EE3-19 Real Time Digital Signal Processing

Coursework Report 5: Speech Enhancement Project

Abstract – This report details an algorithm used to remove background noise from speech, modelling the real word application of noise cancellation in mobile phones. Several enhancements are proposed and implemented to the basic algorithm, and the best combination of enhancements are proposed at the end, along with spectograms showing the spectrum of the input audio file before and after processing.

Jeremy Chan*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
jc4913@ic.ac.uk | 00818433

Chak Yeung Dominic Kwok*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
cyk113@ic.ac.uk | 00827832

*These authors contributed equally to this work

Table of Contents

"When a sailor in a small craft faces the might of the vast Atlantic Ocean today, he takes same risks that generations took before him. But in contrast to them, he can meet any emergency that comes his way, with the confidence that stems from a profound trust in the advances of science. Boats are stronger and more stable, protecting against undue exposure. Tools and instruments are more accurate and more reliable, helping in all weather and conditions. Food and drink are better researched and easier to cook than ever before." – Mike Brookes

I. INTRODUCTION

Mobile phone calls take place in a range of environments, with a variable level of noise present during the conversation. An algorithm is proposed to reduce the effect of this noise without compromising the voice clarity. Since the main purpose of a conversation is to exchange information, the clarity of speech is higher in priority than other factors such as background and musical noise. Difference enhancements are provided and their effect explained, along with justifications if they are implemented in the final design.

II. BACKGROUND

This project aims to reduce noise by frame processing. Frame processing is a method of processing real time signals in consecutive blocks of sampled data. However, performing the FFT on a frame gives rise to discontinuities in the frame due to the rectangular windowing, and hence side lobes. The obvious go-to is a windowing function to smooth the frame edges to zero amplitude, but this then introduces varying signal amplitudes post-IDFT.

Hence, the overlap-add method was used. It is an efficient way of obtaining a spectrum in the frequency domain [1]. As the incoming audio stream is infinitely long, using the overlap-add method achieves our aim of real time noise reduction as a frame can be output after a short delay. Additionally, the oversampling factor was chosen to be 4 as the square root of a Hamming window when doing 4 times frame processing add to 1. To account for time domain discontinuities post-processing, the output is also windowed.

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

III. THE SPECTRAL SUBTRACTION ALGORITHM

The spectral subtraction algorithm aims at reducing the background noise from an audio signal by subtracting the estimated noise spectrum from the estimated signal spectrum, increasing the signal to noise ratio (SNR). We chose to do the processing over four frames and not $312.5 \times 4 = 1250$ frames as the significant reduction in both calculation and storage outweigh the accuracy loss. These four frames form the noise minimum buffer.

A. Estimating the Noise Spectrum

The minimum magnitude of the input signal over the last 10 seconds is assumed to be the background noise as the speaker is unlikely to talk for 10 consecutive seconds without break. This noise minimum is then multiplied by a factor to get the average noise.

In the ideal case, the noise power in each frequency bin is estimated as the minimum of all frames in the last 10 seconds at the respective frequency.

$$M(\omega) = \min\big(Frame_i(k)\big),\ where\ k\ is\ the\ frequency\ bin\ number\ and\ i = 1\ to\ 1250$$

Our four frames in the noise minimum buffer each get updated for 2.5s, every 10s. However, since our four frames are each 2.5s long and frame processing can only happen on full frames, the start and end times of the 'last 10 seconds' are actually at multiples of 2.5s, leading to a small error when calculating the minimum over that period.

The following pseudocode describes our implementation for storing the four estimates of the minimum spectrum in each frame.

```
For i = 1 to 4 every 10 seconds
       For k = 0 to length of frame(i) - 1
              If noise min buffer(k) > incoming frame(k)
                     noise min buffer(k) = incoming frame(k)
              Else NULL;
```

Finally, the estimated average noise computed by scaling the minimum magnitude by a noise estimation factor, α. Hence,

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

B. *Subtracting the Noise Subtrum*

After estimating the noise spectrum, the input signal's magnitude can be reduced according to the estimated noise. The output signal is calculated by the following:

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

The definition of $G(k)$ comes from the fact that a negative $G(k)$ would introduce undesired phase change. Hence, a minimum value is defined using λ – we used a typical value of 0.01. Additionally, since only the magnitude of the noise is maintained and not the phase, $G(k)$ is a frequency-dependent gain block, also known as zero-phase filtering.

IV. Enhancements to the Basic Algorithm

The following are a list of enhancements that were tested against the basic algorithm, given in [2]. It was unrealistic to assume that all enhancements were beneficiary to our algorithm, and therefore each one of them were coded and tested separately, using the provided test audio files. The main determining factors in whether an enhancements was decided to be implemented as the amount of background and musical noise present after the subtraction algorithm was applied, as well as the clarity of the speech. We decided that the clarity of speech held precedence over the amount of noise.

Apart from separate enhancements being tested, different combinations of them were also tested.

A. *Enhancement 1: Low Pass Filtering of Input Spectrum*

1) *Theorectical Improvement*

Since an unsmoothed magnitude spectrum has limited accuracy due to short frame length, we used a first order recursive low pass filter to avoid under and over subtraction on the speech signal during a sudden change in SNR by reducing the change in magnitude of the magnitude spectrum valleys. This enhancement should reduce crackling noises due to discontinuities in the noise estimate from an over subtraction.

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

$P_t(\omega)$ and $P_{t-1}(\omega)$ is the current and last filtered input sample respectively, and $k = e^{-T/\tau}$, where T and τ are TFRAME and the time constant (20ms to 80ms) respectively.

2) *Implementation*

```
for (k = 0; k < FFTLEN; k++) {
        /* calculate absolute of fft bin of signal as phase unknown */
        frame_mag[k] = cabs(inframe[k]);
        if (in_filt == 1) {
                //P_t(\omega)=(1-k) x abs(X(\omega))+k x P_t-1(\omega)
                //Current frame * times + last frame due to decimation
                P[k] = (1 - K) * frame_mag[k] + K * last_P[k];
                last_P[k] = P[k];
        }
}
//downcounter to keep track of which frame for processing
//execute at margins between adjacent frames
if (--frame_count == -1) {
        frame_count = min_frame_rate;
        //handle wraparound of min_buffer_index so 0->3
        if (++min_buffer_index > OVERSAMP - 1)min_buffer_index = 0;
        for (k = 0; k < FFTLEN; k++) {
                if (in_filt == 1) {
                        //noise estimated using LPF'd input spectrum
```

```
                        frame_min_buffer[min_buffer_index * FFTLEN + k] = P[k];
                }
        }
}
```

3) Experimental Result

a) Testing for α

The value of α was tested from varying it at runtime. From the default value of 20, a large alpha reduced a lot of noise, but also reduced the amplitude of the speech as well. Since clarity of speech is very important, the value of alpha was slowly decreased until there was a suitable compromise between clarity of speech and the level of noise present. Test results showed that an alpha in the range 5-10 had good results. We settled on a value of 8 as it provided the best results across all the test files.

b) Testing for τ

The value of τ was also varied from its default value of 20 ms. Changing tau directly affects K, as `K = exp(-TFRAME / tau);` The value of tau was varied between 20ms and 80ms, but there was no obvious change in speech clarity nor noise level, apart from some slight distortion at a high value of tau. This can be attributed to the change in the value of K, where it is used in the low pass filter. A high tau means a larger K, and hence a larger weighting given the past filtered input sample.

c) Overall Effect

The overall effect of enhancement 1 was beneficiary as the distortions introduced from the subtraction algorithm were reduced. The speech remained clear, and therefore enhancement 1 was used.

B. Enhancement 2: Low Pass Filtering of Input Spectrum in the Power Domain

1) Theorectical Improvement

This improvement has the same theoretical improvement as enhancement 1. However, the input signal is now passed through a low pass filter in the power domain with the following expression:

$$P_t(\omega) = (1 - K) \times |X(\omega)|^2 + K \times P_{t-1}^2(\omega), \ \ where\ K = e^{-T/\tau}$$

$P_t(\omega)$ and $P_{t-1}(\omega)$ is the current and last filtered input FFT magnitude spectrum respectively, and $k = e^{-T/\tau}$, where T and τ are TFRAME and the time constant (20ms in enhancement 1) respectively.

