FRAMEINC) != frame_ptr);
197   counter++;
198   //if 312 frames have been processed, then roughly 2.5 seconds have passed
199   //each frame is 8ms. 0.008*312 = 2.5 to 1 decimal point.
200   if(counter == 312)
201     counter = 0;
202   /* then increment the framecount (wrapping if required) */
203   if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;
204 
205   /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer)
```

```
    where the
206   data should be read (inbuffer) and saved (outbuffer) for the purpose of
      processing */
207   io_ptr0=frame_ptr * FRAMEINC;
208
209   /* copy input data from inbuffer into inframe (starting from the pointer
      position) */
210
211   m=io_ptr0;
212     for (i=0;i<FFTLEN;i++)
213   {
214     inframe[i] = cmplx(inbuffer[m] * inwin[i],0);
215     if (++m >= CIRCBUF) m=0; /* wrap if required */
216   }
217
218   /************************ DO PROCESSING OF FRAME  HERE *******/
219
220   enhance();
221
222   /***************************************************************/
223
224     /* multiply outframe by output window and overlap-add into output buffer */
225
226   m=io_ptr0;
227     for (i=0;i<(FFTLEN-FRAMEINC);i++)
228   {                           /* this loop adds into outbuffer */
229       outbuffer[m] = outbuffer[m]+(outframe[i].r)*outwin[i];
230     if (++m >= CIRCBUF) m=0; /* wrap if required */
231   }
232     for (;i<FFTLEN;i++)
233   {
234     outbuffer[m] = (outframe[i].r)*outwin[i];   /* this loop over-writes
      outbuffer */
235       m++;
236   }
237 }
238 /************************** INTERRUPT SERVICE ROUTINE  ************/
239
240 // Map this to the appropriate interrupt in the CDB file
241
242 void ISR_AIC(void)
243 {
244   short sample;
245   /* Read and write the ADC and DAC using inbuffer and outbuffer */
246
247   sample = mono_read_16Bit();
248   inbuffer[io_ptr] = ((float)sample)*ingain;
249     /* write new output data */
250   if(enable==0)
251     mono_write_16Bit((short)(sample));
252   else
253     mono_write_16Bit((short)(outbuffer[io_ptr]*outgain));
254   /* update io_ptr and check for buffer wraparound */
255
256   if (++io_ptr >= CIRCBUF) io_ptr=0;
257 }
258
259 /***************************************************************/
```

```
//function returning the larger of two floats
float max(float x, float y)
{
  if(x>y)
    return x;
  else
    return y;
}

void enhance()
{
  int i;
  //fast fourier transform the discrete-time input signal
  fft(FFTLEN,inframe);
  //store previous noise magnitudes
  noisemag_prev = noisemag;
  //check if 2.5 seconds have passed, and therefore buffers are filled
  if(counter==0)
  {
    //shuffle buffers
    temp = m4;
    m4 = m3;
    m3 = m2;
    m2 = m1;
    m1 = temp;
    //assign m1 as maximum float value to take in new values
    for(i=0;i<FFTLEN;i++)
    {
      m1[i] = FLT_MAX;
    }
  }
  for(i=0;i<FFTLEN;i++)
  {
    //calculate magnitude of input spectrum
    mag[i] = cabs(inframe[i]);
    //calculate the minimum estimated noise in each frequency bin over the past
     10 seconds
    noisemag[i] = m1[i];
    if(noisemag[i]>m2[i])
      noisemag[i] = m2[i];
    if(noisemag[i]>m3[i])
      noisemag[i] = m3[i];
    if(noisemag[i]>m4[i])
      noisemag[i] = m4[i];
    //low-pass filter noise magnitude
    noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
    //low-pass filter magnitude in power domain
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
    //assign the minimum magnitude between current and past inputs
    if(m1[i]>mag[i])
      m1[i] = mag[i];
    //vary oversampling factor depending on frequency bin
    if((i<30)||(i>FFTLEN-30))
    {
      //calculate SNR
      SNR = (mag[i]/noisemag[i]);
```

```
      //vary oversampling factor depending on SNR
      if(SNR>threshold)
        noisemag[i] = ALPHA*noisemag[i];
      else
        noisemag[i] = ALPHA*gain*noisemag[i];
    }
    else
    {
      //penalise high frequency bins
      noisemag[i] = ALPHA*gain*gain*noisemag[i];
    }
    //calculate g
    g = max(LAMBDA*p2[i]/mag[i],1-noisemag[i]/mag[i]);
    //calculate output
    outframe[i].r = g*inframe[i].r;
    outframe[i].i = g*inframe[i].i;
  }
  //inverse fourier transform
  ifft(FFTLEN,outframe);
}
```

Listing 13: Code with final algorithm

C Code with all tested enhancements

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

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

                       PROJECT: Frame Processing

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

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

 *************************************************************
                      By Danny Harvey: 21 July 2006
                Updated for use on CCS v4 Sept 2010
 *************************************************************/
/*
 *  You should modify the code so that a speech enhancement project is built
 *  on top of this template.
 */
/***************************** Pre-processor statements
   ******************************/
//  library required when using calloc
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

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

```
#include "dsk6713.h"
#include "dsk6713_aic23.h"

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

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

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

#define WINCONST 0.85185      /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0    /* sample frequency, ensure this matches Config for AIC */
#define FFTLEN 256          /* fft length = frame length 256/8000 = 32 ms*/
#define NFREQ (1+FFTLEN/2)      /* number of frequency bins from a real FFT */
#define OVERSAMP 4          /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP)  /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC) /* length of I/O buffers */

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

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

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

```
      */\
       /************************************************************************/
};

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

float *inbuffer, *outbuffer;      /* Input/output circular buffers */
complex *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 */
volatile int enable = 1;
volatile int enhance = 10;
volatile int option = 5;
volatile int counter = 0;
float *m1,*m2,*m3,*m4,*mag,*noisemag,*temp,*g,*p,*p_prev,*p2,*p2_prev,*
   noisemag_prev,*ratio_prev;
complex *out_prev, *out_cur, *out_next;
float ALPHA=1;
float LAMBDA=0.005;
float TAU = 0.08;
float lpf_const,threshold=5,gain=3;
float threshold2 = 10;
float upper_bound=20;
float s = 6.67, framelen = 312;
float SNR=0;
float SNR_max=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 */
float max(float x, float y);
//float magnitude(complex x);
void original();
void enhance_1();
void enhance_2();
void enhance_3();
void enhance_4();
void enhance_5();
void enhance_6();
void enhance_7();
void enhance_8();
void enhance_9();
void enhance_10();
void enhance_11();
/********************************** Main routine
   ***********************************/
void main()
{

    int k; // used in various for loops
    lpf_const = exp(-TFRAME/TAU);
```

```
/*  Initialize and zero fill arrays */

  inbuffer  = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
    outbuffer = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
  inframe   = (complex *) calloc(FFTLEN, sizeof(complex));  /* Array for
   processing*/
    outframe  = (complex *) calloc(FFTLEN, sizeof(complex));  /* 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));
  m2      = (float *) calloc(FFTLEN, sizeof(float));
  m3      = (float *) calloc(FFTLEN, sizeof(float));
  m4      = (float *) calloc(FFTLEN, sizeof(float));
  mag     = (float *) calloc(FFTLEN, sizeof(float));
  noisemag  = (float *) calloc(FFTLEN, sizeof(float));
  temp    = (float *) calloc(FFTLEN, sizeof(float));
  g     = (float *) calloc(FFTLEN, sizeof(float));
  p     = (float *) calloc(FFTLEN, sizeof(float));
  p_prev    = (float *) calloc(FFTLEN, sizeof(float));
  p2      = (float *) calloc(FFTLEN, sizeof(float));
  p2_prev   = (float *) calloc(FFTLEN, sizeof(float));
  noisemag_prev = (float *) calloc(FFTLEN, sizeof(float));
  out_prev  = (complex *) calloc(FFTLEN, sizeof(complex));
  out_cur   = (complex *) calloc(FFTLEN, sizeof(complex));
  out_next  = (complex *) calloc(FFTLEN, sizeof(complex));
  ratio_prev  = (float *) calloc(FFTLEN, sizeof(float));

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

    /* initialize hardware interrupts */
    init_HWI();

/* initialize algorithm constants */

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


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

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

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

```
185
186   /* Function below sets the number of bits in word used by MSBSP (serial port)
        for
187   receives from AIC23 (audio port). We are using a 32 bit packet containing two
188   16 bit numbers hence 32BIT is set for  receive */
189   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
190
191   /* Configures interrupt to activate on each consecutive available 32 bits
192   from Audio port hence an interrupt is generated for each L & R sample pair */
193   MCBSP_FSETS(SPCR1, RINTM, FRM);
194
195   /* These commands do the same thing as above but applied to data transfers to
        the
196   audio port */
197   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
198   MCBSP_FSETS(SPCR1, XINTM, FRM);
199
200
201 }
202 /********************************* init_HWI()
        ************************************/
203 void init_HWI(void)
204 {
205   IRQ_globalDisable();       // Globally disables interrupts
206   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
207   IRQ_map(IRQ_EVT_RINT1,4);    // Maps an event to a physical interrupt
208   IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
209   IRQ_globalEnable();        // Globally enables interrupts
210
211 }
212
213 /******************************** process_frame()
        **********************************/
214 void process_frame(void)
215 {
216   int m, i;
217   int io_ptr0;
218   /* work out fraction of available CPU time used by algorithm */
219   cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;
220
221   /* wait until io_ptr is at the start of the current frame */
222   while((io_ptr/FRAMEINC) != frame_ptr);
223   //increment counter after each frame
224   counter++;
225   //if 312 frames have been processed, then roughly 2.5 seconds have passed
226   //each frame is 8ms. framelen = 312 for 2.5 seconds per rotation: 0.008*312 =
        2.5 to 1 decimal point.
227   if(counter == framelen)
228     counter = 0;
229   /* then increment the framecount (wrapping if required) */
230   if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;
231
232   /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer)
        where the
233   data should be read (inbuffer) and saved (outbuffer) for the purpose of
        processing */
234   io_ptr0=frame_ptr * FRAMEINC;
235
```

```
236   /* copy input data from inbuffer into inframe (starting from the pointer
        position) */
237
238   m=io_ptr0;
239     for (i=0;i<FFTLEN;i++)
240   {
241     inframe[i] = cmplx(inbuffer[m] * inwin[i],0);
242     if (++m >= CIRCBUF) m=0; /* wrap if required */
243   }
244
245   /************************* DO PROCESSING OF FRAME  HERE
        *************************/
246
247   switch(enhance)
248   {
249     case 0: original();
250         break;
251     case 1: enhance_1();
252         break;
253     case 2: enhance_2();
254         break;
255     case 3: enhance_3();
256         break;
257     case 4: enhance_4();
258         break;
259     case 5: enhance_5();
260         break;
261     case 6: enhance_6();
262         break;
263     case 7: enhance_7();
264         break;
265     case 8: enhance_8();
266         break;
267     case 9: enhance_9();
268         break;
269     case 10: enhance_10();
270         break;
271     case 11: enhance_11();
272         break;
273     default:original();
274   }
275
276 //     for (k=0;k<FFTLEN;k++)
277 //   {
278 //     outframe[k].r = g[i]*inframe[k].r;/* copy input straight into output */
279 //     outframe[k].i = g[i]*inframe[k].i;
280 //   }
281 //
282   /*************************************************************/
283
284     /* multiply outframe by output window and overlap-add into output buffer */
285
286   m=io_ptr0;
287     for (i=0;i<(FFTLEN-FRAMEINC);i++)
288   {                          /* this loop adds into outbuffer */
289       outbuffer[m] = outbuffer[m]+(outframe[i].r)*outwin[i];
290     if (++m >= CIRCBUF) m=0; /* wrap if required */
291   }
```