2) Implementation

```
for (k = 0; k < FFTLEN; k++) {
        /* calculate absolute of fft bin of signal as phase unknown */
        frame_mag[k] = cabs(inframe[k]);
        /**
        * enhancement 2: in_filt 2
        * low pass filter |X(\omega)|^2, then sqrt
        * Thereotically closest to paper approach as the estimator is estimating noise
power and not noise magnitude.
        */
        if (in_filt == 2) {
                P[k] = (1 - K) * (frame_mag[k] * frame_mag[k]) + K * last_P[k];
                last_P[k] = P[k];
        }
}
//downcounter to keep track of which frame for processing
//execute at margins between adjacent frames
if (--frame_count == -1) {
        frame_count = min_frame_rate;
        //handle wraparound of min_buffer_index so 0->3
        if (++min_buffer_index > OVERSAMP - 1)min_buffer_index = 0;
        for (k = 0; k < FFTLEN; k++) {
                if (in_filt == 2) {
                        frame_min_buffer[min_buffer_index * FFTLEN + k] = sqrt(P[k]);
```

```
                }
        }
}
```

3) Experimental Result

As enhancement 2 does the same thing as enhancement 1, the overall effect was good. Therefore, this enhancement was used, but not in conjunction with enhancement 1. Enhancement 2 was used when the power domain was needed (refer to enhancement 5).

C. Enhancement 3: Low Pass Filtering of Noise Estimate

1) Theorectical Improvement

Feeding the noise estimate through a low pass filter should remove abrupt discontinuities in the spectrum when the minimum buffer is rotated.

2) Implementation

Aside from the magnitude domain, this enhancement can also be implemented in the power domain. However, this was not done due to there being increased background noise, as well as slight distortion added.

```
for (k = 0; k < FFTLEN; k++) {
        /* Enhancement 3: LPF the noise estimate
         * PP is the current noise buffer, last PP is the last noise buffer, N is the
noise estimate buffer
         */
        if (out_filt == 1) {
                PP[k] = (1 - K) * N[k] + K * last_PP[k];
                last_PP[k] = PP[k];
                N[k] = PP[k];
        }
}
```

3) Experimental Result

Enhancement 3 on its own did not give much change. However, testing it with either enhancement 1 or 2 gave a healthy improvement in the level of background noise. Hence, enhancement 3 was used.

D. Enhancement 4: Subtraction Gain Modification in the Magnitude Domain

1) Theorectical Improvement

The previous definition of $G(k)$ was $\max\left\{\lambda, \left(1 - \frac{|N(k)|}{|X(k)|}\right)\right\}$ to provide a minimum allowed value for $G(k)$. A dynamic minimum allowed value can be implemented by multiplying lambda by some factor. A dynamic lower bound of $G(k)$ can reduce the distortion to the original speech spectrum compared to using a constant value [citation].

2) Implementation

```
for (k = 0; k < FFTLEN; k++) {
        N[k] = alpha * min(min(frame_min_buffer[k], frame_min_buffer[FFTLEN + k]),
min(frame_min_buffer[k + FFTLEN * 2], frame_min_buffer[k + FFTLEN * 3]));
        switch (gain_var) {
        //abs and (\omega) are omitted
        //default: max(lambda, 1-N/X)
        case 0: inframe[k] = rmul(max(lambda, (1 - N[k] / frame_mag[k])), inframe[k]);
break;
        //change 1: max(lambda(N/X), 1-N/X)
        case 1: inframe[k] = rmul(max(lambda * N[k] / frame_mag[k], (1 - N[k] /
frame_mag[k])), inframe[k]); break;
        //change 2: max(lambda(P/X), 1-N/X)
        case 2: inframe[k] = rmul(max(lambda * P[k] / frame_mag[k], (1 - N[k] /
frame_mag[k])), inframe[k]); break;
        //change 3: max(lambda(N/P), 1-N/P)
```

```
        case 3: inframe[k] = rmul(max(lambda * N[k] / P[k], (1 - N[k] / P[k])),
inframe[k]); break;
        //change 4: max(lambda, 1-N/P)
        case 4: inframe[k] = rmul(max(lambda, (1 - N[k] / P[k])), inframe[k]); break;
        //default is case 0
        default: inframe[k] = rmul(max(lambda, (1 - N[k] / frame_mag[k])), inframe[k]);
break;
        }
}
```

3) Experimental Result

TABLE I.

Case	Expected Change	Actual Effect
0	$G(\omega) = \max\left(\lambda, 1 - \frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}\right)$ No change as default.	No change as default.
1	$G(\omega) = \max\left(\lambda \frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}, 1 - \frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}\right)$ Worse than default, as the noise estimate is in the numerator, using the non-filtered input spectrum. $X(\omega) = \lvert X(\omega) e^{\angle(X(\omega)j}\rvert$ $X(\omega) \cdot \frac{\lvert N(\omega)\rvert}{X(\omega)} \cdot \lambda = \lambda \lvert N(\omega)\rvert e^{\angle(X(\omega)j}$	More musical noise, some crackling, constant background noise. Speech is clear.
2	$G(\omega) = \max\left(\lambda \frac{\lvert P(\omega)\rvert}{\lvert X(\omega)\rvert}, 1 - \frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}\right)$ Better than default, as the lower bound of the subtraction gain will be relevant to current frame power.	Less musical noise than default, constant background noise, speech clear.
3	$G(\omega) = \max\left(\lambda \frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}, 1 - \frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}\right)$ Worse than default, as the noise estimate is in the numerator (see 1, above). However, the noise estimate is using the low pass filtered input, so it should be better than case 1.	More musical noise, but less than case 1, some crackling, constant background noise. Speech is attenuated slightly but clear. **Worst.**
4	$G(\omega) = \max\left(\lambda, 1 - \frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}\right)$ Better than default, as the noise estimate is based on the filtered input spectrum – there should be less discontinuities.	Slight speech attenuation but clear. Both background and musical noise reduced. Negligible crackling. **Best.**

a) Testing for λ

Testing for the value of lambda was similar to tau in that there wasn't any obvious changes when keeping lambda around the default value. A large lambda (>0.2) does not reduce the background noise in any way, and has no effect on the clarity of the speech. Hence, lambda was set to the default value of 0.01.

b) Overall Effect

After testing enhancement 4's permutation of combinations with enhancement 1 and 3, the best version of enhancement 4 was case 4 as predicted, where the very slight speech attenuation was justified by an obvious improvement in the amount of both musical and background noise. Hence, enhancement 4.4 was used.

E. Enhancement 5: Subtraction Gain Modificationin the Power Domain

1) Theorectical Improvement

The improvements should be in line with the expected ones from enhancement 4. In the power domain, the spectrum subtraction is defined by $|X(\omega)|^2 = |Y(\omega)|^2 - \alpha|N(\omega)|^2$ [3].

2) Implementation

The calculation of $G(\omega)$ is now done in the power domain, so the square and square roots are used like the change from enhancement 1 to 2. The same implementation was used as enhancement 4, and so is omitted.

3) Experimental Result

TABLE II.

Case	Expression and Expected Change	Actual Effect
0	$G(\omega) = \max\left(\lambda, \sqrt{1 - \frac{\lvert N(\omega)\rvert^2}{\lvert X(\omega)\rvert^2}}\right)$ No change as default.	No change as default. More musical noise than enhancement 4.
1	$G(\omega) = \max\left(\lambda \frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}, \sqrt{1 - \frac{\lvert N(\omega)\rvert^2}{\lvert X(\omega)\rvert^2}}\right)$ Worse than default, as the noise estimate is in the numerator, using the non-filtered input spectrum. $X(\omega) = \lvert X(\omega) e^{\angle(X(\omega)j}\rvert$ $X(\omega) \cdot \frac{\lvert N(\omega)\rvert}{X(\omega)} \cdot \lambda = \lambda \lvert N(\omega)\rvert e^{\angle(X(\omega)j}$	More musical noise, some crackling, constant background noise. Speech is clear.
2	$G(\omega) = \max\left(\lambda \frac{\lvert P(\omega)\rvert}{\lvert X(\omega)\rvert}, \sqrt{1 - \frac{\lvert N(\omega)\rvert^2}{\lvert X(\omega)\rvert^2}}\right)$ Better than default, as the lower bound of the subtraction gain will be relevant to current frame power.	Less musical noise than default, no background noise, speech clear. **Best, but worse than enhancement 4.4.**
3	$G(\omega) = \max\left(\lambda \frac{\lvert N(\omega)\rvert}{\lvert P(\omega)\rvert}, \sqrt{1 - \frac{\lvert N(\omega)\rvert^2}{\lvert P(\omega)\rvert^2}}\right)$ Worse than default, as the noise estimate is in the numerator (see above). However, the noise estimate is using the low pass filtered input, so it should be better than case 1.	Huge amounts of distortion. Speech attenuated. **Worst, worse than enhancement 4.3.**
4	$G(\omega) = \max\left(\lambda, \sqrt{1 - \frac{\lvert N(\omega)\rvert^2}{\lvert P(\omega)\rvert^2}}\right)$ Better than default, as the noise estimate is based on the filtered input spectrum – there should be less discontinuities.	Negligible noise (almost completely gone) but speech is extremely muffled. Track is clean of noise but words cannot be clearly distinguished.

Lambda was kept at 0.01. Using the methods of enhancement 4, permutations of enhancement 5 were tested with enhancements 2 and 3. However, the best version of enhancement 5 lost out to enhancement 4.4, and therefore enhancement 5 was not used.