```
292     for (;i<FFTLEN;i++)
293   {
294     outbuffer[m] = (outframe[i].r)*outwin[i];   /* this loop over-writes
      outbuffer */
295       m++;
296   }
297 }
298 /************************** INTERRUPT SERVICE ROUTINE
      ***************************/
299
300 // Map this to the appropriate interrupt in the CDB file
301
302 void ISR_AIC(void)
303 {
304   short sample;
305   /* Read and write the ADC and DAC using inbuffer and outbuffer */
306
307   sample = mono_read_16Bit();
308   inbuffer[io_ptr] = ((float)sample)*ingain;
309     /* write new output data */
310   if(enable==0)
311     mono_write_16Bit((short)(sample));
312   else
313     mono_write_16Bit((short)(outbuffer[io_ptr]*outgain));
314   /* update io_ptr and check for buffer wraparound */
315
316   if (++io_ptr >= CIRCBUF) io_ptr=0;
317 }
318
319 /*************************************************************/
320 //function returning the larger value between two floats
321 float max(float x, float y)
322 {
323   if(x>y)
324     return x;
325   else
326     return y;
327 }
328
329 void original()
330 {
331   float noiseratio;
332   int i;
333   //fast fourier transform the discrete-time input signal
334   fft(FFTLEN,inframe);
335   //check if 2.5 seconds have passed, and therefore buffers are filled
336   if(counter==0)
337   {
338     //shuffle buffers
339     temp = m4;
340     m4 = m3;
341     m3 = m2;
342     m2 = m1;
343     m1 = temp;
344     for(i=0;i<FFTLEN;i++)
345     {
346       //calculate the minimum estimated noise in each frequency bin over the
      past 10 seconds
```

```
347       noisemag[i] = m1[i];
348       if(noisemag[i]>m2[i])
349         noisemag[i] = m2[i];
350       if(noisemag[i]>m3[i])
351         noisemag[i] = m3[i];
352       if(noisemag[i]>m4[i])
353         noisemag[i] = m4[i];
354       //multiply noise magnitude by oversubtraction factor
355       noisemag[i] = ALPHA*noisemag[i];
356     }
357     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
358     for(i=0;i<FFTLEN;i++)
359     {
360       m1[i] = FLT_MAX;
361     }
362   }
363   for(i=0;i<FFTLEN;i++)
364   {
365     //calculate the magnitude of the input signal as it is currently a complex
      value
366     mag[i] = cabs(inframe[i]);
367     //assign the minimum magnitude between current and past inputs
368     if(m1[i]>mag[i])
369       m1[i] = mag[i];
370     //calculate the noise to signal ratio
371     noiseratio = noisemag[i]/mag[i];
372     //assign g. Precaution for if 1-noiseratio is negative
373     g[i] = max(LAMBDA,1-noiseratio);
374     //calculate the complex output value
375     outframe[i].r = g[i]*inframe[i].r;
376     outframe[i].i = g[i]*inframe[i].i;
377   }
378   //inverse fourier transform
379   ifft(FFTLEN,outframe);
380 }
381
382 void enhance_1() //enhancement 1
383 {
384   float noiseratio;
385   int i;
386   //fast fourier transform the discrete-time input signal
387   fft(FFTLEN,inframe);
388   //check if 2.5 seconds have passed, and therefore buffers are filled
389   if(counter==0)
390   {
391     //shuffle buffers
392     temp = m4;
393     m4 = m3;
394     m3 = m2;
395     m2 = m1;
396     m1 = temp;
397     for(i=0;i<FFTLEN;i++)
398     {
399       //calculate the minimum estimated noise in each frequency bin overt the
      past 10 seconds
400       noisemag[i] = m1[i];
401       if(noisemag[i]>m2[i])
```

```
402         noisemag[i] = m2[i];
403       if(noisemag[i]>m3[i])
404         noisemag[i] = m3[i];
405       if(noisemag[i]>m4[i])
406         noisemag[i] = m4[i];
407       //multiply noise magnitude by oversubtraction factor
408       noisemag[i] = ALPHA*noisemag[i];
409     }
410     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
411     for(i=0;i<FFTLEN;i++)
412     {
413       m1[i] = FLT_MAX;
414     }
415   }
416   for(i=0;i<FFTLEN;i++)
417   {
418     //calculate the magnitude of the input signal as it is currently a complex
      value
419     mag[i] = cabs(inframe[i]);
420     //low-pass filter input signal magnitude
421     p[i] = (1-lpf_const)*mag[i] + lpf_const*p[i];
422     //assign magnitude as the low-pass filtered version
423     mag[i] = p[i];
424     //assign the minimum magnitude between current and past inputs
425     if(m1[i]>mag[i])
426       m1[i] = mag[i];
427     //calculate the noise to signal ratio
428     noiseratio = noisemag[i]/mag[i];
429     //assign g. Precaution for if 1-noiseratio is negative
430     g[i] = max(LAMBDA,1-noiseratio);
431     //calculate the complex output value
432     outframe[i].r = g[i]*inframe[i].r;
433     outframe[i].i = g[i]*inframe[i].i;
434   }
435   //inverse fourier transform
436   ifft(FFTLEN,outframe);
437 }
438
439 void enhance_2() //enhancement 2
440 {
441   float noiseratio;
442   int i;
443   //fast fourier transform the discrete-time input signal
444   fft(FFTLEN,inframe);
445   //check if 2.5 seconds have passe, and therefore buffers are filled
446   if(counter==0)
447   {
448     //shuffle buffers
449     temp = m4;
450     m4 = m3;
451     m3 = m2;
452     m2 = m1;
453     m1 = temp;
454     for(i=0;i<FFTLEN;i++)
455     {
456       //calculate the minimum estimated noise in each frequency bin over the
      past 10 seconds
```

```
457       noisemag[i] = m1[i];
458       if(noisemag[i]>m2[i])
459         noisemag[i] = m2[i];
460       if(noisemag[i]>m3[i])
461         noisemag[i] = m3[i];
462       if(noisemag[i]>m4[i])
463         noisemag[i] = m4[i];
464       //multiply noise magnitude by oversubtraction factor
465       noisemag[i] = ALPHA*noisemag[i];
466     }
467     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
468     for(i=0;i<FFTLEN;i++)
469     {
470       m1[i] = FLT_MAX;
471     }
472   }
473   for(i=0;i<FFTLEN;i++)
474   {
475     //calculate the magnitude of the input signal as it is currently a complex
      value
476     mag[i] = cabs(inframe[i]);
477     //low-pass filter input magnitudes in power domain
478     p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
479     //assign magnitude as square root
480     mag[i] = sqrt(p2[i]);
481     //assign the minimum magnitude between current and past inputs
482     if(m1[i]>mag[i])
483       m1[i] = mag[i];
484     //calculate the noise to signal ratio
485     noiseratio = noisemag[i]/mag[i];
486     //assign g. Precaution for if 1-noiseratio is negative
487     g[i] = max(LAMBDA,1-noiseratio);
488     //calculate complex output value
489     outframe[i].r = g[i]*inframe[i].r;
490     outframe[i].i = g[i]*inframe[i].i;
491   }
492   //inverse fourier transform
493   ifft(FFTLEN,outframe);
494 }
495
496 void enhance_3() //enhancement 3
497 {
498   float noiseratio;
499   int i;
500   //fast fourier transform the discrete-time input signal
501   fft(FFTLEN,inframe);
502   //check if 2.5 seconds have passed, and therefore buffers are filled
503   if(counter==0)
504   {
505     //shuffle buffers
506     temp = m4;
507     m4 = m3;
508     m3 = m2;
509     m2 = m1;
510     m1 = temp;
511     //store previous noise magnitude sepctrum
512     noisemag_prev = noisemag;
```

```
513     for(i=0;i<FFTLEN;i++)
514     {
515       //calculate the minimum estimated noise in each frequency bin overt the
      past 10 seconds
516       noisemag[i] = m1[i];
517       if(noisemag[i]>m2[i])
518         noisemag[i] = m2[i];
519       if(noisemag[i]>m3[i])
520         noisemag[i] = m3[i];
521       if(noisemag[i]>m4[i])
522         noisemag[i] = m4[i];
523         //multiply noise magnitude by oversubtraction factor
524       noisemag[i] = ALPHA*noisemag[i];
525       //low-pass filter noise magnitude
526       noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
527     }
528     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
529     for(i=0;i<FFTLEN;i++)
530     {
531       m1[i] = FLT_MAX;
532     }
533   }
534   for(i=0;i<FFTLEN;i++)
535   {
536     //calculate the magnitude of the input signal as it is currently a complex
      value
537     mag[i] = cabs(inframe[i]);
538     //low-pass filter input magnitudes in power domain
539     p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
540     //assign magnitude as square root
541     mag[i] = sqrt(p2[i]);
542     //assign the minimum magnitude between current and past inputs
543     if(m1[i]>mag[i])
544       m1[i] = mag[i];
545     //calculate the noise to signal ratio
546     noiseratio = noisemag[i]/mag[i];
547     //assign g. Precaution for if 1-noiseratio is negative
548     g[i] = max(LAMBDA,1-noiseratio);
549     //calculate complex output value
550     outframe[i].r = g[i]*inframe[i].r;
551     outframe[i].i = g[i]*inframe[i].i;
552   }
553   //inverse fourier transform
554   ifft(FFTLEN,outframe);
555 }
556 void enhance_4() //enhancement 4
557 {
558   float noiseratio;
559   int i;
560   //fast fourier transform the discrete-time input signal
561   fft(FFTLEN,inframe);
562   //check if 2.5 seconds have passed, and therefore buffers are filled
563   if(counter==0)
564   {
565     //shuffle buffers
566     temp = m4;
567     m4 = m3;
```

```
    m3 = m2;
    m2 = m1;
    m1 = temp;
    //store previous noise magnitude
    noisemag_prev = noisemag;
    for(i=0;i<FFTLEN;i++)
    {
      //calculate the minimum estimated noise in each frequency bin over the
   past 10 seconds
      noisemag[i] = m1[i];
      if(noisemag[i]>m2[i])
        noisemag[i] = m2[i];
      if(noisemag[i]>m3[i])
        noisemag[i] = m3[i];
      if(noisemag[i]>m4[i])
        noisemag[i] = m4[i];
      //multiply noise magnitude by oversubtraction factor
      noisemag[i] = ALPHA*noisemag[i];
      //low-pass filter noise magnitude
      noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
    }
    //set m1 to maximum float value so that all values will be overwritten by
   the magnitudes of the most recent input
    for(i=0;i<FFTLEN;i++)
    {
      m1[i] = FLT_MAX;
    }
  }
  for(i=0;i<FFTLEN;i++)
  {
    //calculate the magnitude of the input signal as it is currently a complex
   value
    mag[i] = cabs(inframe[i]);
    //low-pass filter input magnitudes in power domain
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
    //assign the minimum magnitude between current and past inputs
    if(m1[i]>mag[i])
      m1[i] = mag[i];
    //calculate the noise to signal ratio
    noiseratio = noisemag[i]/mag[i];
    //different options for g
    switch(option)
    {
      case 1: g[i] = max(LAMBDA,1-noiseratio);
          break;
      case 2: g[i] = max(LAMBDA*noiseratio,1-noiseratio);
          break;
      case 3: g[i] = max(LAMBDA*p2[i]/mag[i],1-noiseratio);
          break;
      case 4: g[i] = max(LAMBDA*noisemag[i]/p2[i],1-noisemag[i]/p2[i]);
          break;
      case 5: g[i] = max(LAMBDA,1-noisemag[i]/p2[i]);
          break;
      default:g[i] = max(LAMBDA,1-noiseratio);
    }
    //calculate complex output value
```