F. *Enhancement 6: Variable alpha for Oversubtraction of the Noise Spectrum*

1) Theorectical Improvement

Since the noise estimate is multiplied by a noise estimation factor, α, as alpha increases, the estimated noise increases. However, if alpha is set too high, the speech signal will become attenuated as well. Therefore, a variable alpha is needed to dynamically change the noise estimate for periods of noise and periods of speech. The method of discriminating between noise and speech was to calculate the SNR. The lower frequencies often correspond to noise and hence a high alpha is preferred for a low SNR value. For a high SNR, a low alpha is desired. However, only have two alpha values would be in the form of a step function – distortion would be introduced. Hence, two thresholds were chosen, with a linear line connecting the high and low alphas. The lower/upper thresholds for the SNR were taken from [4], with the lower α changed to 2 from testing.

2) Implementation

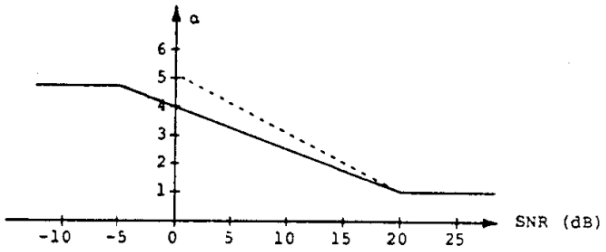


Figure 1: Alpha and SNR Graph, showing two thresholds [4]

Initially, `log10f` from `math.h` was used to compute the SNR. However, calculating logarithms at run-time pushed our `cpufrac` to over 1 – we were doing non real time digital signal processing. A lookup table method was much preferred as the as it kills two birds with one stone – it bypasses the need for the logarithm calculation, and it also bypasses the multiplication and addition needed to calculate the value of alpha in between the two thresholds, where the alpha lies on a line of form $y = mx + c$. As such, a function `ilog10c` was created to return the value of alpha with an input of a floating point SNR. `inline` is specified to the compiler for possible optimizations regarding replacing the function call with the actual function code.

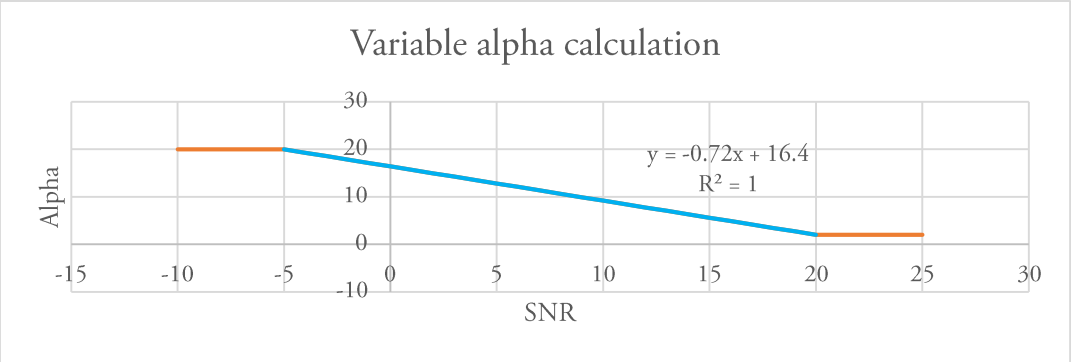


Figure 2: Excel plot showing graph of values returned by custom ilog10c function

```
/**
 * lookup table created using excel, returns alpha value from snr
 * @param  ratio [float snr]
 * @return       [int alpha]
 */
inline int ilog10c(float ratio) {
        //SNR < -5
        if (ratio <    0.316227766    ) return              20     ;
        //SNR between -5 and 20
        else if (ratio <      0.398107171    ) return       19.28   ;
        else if (...) return ...;
        //SNR is 20+
        else return 2;
}
```

3) Experimental Result

This enhancement did not prove to be very successful. In theory, it should be a sizable improvement as modifying the alpha value is the most direct way of manipulating the noise estimate. However, after implementing enhancement 6, the speech was severely attenuated. Stepping through the code revealed that the alpha values were always on either side of the two thresholds, and never in the mid-range. Turning back to the more obvious step change in alpha with one threshold, enhancement 6 did not provide any obvious improvements. Due to time constraints while testing, this enhancement was not applied as the current combination of enhancements 1,3,4.4 provide a great clarity in speech already with noise levels close to none.

G. *Enhancement 7: Modifying the Frame Length*

1) *Theorectical Improvement*

A longer frame length gives a higher resolution for our noise estimate, but comes at a cost of increased computation time. A shorter frame length gives decreased computation time (directly affects the `for` loop size), but may give an inaccurate noise and speech estimation; the noise subtraction will remove less noise and distort the speech.

2) *Implementation*

```
#define FFTLEN ___                    /* fft length = frame length 256/8000 = 32 ms*/
```

3) *Experimental Result*

A frame length of 128 decreased our `cpufrac` significantly, but led to distortion in the output. A frame length of 512 led to `cpufrac` rising above 1. Since the added distortion is non negligible, the default of 256 is retained – enhancement 7 was not used.

H. *Enhancement 8: Apply Residual Noise Reduction to Reduce Musical Noise*

1) *Theorectical Improvement*

From testing our previous enhancements, there were still musical noise present. Musical noise are isolated peaks left in a spectrum after processing, such as our subtraction algorithm, is applied [5]. This musical noise can be reduced by applying 'residual noise reduction' [6]. Theoretically, it should reduce the amount of musical noise by setting a noise threshold.

2) *Implementation*

From [6], if the inverse of the square root of the posterior SNR, i.e. $\frac{|N(\omega)|}{|X(\omega)|}$, exceeds a predefined threshold, then $Y(\omega)$ is replaced by the minimum of the three adjacent frames. This procedure induces a one frame delay [6], and since new buffers are used to implement this comparison.

```
//////////////////////////////////////////
//enhancement 8: residual noise reduction //
//////////////////////////////////////////
//if noise over some threshold, get minimum from adjacent frames instead
if (RNR_flag == 1) {
	if ((last_N[k] / last_frame_mag[k]) > RNR_threhold) {
		//over threshold, get min from adjacent
		RNRbuffer[k] = complex_min(inframe[k], complex_min(last_frame[k],
last_last_frame[k]));
	}
	else {
		//else get last inframe is the future frame
		RNRbuffer[k] = last_frame[k];
		switch (gain_var) {
		case 0: (...)
		}
	}
	//shuffle buffers
	last_frame_mag[k] = frame_mag[k];
	last_last_frame[k] = last_frame[k];
	last_frame[k] = inframe[k];
	last_N[k] = N[k];
}
```

3) *Experimental Result*

This enhancement again brought in distortion and crackling. The speech was unchanged. Hence, this enhancement was not used.

I. *Enhancement 9: Take the Minimum Spectra over a Shorter Period*

1) *Theorectical Improvement*

By looking at the waveform of `clean.wav`, we found that the first 3 seconds have no speech. Therefore, they will be purely noise when the noise is added. The default number seconds before our noise subtraction kicks in is 7.5 seconds, as we have to fill the four buffers, each of length 312 frames.

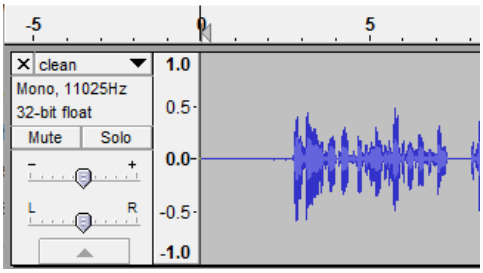


Figure 3: Audacity waveform of clean.wav, shows silence for first few seconds

2) *Implementation*

Ideally, the four buffers should be filled before the speech starts. Hence, the buffers were changed to have 312÷4 = 78 frames. The buffers will be filled before the speech starts, i.e. in 7.5÷4 = 1.9 seconds.

```
int  min_frame_rate = 78;
```

3) *Experimental Result*

The noise is now reduced before the speech starts and hence the first few words will have the noise reduction applied. This also makes our system respond more quickly to a change in the noise level. There were no obvious increases of distortion or musical noise. This enhancement was used with a value of 78.

J. *Enhancement 10: Estimating the non-stationary Noise*

1) *Theorectical Improvement*

The 9 enhancements above have an underlying assumption that the noise is stationary, that is, there is a constant mean and variance. Of course, most artificial noise sources actually have a time-variant mean and variance [7], and hence, a better method of estimating this non-stationary noise was implemented.

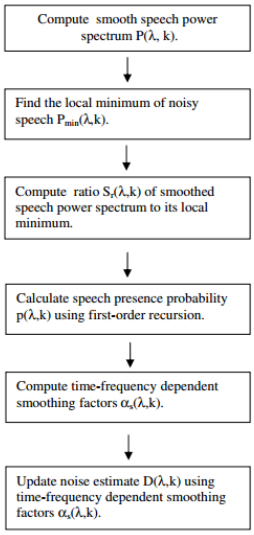


Figure 4: Block diagram for enhancement 10, found in [8]

2) *Implementation*

The implementation is based on Figure 4. All equations in the implementation are from [8]. First, the input power spectrum was smoothed using first order recursion - similar to the LPF in enhancement 2.

$$P(\lambda,k) = \eta P(\lambda-1,k) + (1-\eta)|Y(\lambda,k)|^2$$

Then, the local minimum power is updated using a non-linear model, improving the performance of the buffer when the noise is of high variance.

$$If\ P_{min}(\lambda-1,k) < P(\lambda,k)$$
$$P_{min}(\lambda,k) = \gamma P_{min}(\lambda-1,k) + \frac{1-\gamma}{1-\beta}[P(\lambda,k) - \beta P(\lambda-1,k)]$$
$$Else\ P_{min}(\lambda,k) = P(\lambda,k)$$

```
//implement non-linear model, improve performance when high var noise
for (k = 0; k < FFTLEN; k++) {
	if (P[k] < frame_min_buffer[min_buffer_index * FFTLEN + k])
		frame_min_buffer[min_buffer_index * FFTLEN + k] = P[k];
	else {
		frame_min_buffer[min_buffer_index * FFTLEN + k] = gamma *
frame_min_buffer[min_buffer_index * FFTLEN + k] + ((1 - gamma) / (1 - beta)) * (P[k] -
beta * last_P[k]);
	}
}
```

Next, a factor based on the probability of there being speech is used to estimate the noise power. This probability calculation is based on a comparison between the ratio of the smoothed input power spectrum and the minimum buffer power spectrum, to a frequency dependent threshold value. Hence,

$$If\ S_r(\lambda,k) > \delta(k), speech\ is\ present, hence\ I(\lambda,k) = 1$$
$$Else, speech\ is\ not\ present, I(\lambda,k) = 0$$
$$I(\lambda,k)\ replaced\ by\ p(\lambda,k) = \alpha_p p(\lambda-1,k) + (1-\alpha_p)I(\lambda,k)$$

```
//use speech probability constant to check for presence of voice
for (k = 0; k < FFTLEN; k++) {
	local_min[k] = min(min(frame_min_buffer[k], frame_min_buffer[FFTLEN + k]),
min(frame_min_buffer[k + FFTLEN * 2], frame_min_buffer[k + FFTLEN * 3]));
		if ((P[k] / local_min[k]) > speech_thres[k]) {
			speech_prob[k] = prob_const * speech_prob[k] + 1 - prob_const;
		}
		else {
			speech_prob[k] = prob_const * speech_prob[k];
		}
}
```

To combat the problem of the noise being of a high variance, the probability adapts a smoothing scheme, i.e., the updating of the estimated noise is based on three factors: the estimate in the last frame, the frequency dependent smoothing factor, and the current input power. α_d is represented by `sub_const`.

$$D(\lambda,k) = \alpha_s(\lambda,k)D(\lambda-1,k) + [1-\alpha_s(\lambda,k)]|Y(\lambda,k)|^2$$
$$\alpha_s(\lambda,k) \triangleq= \alpha_d + (1-\alpha_d)p(\lambda,k)$$

Lastly, the spectral subtraction gain is calculated using a Wiener-type speech enhancement algorithm.

$$C(\lambda,k) = \max\{|Y(\lambda,k)|^2 - D(\lambda,k), vD(\lambda,k)\}$$
$$G(\lambda,k) = \frac{C(\lambda,k)}{C(\lambda,k) + \mu_k D(\lambda,k)}$$

Here, μ_k was a custom sub-band dependent quantity. Low frequencies have a lower μ_k so to avoid over subtraction in the mid-band of the speech; high frequencies have a much higher (close to 6 to 7 times) higher μ_k to supress the noise components.

The idea of having a variable μ_k originates from the spectrogram of `clean.wav` (shown in Figure 5), which show that the majority of the speech signal components lie in the low frequencies and mid-bands.

```
//probablity smoothing and spectral subtraction calculation
```

```
for (k = 0; k < FFTLEN; k++) {
        subfactor[k] = min(max(sub_const + (1 - sub_const) * speech_prob[k], prob_const),
1);
        N[k] = subfactor[k] * N[k] + (1 - subfactor[k]) * frame_mag[k] * frame_mag[k];
        //enhancement 3
        PP[k] = (1 - K) * N[k] + K * last_PP[k];
        last_PP[k] = PP[k];
        N[k] = PP[k];
        //wiener filter implementation, mu_k is custom variable
        C = max(frame_mag[k] * frame_mag[k] - N[k], v * N[k]);
        G[k] = C / ((C + mu_k[k]) * N[k]);
        out_buff[k] = rmul(min(G[k], 1), inframe[k]);
        last_P[k] = P[k];
}
```

3) Experimental Result

This enhancement brought in mixed results. In some test tracks, such as the phantom set, it would completely remove all noise, with the speech left more or less intact. However, there would be distortion and lots of musical noise in other tracks. For the most difficult test track, phantom4, it removed almost all the noise but left the speech severely attenuated, and it was decided that the very effective noise removal in phantom4 was not justified for the amount of speech is attenuated. Our previous enhancements did not remove as much noise but also did not attenuate the speech as severely. Since clarity of speech is the priority, enhancement 10 was not used. Moreover, since enhancement 10 is standalone, it cannot be added to the previous 9 enhancements. Enhancement 10 also had slight issues with computation time (solved by using a shorter FFT length, but this is non-ideal, see enhancement 7).

V. Spectrogram Analysis of Best Combination of Enhancements

The previous experimental results were all done by ear, and since every person's view on the clarity of our algorithm is subjective, the output audio was recorded using Audacity, then MATLAB's `audioread` and `spectrogram` functions were used to plot the spectrogram of both the unenhanced and enhanced files, for all the provided test files. A reference spectrogram of `clean.wav` is provided in Figure 5.

There was a different set of best parameters and enhancements for each test track from our extensive testing, but there was one combination of enhancements which provided the best mix of background noise reduction, musical noise reduction, and speech clarity. That combination is shown in Table III.

TABLE III.

Enhancement	1	2	3	4	5	6	7	8	9	10
Used?	Y	N	Y	Y (v4)	N	N	N	N	Y	N
$\alpha = 8; \lambda = 0.01; \tau = 0.02s$										

For each plotted graph, the left plot is the original file; the right is the enhanced file. The legend is also chosen so that red corresponds to the highest amplitude, and blue the lowest.

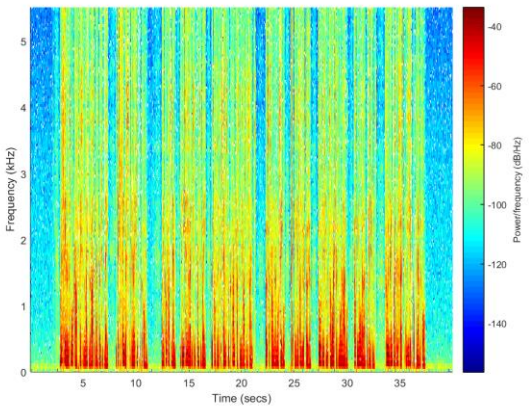


Figure 5: Spectrogram of clean.wav

From Figures 6 to 25, we can clearly see the difference the spectral subtraction algorithm makes. The high frequency noise is clearly reduced in all the test files. However, there is a variable amount of suppressed speech is the test tracks (red). The lowest amount of speech suppression occurred in *factory1* and *phantom1*, but tracks like *car1* and *phantom4* had attenuated speech (less red).

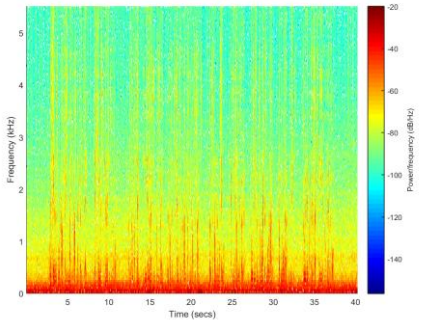
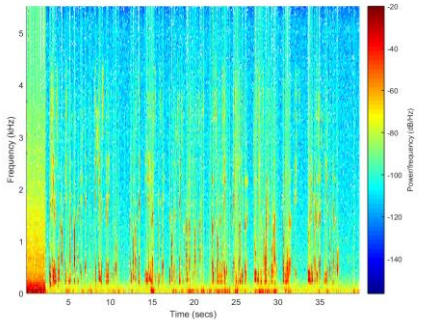
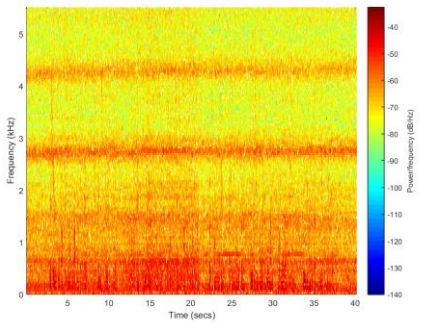
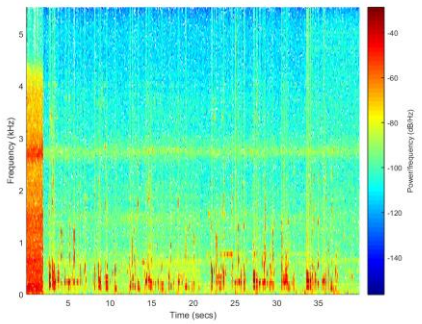
The spectrograms of the most challenging two tracks, car1 and phantom4, are placed in Table IV (Figures 6-9). The rest are placed in Table V (Figures 10-31), in the appendix.