```
623     outframe[i].r = g[i]*inframe[i].r;
624     outframe[i].i = g[i]*inframe[i].i;
625   }
626   //inverse fourier transform
627   ifft(FFTLEN,outframe);
628 }
629
630 void enhance_5() //enhancement 5
631 {
632   float noiseratio;
633   int i;
634   //fast fourier transform the discrete-time input signal
635   fft(FFTLEN,inframe);
636   //check if 2.5 seconds have passed, and therefore buffers are filled
637   if(counter==0)
638   {
639     //shuffle buffers
640     temp = m4;
641     m4 = m3;
642     m3 = m2;
643     m2 = m1;
644     m1 = temp;
645     for(i=0;i<FFTLEN;i++)
646     {
647       //calculate the minimum estimated noise in each frequency bin over the
    past 10 seconds
648       noisemag[i] = m1[i];
649       if(noisemag[i]>m2[i])
650         noisemag[i] = m2[i];
651       if(noisemag[i]>m3[i])
652         noisemag[i] = m3[i];
653       if(noisemag[i]>m4[i])
654         noisemag[i] = m4[i];
655       //multiply noise magnitude by oversubtraction factor
656       noisemag[i] = ALPHA*noisemag[i];
657       //low-pass filter noise magnitude
658       noisemag[i] = (1-lpf_const)*noisemag[i];
659     }
660     //set m1 to maximum float value so that all values will be overwritten by
    the magnitudes of the most recent input
661     for(i=0;i<FFTLEN;i++)
662     {
663       m1[i] = FLT_MAX;
664     }
665   }
666   for(i=0;i<FFTLEN;i++)
667   {
668     //calculate the magnitude of the input signal as it is currently a complex
    value
669     mag[i] = cabs(inframe[i]);
670     //low-pass filter input magnitudes in power domain
671     p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
672     //assign magnitude as square root
673     mag[i] = sqrt(p2[i]);
674     //assign the minimum magnitude between current and past inputs
675     if(m1[i]>mag[i])
676       m1[i] = mag[i];
677     //calculate the noise to signal ratio
```

```
678     noiseratio = noisemag[i]/mag[i];
679     //assign g. Precaution for if 1-noiseratio is negative
680     g[i] = max(LAMBDA*p2[i]/mag[i],sqrt(1-noiseratio*noiseratio));
681     //calculate complex output value
682     outframe[i].r = g[i]*inframe[i].r;
683     outframe[i].i = g[i]*inframe[i].i;
684   }
685   //inverse fourier transform
686   ifft(FFTLEN,outframe);
687 }
688
689 void enhance_6() //enhancement 6 with oversubtraction factor varying with
      frequency bin
690 {
691   float noiseratio;
692   int i;
693   //initialise rejection which varies depending on SNR of bin
694   float rejection = ALPHA;
695   //fast fourier transform the discrete-time input signal
696   fft(FFTLEN,inframe);
697   //check if 2.5 seconds have passed, and therefore buffers are filled
698   if(counter==0)
699   {
700     //shuffle buffers
701     temp = m4;
702     m4 = m3;
703     m3 = m2;
704     m2 = m1;
705     m1 = temp;
706     //store previous noise magnitude
707     noisemag_prev = noisemag;
708     for(i=0;i<FFTLEN;i++)
709     {
710       //calculate the minimum estimated noise in each frequency bin over the
      past 10 seconds
711       noisemag[i] = m1[i];
712       if(noisemag[i]>m2[i])
713         noisemag[i] = m2[i];
714       if(noisemag[i]>m3[i])
715         noisemag[i] = m3[i];
716       if(noisemag[i]>m4[i])
717         noisemag[i] = m4[i];
718       //vary the oversubtraction factor depending on frequency bin
719       if(((i<3)||(i>50))&&(i<128))
720         rejection = ALPHA;
721       else if(((i<160)||(i>253))&&(i>128))
722         rejection = 30*ALPHA;
723       else
724         rejection = ALPHA;
725       //multiply by oversubtraction factor
726       noisemag[i] = rejection*noisemag[i];
727       //low-pass filter noise magnitude
728       noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
729     }
730     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
731     for(i=0;i<FFTLEN;i++)
732     {
```

```
      m1[i] = FLT_MAX;
    }
  }
  for(i=0;i<FFTLEN;i++)
  {
    //calculate the magnitude of the input signal as it is currently a complex
    value
    mag[i] = cabs(inframe[i]);
    //low-pass filter input magnitudes in power domain
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
    //assign the minimum magnitude between current and past inputs
    if(m1[i]>mag[i])
      m1[i] = mag[i];
    //calculate the noise to signal ratio
    noiseratio = noisemag[i]/mag[i];
    //different options for g
    switch(option)
    {
      case 1: g[i] = max(LAMBDA,1-noiseratio);
          break;
      case 2: g[i] = max(LAMBDA*noiseratio,1-noiseratio);
          break;
      case 3: g[i] = max(LAMBDA*p2[i]/mag[i],1-noiseratio);
          break;
      case 4: g[i] = max(LAMBDA*noisemag[i]/p2[i],1-noisemag[i]/p2[i]);
          break;
      case 5: g[i] = max(LAMBDA,1-noisemag[i]/p2[i]);
          break;
      case 6: g[i] = max(LAMBDA,sqrt(1-noiseratio*noiseratio));
          break;
      default:g[i] = max(LAMBDA,1-noiseratio);
    }
    //calculate complex output value
    outframe[i].r = g[i]*inframe[i].r;
    outframe[i].i = g[i]*inframe[i].i;
  }
  //inverse fourier transform
  ifft(FFTLEN,outframe);
}

void enhance_7() //enhancement 6 with oversubtraction factor varying with SNR
{
  float noiseratio;
  int i;
  float SNR=0;
  //initialise alpha_cur which varies depending on SNR of bin
  float alpha_cur=ALPHA;
  //fast fourier transform the discrete-time input signal
  fft(FFTLEN,inframe);
  //check if 2.5 seconds have passed, and therefore buffers are filled
  if(counter==0)
  {
    //shuffle buffers
    temp = m4;
    m4 = m3;
    m3 = m2;
```