In the case of the best tracks like *factory1* and *phantom1*, the noise has a large variance – after subtraction, the end spectrogram is closer to the spectrogram of clean than others due to the speech being in the low frequencies.

In the case of suppressed speech, i.e., *car1*, the noise is mainly distributed around low frequencies. The noise spectral subtraction will be done mainly on the low frequencies as there is a decreased ability to differentiate between noise and speech. However, the speech is still clear albeit slightly attenuated.

In the case of *phantom4*, the noise power is very large in comparison to the speech, hence there is a low SNR. The speech magnitude is almost the same as the noise magnitude. Hence, the minimum statistic subtraction algorithm will also remove the speech components, leading to suppressed speech.

TABLE IV.

File	Original Spectrogram	After Processing
car1	 *Figure 6: car1 before*	 *Figure 7: car1 after*
phantom4	 *Figure 8: phantom4 before*	 *Figure 9: phantom4 after*

VI. Conclusion

The ideal set of enhancements and parameters came after extensive testing. α could not be too high to avoid attenuating the speech, but could not be too low to subtract away too little noise. λ had to chosen to be a low value to avoid too much background noise, and τ had to be chosen to avoid distortion. The frame length and frame rate had to be chosen to be lower than the limit of `cpufrac` to avoid non real time processing.

In conclusion, although some compromises had to be made, our combination of enhancements and parameters achieved a high amount of noise reduction without affecting the clarity of speech as shown by the spectrogram analysis. This project moreover gave us insight into frame processing and the spectral subtraction algorithm, and as shown in enhancement 10, further improvements can be implemented with more time.

VII. References

[1] dsprelated.com, "Overlap-add Decomposition," [Online]. Available: https://www.dsprelated.com/freebooks/sasp/Overlap_Add_Decomposition.html. [Accessed 25 3 2016].

[2] P. D. Mitcheson, "Project: Speech Enhancement," 2 1 2013. [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-591046-dt-content-rid-2714546_1/courses/DSS-EE3_19-15_16/project_description.pdf. [Accessed 24 3 2016].

[3] S. V. Vaseghi, "Spectral Subtraction," 2000. [Online]. Available: http://dsp-book.narod.ru/304.pdf. [Accessed 16 3 2016].

[4] M. Berouti, R. Schwartz and J. Makhoul, "Enhancement of speech corrupted by acoustic noise," in *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '79., Volume: 4*, Cambridge, MA.

[5] Vocal, "Musical Noise in Acoustic Noise Reduction," Vocal, [Online]. Available: http://www.vocal.com/noise-reduction/musical-noise/. [Accessed 24 3 2016].

[6] S. F. Boll, "Suppression of acoustic noise in speech using spectral subtraction," *IEEE TRANSACTIONS ON ACOUSTICS SPEECH AND SIGNAL PROCESSING,* pp. 113-120, 5 1979.

[7] L. Galleani and L. Cohen, "Nonstationary and stationary noise," in *Automatic Target Recognition XVI. Edited by Sadjadi, Firooz A.. Proceedings of the SPIE, Volume 6234, id. 623416*, New York, USA, 2006.

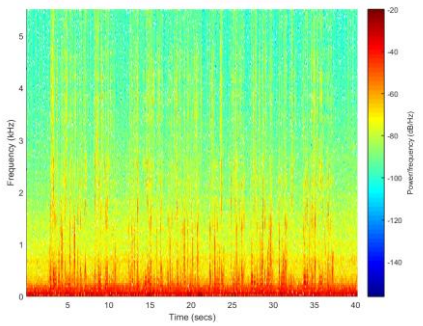
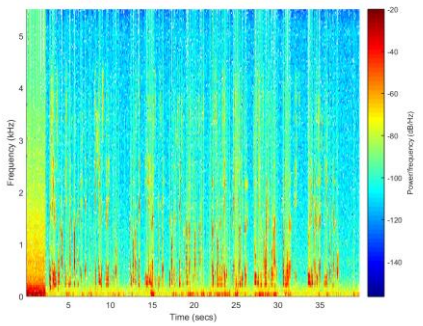
[8] S. Rangachari and P. C. Loizou, "A noise-estimation algorithm for highly non-stationary environments," 17 12 2004. [Online]. Available: http://ecs.utdallas.edu/loizou/speech/noise_estim_article_feb2006.pdf. [Accessed 24 3 2016].

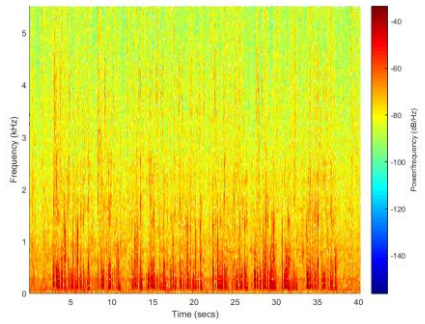
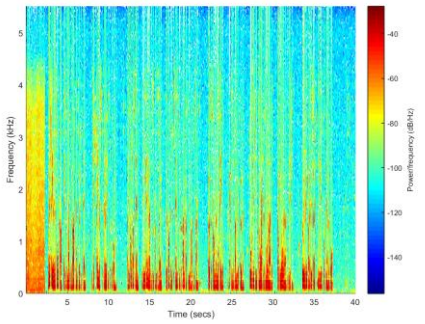
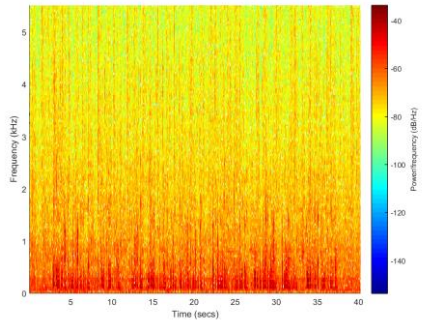
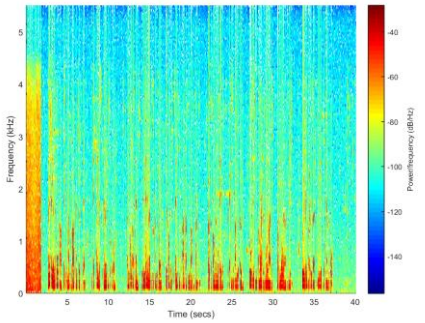
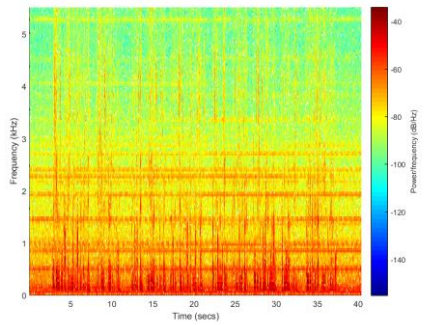
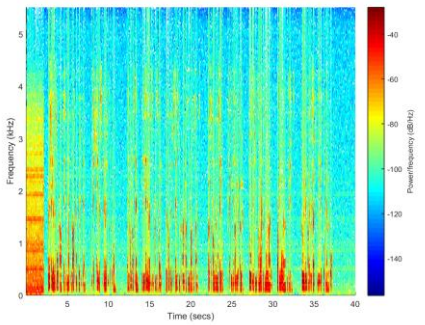
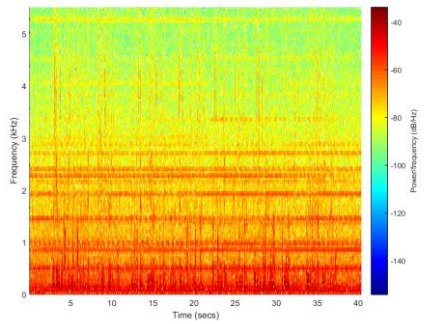
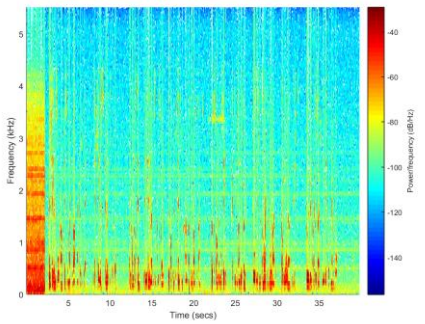
[9] R. Martin, "Spectral Subtraction Based on Minimum Statistics," [Online]. Available: http://www.ruhr-uni-bochum.de/ika/forschung/forschungsbereich_martin/speech_audio_processing/eusipco1994_martin.pdf. [Accessed 22 3 2016].

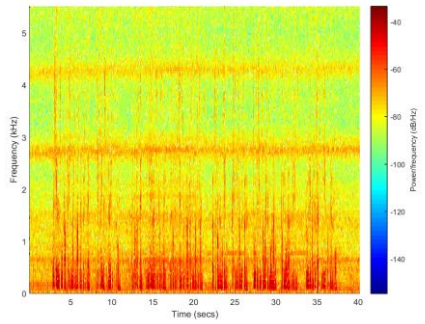
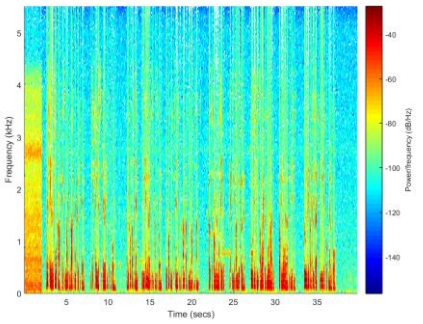
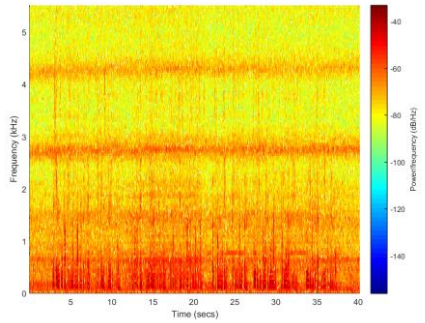
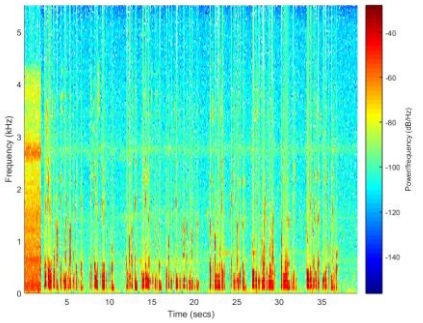
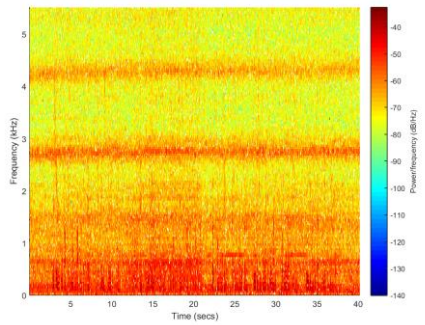
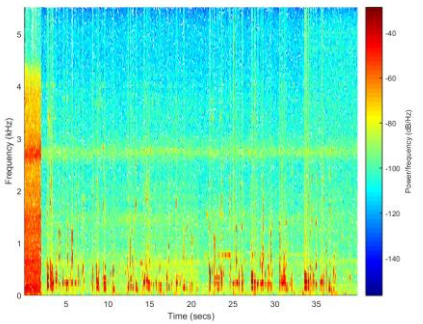
VIII. Appendix

A. *Spectograms of all test tracks using enhancements*

TABLE V.

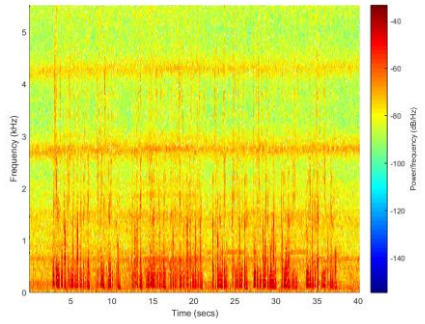
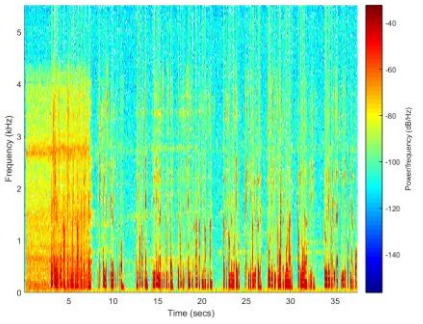
File	Original Spectrogram	After Processing
car1	 *Figure 10: car1 before*	 *Figure 11: car1 after*

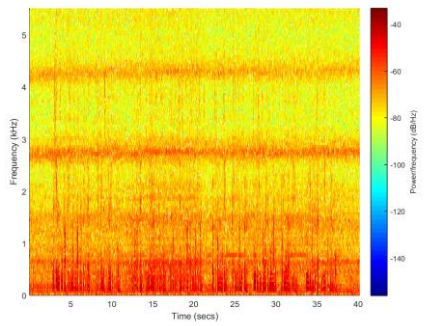
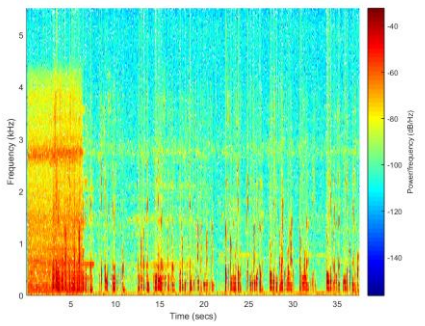
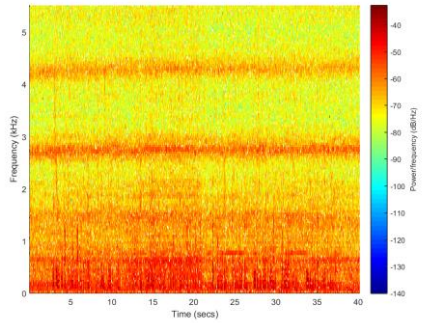
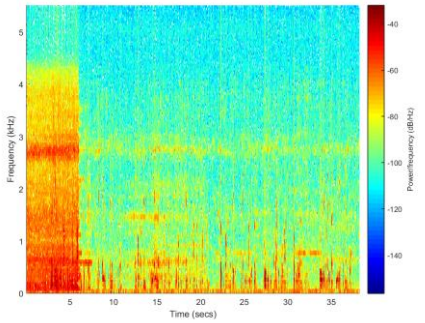
factory1		
	Figure 12: factory1 before	*Figure 13: factory1 after*
factory2		
	Figure 14: factory 2 before	*Figure 15: factory2 after*
lynx1		
	Figure 16: lynx1 before	*Figure 17: lynx1 after*
lynx2		
	Figure 18: lynx2 before	*Figure 19: lynx2 after*

phantom1		
	Figure 20: phantom1 before	*Figure 21: phantom1 after*
phantom2		
	Figure 22: phantom2 before	*Figure 23: phantom2 after*
phantom4		
	Figure 24: phantom4 before	*Figure 25: phantom4 after*

B. *Spectrograms of phantom test set using enhancement 10*

TABLE VI.

File	Original Spectrogram	After Processing
phantom1		
	Figure 26: Enhancement 10 phantom1 before	*Figure 27: Enhancement 10 phantom1 after*

phantom2	 *Figure 28: Enhancement 10 phantom2 before*	 *Figure 29: Enhancement 10 phantom2 after*
phantom4	 *Figure 30: Enhancement 10 phantom4 before*	 *Figure 31: Enhancement 10 phantom4 after*