```
790     m2 = m1;
791     m1 = temp;
792     //store previous noise magnitude
793     noisemag_prev = noisemag;
794     for(i=0;i<FFTLEN;i++)
795     {
796       //calculate the minimum estimated noise in each frequency bin over the
      past 10 seconds
797       noisemag[i] = m1[i];
798       if(noisemag[i]>m2[i])
799         noisemag[i] = m2[i];
800       if(noisemag[i]>m3[i])
801         noisemag[i] = m3[i];
802       if(noisemag[i]>m4[i])
803         noisemag[i] = m4[i];
804       //lopass filter noise magnitude
805       noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
806     }
807     //Calculate SNR
808     SNR = 20*log(p[i]/noisemag[i]);
809     //vary oversubtration factor depending on SNR
810     if((SNR<10)&&(SNR>5))
811       alpha_cur = ALPHA*gain*0.5;
812     else if((SNR<5)&&(SNR>0))
813       alpha_cur = ALPHA*gain;
814     else if((SNR<0)&&(SNR>-5))
815       alpha_cur = ALPHA*gain*gain;
816     else if(SNR<-5)
817       alpha_cur = ALPHA*gain*gain*gain;
818     else
819       alpha_cur = ALPHA;
820     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
821     for(i=0;i<FFTLEN;i++)
822     {
823       m1[i] = FLT_MAX;
824     }
825   }
826   for(i=0;i<FFTLEN;i++)
827   {
828     //calculate the magnitude of the input signal as it is currently a complex
      value
829     mag[i] = cabs(inframe[i]);
830     //low-pass filter input magnitude
831     p[i] = (1-lpf_const)*mag[i] + lpf_const*p[i];
832     //assign magnitude
833     mag[i] = p[i];
834     //assign the minimum magnitude between current and past inputs
835     if(m1[i]>mag[i])
836       m1[i] = mag[i];
837     //calculate noise to signal ratio
838     noiseratio = noisemag[i]/mag[i];
839     //multiply by oversubtraction factor
840     noiseratio = noiseratio*alpha_cur;
841     //options for g
842     switch(option)
843     {
844       case 1: g[i] = max(LAMBDA,1-noiseratio);
```

```
            break;
        case 2: g[i] = max(LAMBDA*noiseratio,1-noiseratio);
            break;
        case 3: g[i] = max(LAMBDA*p[i]/mag[i],1-noiseratio);
            break;
        case 4: g[i] = max(LAMBDA*noisemag[i]/p[i],1-noisemag[i]/p[i]);
            break;
        case 5: g[i] = max(LAMBDA,1-alpha_cur*noisemag[i]/p[i]);
            break;
        case 6: g[i] = max(LAMBDA,sqrt(1-noiseratio*noiseratio));
            break;
        default:g[i] = max(LAMBDA,1-noiseratio);
      }
      //calculate complex output value
      outframe[i].r = g[i]*inframe[i].r;
      outframe[i].i = g[i]*inframe[i].i;
    }
    //inverse fourier transform
    ifft(FFTLEN,outframe);
}

void enhance_8() //enhancement 8
{
    int i;
    //initialise alpha_cur which varies depending on SNR of bin
    float alpha_cur=ALPHA;
    float noiseratio;
    //fast fourier transform the discrete-time input signal
    fft(FFTLEN,inframe);
    //check if 2.5 seconds have passed, and therefore buffers are filled
    if(counter==0)
    {
      //shuffle buffers
      temp = m4;
      m4 = m3;
      m3 = m2;
      m2 = m1;
      m1 = temp;
      //store previous noise magnitude
      noisemag_prev = noisemag;
      //Exploit fft symmetry. Only need to make calculations up to half way
      for(i=0;i<NFREQ;i++)
      {
        //calculate the minimum estimated noise in each frequency bin over the
     past 10 seconds
        noisemag[i] = m1[i];
        if(noisemag[i]>m2[i])
        noisemag[i] = m2[i];
        if(noisemag[i]>m3[i])
          noisemag[i] = m3[i];
        if(noisemag[i]>m4[i])
          noisemag[i] = m4[i];
        //low-pass filter noise magnitude
        noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
        //apply symmetry
        noisemag[FFTLEN-i] = noisemag[i];
      }
      //Calculate SNR
```