C. *Table of Figures*

D. Enhancement 10 Function

```
/**
 * custom mu_k threshold initializtion
 */
void init_thres(void) {
	int k = 0;
	for (k = 0; k < FFTLEN; k++) {
		if (k <= (int)(1000 / FSAMP)*FFTLEN) {
			speech_thres[k] = 2;
			mu_k[k] = 5;
		}
		else {

			if ((k <= (int)(3000 / FSAMP)*FFTLEN) && (k > (int)(1000 /
FSAMP)*FFTLEN)) {
				speech_thres[k] = 2;
				mu_k[k] = 30;
			}
			else {
				speech_thres[k] = 5;
				mu_k[k] = 60;
			}
		}
	}
}
void ProcVswitch_estimator(void) {
	int k;

	fft(FFTLEN, inframe);
	////////////////////////////////////////////////////////////
	//previous structure from basic algorithm, enhancement 2 //
	////////////////////////////////////////////////////////////
	for (k = 0; k < FFTLEN; k++) {
		frame_mag[k] = cabs(inframe[k]);
		P[k] = (1 - K) * (frame_mag[k] * frame_mag[k]) + K * last_P[k];
	}

	if (--frame_count == -1) {
		frame_count = min_frame_rate;
		if (++min_buffer_index > OVERSAMP - 1)min_buffer_index = 0;
		for (k = 0; k < FFTLEN; k++) {
			frame_min_buffer[min_buffer_index * FFTLEN + k] = P[k];
		}
	}
	else {
		//implement non-linear model, improve performance when high var noise
		for (k = 0; k < FFTLEN; k++) {
			if (P[k] < frame_min_buffer[min_buffer_index * FFTLEN + k])
				frame_min_buffer[min_buffer_index * FFTLEN + k] = P[k];
			else {
				frame_min_buffer[min_buffer_index * FFTLEN + k] = gamma *
frame_min_buffer[min_buffer_index * FFTLEN + k] + ((1 - gamma) / (1 - beta)) * (P[k] -
beta * last_P[k]);
			}
		}
	}
	//use speech probability constant to check for presence of voice
	for (k = 0; k < FFTLEN; k++) {
		local_min[k] = min(min(frame_min_buffer[k], frame_min_buffer[FFTLEN +
k]), min(frame_min_buffer[k + FFTLEN * 2], frame_min_buffer[k + FFTLEN * 3]));
```

```
                if ((P[k] / local_min[k]) > speech_thres[k]) {
                        speech_prob[k] = prob_const * speech_prob[k] + 1 - prob_const;
                }
                else {
                        speech_prob[k] = prob_const * speech_prob[k];
                }
        }
        //probablity smoothing and spectral subtraction calculation
        for (k = 0; k < FFTLEN; k++) {
                subfactor[k] = min(max(sub_const + (1 - sub_const) * speech_prob[k],
prob_const), 1);
                N[k] = subfactor[k] * N[k] + (1 - subfactor[k]) * frame_mag[k] *
frame_mag[k];

                //enhancement 3
                PP[k] = (1 - K) * N[k] + K * last_PP[k];
                last_PP[k] = PP[k];
                N[k] = PP[k];

                //wiener filter implementation, mu_k is custom variable
                C = max(frame_mag[k] * frame_mag[k] - N[k], v * N[k]);
                G[k] = C / ((C + mu_k[k]) * N[k]);
                out_buff[k] = rmul(min(G[k], 1), inframe[k]);
                last_P[k] = P[k];

        }

        ifft(FFTLEN, out_buff);

}
```

E. *Source Code of enhance.c*

```
/*************************************************************************************
                                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 <stdio.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>
#include <float.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 */



//////////////////////swithchable parameter
int alpha = 8;
int alphatmp = 8;
int alphathresholdlow = 5;
int alphathresholdhigh = 20;
int enchance6 = 0;
float SNRtmp = 0.0;
float lambda = 0.01;
float tau = 0.02; //from 0.02 to 0.08
int in_filt = 1;
int out_filt = 1;
int gain_var = 4; //2,3,4 only work with in- filter on
float RNR_threhold = 0.9;
#define RNR_flag 0
//int RNR_flag = 0;
int  min_frame_rate = 78;
float speech_thres = 4;
float prob_const = 0.2;
float sub_const = 0.85;
float delta = 0.0;

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

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \

/*************************************************************************/
                                /*     REGISTER                                FUNCTION
                                SETTINGS               */

/*************************************************************************/\
                                0x0017,       /* 0 LEFTINVOL        Left line input
channel volume  0dB                                 */\
                                0x0017,       /* 1 RIGHTINVOL Right line input channel
volume 0dB                              */\
                                0x01f9,       /* 2 LEFTHPVOL        Left channel
headphone volume       0dB                          */\
```

```
                                0x01f9,       /* 3 RIGHTHPVOL Right channel headphone
volume   0dB                                  */\
                                0x0011,       /* 4 ANAPATH   Analog audio path control
                DAC on, Mic boost 20dB*/\
                                0x0000,       /* 5 DIGPATH   Digital audio path control
                All Filters off         */\
                                0x0000,       /* 6 DPOWERDOWN Power down control
                        All Hardware on         */\
                                0x0043,       /* 7 DIGIF             Digital audio
interface format       16 bit                         */\
                                0x008d,       /* 8 SAMPLERATE Sample rate control
        8 KHZ-ensure matches FSAMP */\
                                0x0001        /* 9 DIGACT    Digital interface
activation      On                                      */\

/************************************************************************/
                              };

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

float *inbuffer, *outbuffer;          /* Input/output circular buffers */
float *outframe;                /* Input and output frames */
complex *inframe;
float *last_P;
float *last_PP;
float *P;
float *PP;
float *inwin, *outwin;                      /* Input and output windows */
float *frame_mag;
float *N;
float *last_N;
float *last_frame_mag;
complex *RNRbuffer;
complex *last_frame;
complex *last_last_frame;
complex* out_buff;
float* speech_prob;
float* local_min;
float* subfactor;
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 frame_count = 0;
volatile int min_buffer_index = 0;
float frame_min_buffer[OVERSAMP * FFTLEN] = {FLT_MAX};
float noise_amp = FLT_MAX;
float K;



/****************************** Function prototypes ******************************/
void init_hardware(void);            /* Initialize codec */
void init_HWI(void);                 /* Initialize hardware interrupts */
void ISR_AIC(void);                         /* Interrupt service routine for codec */
void process_frame(void);            /* Frame processing routine */
void init_buffer(void);
void ProcVswitch(void);
void ProcVswitch_estimator(void);
inline float max(float a, float b);
inline float min(float a, float b);
complex complex_min(complex a, complex b);
inline int ilog10c(float input);
inline float my_sqrt(const float m);

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

```
{

        int k; // used in various for loops

        /*      Initialize and zero fill arrays */

        inbuffer       = (float *) calloc(CIRCBUF, sizeof(float));  /* Input array */
        outbuffer      = (float *) calloc(CIRCBUF, sizeof(float));  /* Output array */
        inframe        = (complex *) calloc(FFTLEN, sizeof(complex));      /* Array for
processing*/
        outframe       = (float *) calloc(FFTLEN, sizeof(float));   /* Array for
processing*/
        inwin          = (float *) calloc(FFTLEN, sizeof(float));   /* Input window */
        outwin         = (float *) calloc(FFTLEN, sizeof(float));   /* Output window */
        frame_mag      = (float *) calloc(FFTLEN, sizeof(float));
        N                       = (float *) calloc(FFTLEN, sizeof(float));
        last_frame     = (complex *) calloc(FFTLEN, sizeof(float));
        last_last_frame = (complex *) calloc(FFTLEN, sizeof(float));
        last_frame_mag = (float *) calloc(FFTLEN, sizeof(float));
        last_N = (float *) calloc(FFTLEN, sizeof(float));
        last_P = (float *) calloc(FFTLEN, sizeof(float));
        P = (float *) calloc(FFTLEN, sizeof(float));
        last_PP = (float *) calloc(FFTLEN, sizeof(float));
        PP      = (float *) calloc(FFTLEN, sizeof(float));
        RNRbuffer = (complex *) calloc(FFTLEN, sizeof(complex));
        out_buff = (complex*)calloc(FFTLEN, sizeof(complex));
        speech_prob = (float*)calloc(FFTLEN, sizeof(float));
        local_min = (float*)calloc(FFTLEN, sizeof(float));
        subfactor = (float*)calloc(FFTLEN, sizeof(float));
        K = exp(-TFRAME / tau);


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

        //puts("test\n");

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


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

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

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

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

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

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


}
/********************************* init_buffer() *******************************/
void init_buffer(void) {
        int i = 0;
        for (i = 0; i < FFTLEN * OVERSAMP; i++) {
                frame_min_buffer[i] = FLT_MAX;
        }
}
/********************************* maxmin() *******************************/
inline float max (float a, float b) {
        return (((a) > (b)) ? (a) : (b));
}
inline float min (float a, float b) {
        return (((a) < (b)) ? (a) : (b));
}
complex complex_min(complex a, complex b) {
        return (((cabs(a)) < (cabs(b))) ? (a) : (b));
}

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

}

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


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

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

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

        /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where
the
        data should be read (inbuffer) and saved (outbuffer) for the purpose of
processing */
        io_ptr0 = frame_ptr * FRAMEINC;

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

        m = io_ptr0;
```

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

        /************************ DO PROCESSING OF FRAME  HERE
*************************/
        ProcVswitch();

        /* please add your code, at the moment the code simply copies the input to the
        ouptut with no processing */


        for (k = 0; k < FFTLEN; k++)
        {       if (RNR_flag == 0) {
                        outframe[k] = inframe[k].r;/* copy input straight into output */
                }
                else {
                        outframe[k] = RNRbuffer[k].r;
                }

        }

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

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

        m = io_ptr0;

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

/********************************* end of process_frame()
*******************************/



/************************** INTERRUPT SERVICE ROUTINE
        ****************************/

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

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

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

        /* update io_ptr and check for buffer wraparound */

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