```
902     SNR = 20*log(mag[i]/noisemag[i]);
903     //vary oversubtraction factor depending on SNR
904     if((SNR<upper_bound)&&(SNR>0))
905       alpha_cur = ALPHA*gain;
906     else if((SNR<0)&&(SNR>-5))
907       alpha_cur = ALPHA*gain*gain;
908     else if(SNR<-5)
909       alpha_cur = ALPHA*gain*gain*gain;
910     else
911       alpha_cur = ALPHA;
912     //set m1 to maximum float value so that all values will be overwritten by
      the magnitudes of the most recent input
913     for(i=0;i<FFTLEN;i++)
914     {
915       m1[i] = FLT_MAX;
916     }
917   }
918   //rotate buffers storing previous outputs
919   out_prev = out_cur;
920   out_cur = out_next;
921
922   for(i=0;i<FFTLEN;i++)
923   {
924     //calculate the magnitude of the input signal as it is currently a complex
      value
925     mag[i] = cabs(inframe[i]);
926     //low-pass filter input magnitudes in power domain
927     p[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p[i];
928     //assign magnitude as square root
929     mag[i] = sqrt(p[i]);
930     //assign the minimum magnitude between current and past inputs
931     if(m1[i]>mag[i])
932       m1[i] = mag[i];
933     //calculate the noise to signal ratio
934     noiseratio = alpha_cur*noisemag[i]/mag[i];
935     //varying options for g
936     switch(option)
937     {
938       case 1: g[i] = max(LAMBDA,1-noiseratio);
939           break;
940       case 2: g[i] = max(LAMBDA*noiseratio,1-noiseratio);
941           break;
942       case 3: g[i] = max(LAMBDA*p[i]/mag[i],1-noiseratio);
943           break;
944       case 4: g[i] = max(LAMBDA*noisemag[i]/p2[i],1-noisemag[i]/p2[i]);
945           break;
946       case 5: g[i] = max(LAMBDA,1-alpha_cur*noisemag[i]/p[i]);
947           break;
948       case 6: g[i] = max(LAMBDA,sqrt(1-noiseratio*noiseratio));
949           break;
950       default:g[i] = max(LAMBDA,1-noiseratio);
951     }
952     //calculate next output
953     out_next[i].r = g[i]*inframe[i].r;
954     out_next[i].i = g[i]*inframe[i].i;
955     //assign current output
956     outframe[i]=out_cur[i];
957     //if ratio exceeded, assign output as minimum between two adjacent frames
```

```
958     if(ratio_prev[i]>=threshold)
959     {
960       if(cabs(outframe[i])>cabs(out_prev[i]))
961       {
962         outframe[i] = out_prev[i];
963       }
964       if(cabs(outframe[i])>cabs(out_next[i]))
965       {
966         outframe[i] = out_next[i];
967       }
968     }
969     //store noise ratio
970     ratio_prev[i] = noiseratio;
971   }
972   //inverse fourier transform
973   ifft(FFTLEN,outframe);
974 }
975
976 void enhance_9() //enhancement 8 with alternative oversubtraction factor
      variation
977 {
978   int i;
979   //initialise rejection which varies depending on SNR of bin
980   float alpha_cur=ALPHA;
981   float noiseratio;
982   //fast fourier transform the discrete-time input signal
983   fft(FFTLEN,inframe);
984   //check if 2.5 seconds have passed, and therefore buffers are filled
985   if(counter==0)
986   {
987     //shuffle buffers
988     temp = m4;
989     m4 = m3;
990     m3 = m2;
991     m2 = m1;
992     m1 = temp;
993     //store previous noise magnitude
994     noisemag_prev = noisemag;
995     //exploit FFT symmetry
996     for(i=0;i<NFREQ;i++)
997     {
998       //calculate the minimum estimated noise in each frequency bin over the
      past 10 seconds
999       noisemag[i] = m1[i];
1000      if(noisemag[i]>m2[i])
1001      noisemag[i] = m2[i];
1002      if(noisemag[i]>m3[i])
1003        noisemag[i] = m3[i];
1004      if(noisemag[i]>m4[i])
1005        noisemag[i] = m4[i];
1006      //low-pass filter noise magnitude
1007      noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
1008      noisemag[FFTLEN-i] = noisemag[i];
1009    }
1010    //caluculate SNR
1011    SNR = 20*log(mag[i]/noisemag[i]);
1012    //vary oversubtraction factor depending on SNR
1013    if((SNR<upper_bound)&&(SNR>-5))
```

```
      alpha_cur = ALPHA - SNR/s;
    else if((SNR<0)&&(SNR>-5))
      alpha_cur = ALPHA*gain*gain;
    else if(SNR<-5)
      alpha_cur = ALPHA + 5/s;
    else
      alpha_cur = ALPHA + 20/s;
    //set m1 to maximum float value so that all values will be overwritten by
   the magnitudes of the most recent input
    for(i=0;i<FFTLEN;i++)
    {
      m1[i] = FLT_MAX;
    }
  }
  //rotate buffers storing previous outputs
  out_prev = out_cur;
  out_cur = out_next;

  for(i=0;i<FFTLEN;i++)
  {
    //calculate the magnitude of the input signal as it is currently a complex
   value
    mag[i] = cabs(inframe[i]);
    //low-pass filter input magnitudes in magnitude domain
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
    //assign the minimum magnitude between current and past inputs
    if(m1[i]>mag[i])
      m1[i] = mag[i];
    //calculate the noise to signal ratio
    noiseratio = alpha_cur*noisemag[i]/mag[i];
    //varying options for g
    switch(option)
    {
      case 1: g[i] = max(LAMBDA,1-noiseratio);
          break;
      case 2: g[i] = max(LAMBDA*noiseratio,1-noiseratio);
          break;
      case 3: g[i] = max(LAMBDA*p2[i]/mag[i],1-noiseratio);
          break;
      case 4: g[i] = max(LAMBDA*noisemag[i]/p2[i],1-noisemag[i]/p2[i]);
          break;
      case 5: g[i] = max(LAMBDA,1-alpha_cur*noisemag[i]/p2[i]);
          break;
      case 6: g[i] = max(LAMBDA,sqrt(1-noiseratio*noiseratio));
          break;
      default:g[i] = max(LAMBDA,1-noiseratio);
    }
    //calculate next output
    out_next[i].r = g[i]*inframe[i].r;
    out_next[i].i = g[i]*inframe[i].i;
    //assign current output
    outframe[i]=out_cur[i];
    //if ratio exceeded, assign output as minimum between two adjacent frames
    if(ratio_prev[i]>=threshold)
    {
      if(cabs(outframe[i])>cabs(out_prev[i]))
```

```
      {
        outframe[i] = out_prev[i];
      }
      if(cabs(outframe[i])>cabs(out_next[i]))
      {
        outframe[i] = out_next[i];
      }
    }
    //store noise ratio
    ratio_prev[i] = noiseratio;
  }
  //inverse fourier transform
  ifft(FFTLEN,outframe);
}

void enhance_10() //final algorithm
{
  int i;
  //fast fourier transform the discrete-time input signal
  fft(FFTLEN,inframe);
  //store previous noise magnitudes
  noisemag_prev = noisemag;
  //check if 2.5 seconds have passed, and therefore buffers are filled
  if(counter==0)
  {
    //shuffle buffers
    temp = m4;
    m4 = m3;
    m3 = m2;
    m2 = m1;
    m1 = temp;
    //assign m1 as maximum float value to take in new values
    for(i=0;i<FFTLEN;i++)
    {
      m1[i] = FLT_MAX;
    }
  }
  for(i=0;i<FFTLEN;i++)
  {
      //calculate magnitude of input spectrum
    mag[i] = cabs(inframe[i]);
    //low-pass filter magnitude in power domain
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
    //assign the minimum magnitude between current and past inputs
    if(m1[i]>mag[i])
      m1[i] = mag[i];
    //vary oversampling factor depending on frequency bin
    //calculate the minimum estimated noise in each frequency bin over the past
   10 seconds
    noisemag[i] = m1[i];
    if(noisemag[i]>m2[i])
      noisemag[i] = m2[i];
    if(noisemag[i]>m3[i])
      noisemag[i] = m3[i];
    if(noisemag[i]>m4[i])
      noisemag[i] = m4[i];
```

```
    //low-pass filter noise magnitude
    noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
    if((i<30)||(i>FFTLEN-30))
    {
      //calculate SNR
      SNR = (mag[i]/noisemag[i]);
      //vary oversampling factor depending on SNR
      if(SNR>threshold)
        noisemag[i] = ALPHA*noisemag[i];
      else
        noisemag[i] = ALPHA*gain*noisemag[i];
    }
    else
    {
      //penalise high frequency bins
      noisemag[i] = ALPHA*gain*gain*noisemag[i];
    }
    //calculate g
    g[i] = max(LAMBDA*p2[i]/mag[i],1-noisemag[i]/mag[i]);
    //calculate output
    outframe[i].r = g[i]*inframe[i].r;
    outframe[i].i = g[i]*inframe[i].i;
  }
  //inverse fourier transform
  ifft(FFTLEN,outframe);
}

void enhance_11() //final algorithm with enhancement 8, for testing purposes
{
  float noiseratio;
  int i;
  //fast fourier transform the discrete-time input signal
  fft(FFTLEN,inframe);
  //store previous noise magnitudes
  noisemag_prev = noisemag;
  //check if 2.5 seconds have passed, and therefore buffers are filled
  if(counter==0)
  {
    //shuffle buffers
    temp = m4;
    m4 = m3;
    m3 = m2;
    m2 = m1;
    m1 = temp;
    //assign m1 as maximum float value to take in new values
    for(i=0;i<FFTLEN;i++)
    {
      m1[i] = FLT_MAX;
    }
  }
  //shuffle buffers contining adjacent outputs
  out_prev = out_cur;
  out_cur = out_next;
  for(i=0;i<FFTLEN;i++)
  {
      //calculate input magnitude
    mag[i] = cabs(inframe[i]);
    //low-pass filter input magnitude in power domain
```

```
    p2[i] = (1-lpf_const)*mag[i]*mag[i] + lpf_const*p2[i];
    //assign magnitude as square root
    mag[i] = sqrt(p2[i]);
    //assign the minimum magnitude between current and past inputs
    if(m1[i]>mag[i])
      m1[i] = mag[i];
    //vary oversampling factor depending on frequency bin
    //calculate the minimum estimated noise in each frequency bin over the past
   10 seconds
    noisemag[i] = m1[i];
    if(noisemag[i]>m2[i])
      noisemag[i] = m2[i];
    if(noisemag[i]>m3[i])
      noisemag[i] = m3[i];
    if(noisemag[i]>m4[i])
      noisemag[i] = m4[i];
    //low-pass filter noise magnitude
    noisemag[i] = (1-lpf_const)*noisemag[i]+lpf_const*noisemag_prev[i];
    if((i<30)||(i>FFTLEN-30))
    {
      //calculate SNR
      SNR = (mag[i]/noisemag[i]);
      //vary oversampling factor depending on SNR
      if(SNR>threshold)
        noisemag[i] = ALPHA*noisemag[i];
      else
        noisemag[i] = ALPHA*gain*noisemag[i];
    }
    else
    {
      //penalise high frequency bins
      noisemag[i] = ALPHA*gain*gain*noisemag[i];
    }
    //calculate noise to signal ratio
    noiseratio = noisemag[i]/mag[i];
    //calculate g
    g[i] = max(LAMBDA*p2[i]/mag[i],1-noisemag[i]/mag[i]);
    //calculate output
    out_next[i].r = g[i]*inframe[i].r;
    out_next[i].i = g[i]*inframe[i].i;
    //assign outout
    outframe[i]=out_cur[i];
    //if ratio exceeds threshold, assign output as minimum of adjacent frames
    if(ratio_prev[i]>=threshold2)
    {
      if(cabs(outframe[i])>cabs(out_prev[i]))
      {
        outframe[i] = out_prev[i];
      }
      if(cabs(outframe[i])>cabs(out_next[i]))
      {
        outframe[i] = out_next[i];
      }
    }
    //store noise ratio
    ratio_prev[i] = noiseratio;
  }
  //inverse fourier transform
```

```
1242    ifft(FFTLEN,outframe);
1243 }
```

Listing 14: Code with all enhancements. Real-time switching between enhancement for comparison is possible