```
/******************************** all Versions of  proces function
*********************************/
///vswitch is original setting with no filtering,G = max(lambda,1-|N|/|X|),
void ProcVswitch(void) {
	int k;

	////////////////////////////////
	//compute FFT of incoming frame //
	////////////////////////////////
	fft(FFTLEN, inframe);

	for (k = 0; k < FFTLEN; k++) {
		/* calculate absolute of fft bin of signal as phase unknown */
		frame_mag[k] = cabs(inframe[k]);

		//////////////////////////////
		//enhancement 2: in_filt 2 //
		//////////////////////////////
		//low pass filter |X(\omega)|^2, then sqrt
		//Thereotically closest to paper approach as the estimator is estimating
noise power and not noise magnitude.

		if (in_filt == 2) {
			P[k] = (1 - K) * (frame_mag[k] * frame_mag[k]) + K * last_P[k];
			last_P[k] = P[k];
		}
		else {

			//////////////////////////////
			//enhancement 1: in_filt 1 //
			//////////////////////////////
			//P_t(\omega)=(1-k) x abs(X(\omega))+k x P_t-1(\omega)
			//LPF on noise estimator, used to smooth subbands
			//Current frame * times + last frame due to decimation

			if (in_filt == 1) {
				P[k] = (1 - K) * frame_mag[k] + K * last_P[k];
				last_P[k] = P[k];
			}
		}
	}
	//downcounter to keep track of which frame for processing
	//execute at margins between adjacent frames
	if (--frame_count == -1) {
		frame_count = min_frame_rate;
		//handle wraparound of min_buffer_index so 0->3
		if (++min_buffer_index > OVERSAMP - 1)min_buffer_index = 0;
		for (k = 0; k < FFTLEN; k++) {
			///////////////////////////////////
			//enhancement 2 update min buffer //
			///////////////////////////////////
			if (in_filt == 2) {
				frame_min_buffer[min_buffer_index * FFTLEN + k] =
sqrt(P[k]);
			}
			else {
				///////////////////////////////////
				//enhancement 1 update min buffer //
				///////////////////////////////////
				if (in_filt == 1) {
					frame_min_buffer[min_buffer_index * FFTLEN + k] =
P[k];
				}
				else {
					frame_min_buffer[min_buffer_index * FFTLEN + k] =
frame_mag[k];
				}
			}
```

```
                }
        }

        ////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
        //main objective: implement the spectral subtraction technique
//
        //Assuming non-musical noise, so no convolution between speech and noise.
//
        //Therefore, all noise is in the background and statistically independent,
therefore treat as additive noise, so the subtraction technique works. //
        //Peaks in FFT correspond to speech, therefore find lowest valley (local minimum
thereotically be noise).                                        //
        //In C, compare minimum by looping through frequency bins, and find minimum
frequency bins                                                  //
        ////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

        //same updating min buffer
        else {
                for (k = 0; k < FFTLEN; k++) {
                        if (in_filt == 2) {
                                if (sqrt(P[k]) < frame_min_buffer[min_buffer_index *
FFTLEN + k])frame_min_buffer[min_buffer_index * FFTLEN + k] = sqrt(P[k]);
                        }
                        else {
                                if ( in_filt == 1) {
                                        if (P[k] < frame_min_buffer[min_buffer_index *
FFTLEN + k])frame_min_buffer[min_buffer_index * FFTLEN + k] = P[k];
                                }
                                else {
                                        if (frame_mag[k] <
frame_min_buffer[min_buffer_index * FFTLEN + k])frame_min_buffer[min_buffer_index *
FFTLEN + k] = frame_mag[k];
                                }
                        }
                }
        }

        for (k = 0; k < FFTLEN; k++) {
                {
                        ////////////////////////////////////////////
                        //enhancement 6, calc snr, and adjust alpha //
                        ////////////////////////////////////////////
                        if (enchance6) {
                                //calculate SNR
                                SNRtmp = (frame_mag[k] * frame_mag[k]) / (N[k] * N[k]);
                                //find alpha value from SNR using lookup table
                                alphatmp = ilog10c(SNRtmp);
                        }
                }

                //update noise buffer
                N[k] = alphatmp * min(min(frame_min_buffer[k], frame_min_buffer[FFTLEN +
k]), min(frame_min_buffer[k + FFTLEN * 2], frame_min_buffer[k + FFTLEN * 3]));


                ////////////////////
                //enhancement 3: //
                ////////////////////
                //LPF the noise estimate to prevent subtracting discontinuties
                //Smooth noise estimator using LPF

                if (out_filt == 1) {
                        PP[k] = (1 - K) * N[k] + K * last_PP[k];
                        last_PP[k] = PP[k];
                        N[k] = PP[k];
                }
```

```
                ////////////////////////////////////////
                //enhancement 8: residual noise reduction //
                ////////////////////////////////////////
                //if noise over some threshold, get minimum from adjacent frames instead
                if (RNR_flag == 1) {
                        if ((last_N[k] / last_frame_mag[k]) > RNR_threhold) {
                                //over threshold, get min from adjacent
                                RNRbuffer[k] = complex_min(inframe[k],
complex_min(last_frame[k], last_last_frame[k]));
                        }
                        else {
                                //else get last inframe is the future frame
                                RNRbuffer[k] = last_frame[k];
                                switch (gain_var) {
                                //implement gain variations
                                case 0: inframe[k] = rmul(max(lambda, (1 - N[k] /
frame_mag[k])), inframe[k]); break;
                                case 1: inframe[k] = rmul(max(lambda * N[k] /
frame_mag[k], (1 - N[k] / frame_mag[k])), inframe[k]); break;
                                case 2: inframe[k] = rmul(max(lambda * P[k] /
frame_mag[k], (1 - N[k] / frame_mag[k])), inframe[k]); break;
                                case 3: inframe[k] = rmul(max(lambda * N[k] / P[k], (1 -
N[k] / P[k])), inframe[k]); break;
                                case 4: inframe[k] = rmul(max(lambda, (1 - N[k] / P[k])),
inframe[k]); break;
                                case 5: inframe[k] = rmul(max(lambda, sqrt(1 - (N[k] *
N[k]) / (frame_mag[k] * frame_mag[k]))), inframe[k]); break;
                                case 6: inframe[k] = rmul(max(lambda * N[k] /
frame_mag[k], sqrt((1 - (N[k] * N[k]) / (frame_mag[k] * frame_mag[k])))), inframe[k]);
break;
                                case 7: inframe[k] = rmul(max(lambda * P[k] /
frame_mag[k], sqrt((1 - (N[k] * N[k]) / (frame_mag[k] * frame_mag[k])))), inframe[k]);
break;
                                case 8: inframe[k] = rmul(max(lambda * N[k] / P[k],
sqrt((1 - (N[k] * N[k]) / (P[k] * P[k])))), inframe[k]);    break;
                                case 9: inframe[k] = rmul(max(lambda, sqrt((1 - (N[k] *
N[k]) / (P[k] * P[k])))), inframe[k]);  break;
                                default: inframe[k] = rmul(max(lambda, (1 - N[k] /
frame_mag[k])), inframe[k]); break;
                                }
                        }
                        //shuffle buffers
                        last_frame_mag[k] = frame_mag[k];
                        last_last_frame[k] = last_frame[k];
                        last_frame[k] = inframe[k];
                        last_N[k] = N[k];
                }
                else {
                        switch (gain_var) {
                        ////////////////////////////
                        //enhancement 4: case 0-4 //
                        ////////////////////////////
                        //implement different variations of gain factor
                        case 0: inframe[k] = rmul(max(lambda, (1 - N[k] / frame_mag[k])),
inframe[k]); break;
                        case 1: inframe[k] = rmul(max(lambda * N[k] / frame_mag[k], (1 -
N[k] / frame_mag[k])), inframe[k]); break;
                        case 2: inframe[k] = rmul(max(lambda * P[k] / frame_mag[k], (1 -
N[k] / frame_mag[k])), inframe[k]); break;
                        case 3: inframe[k] = rmul(max(lambda * N[k] / P[k], (1 - N[k] /
P[k])), inframe[k]); break;
                        case 4: inframe[k] = rmul(max(lambda, (1 - N[k] / P[k])),
inframe[k]); break;
                        ////////////////////////////
                        //enhancement 5: case 5-9 //
                        ////////////////////////////
                        case 5: inframe[k] = rmul(max(lambda, sqrt(1 - (N[k] * N[k]) /
(frame_mag[k] * frame_mag[k]))), inframe[k]); break;
```

```
                        case 6: inframe[k] = rmul(max(lambda * N[k] / frame_mag[k],
sqrt((1 - (N[k] * N[k]) / (frame_mag[k] * frame_mag[k])))), inframe[k]);    break;
                        case 7: inframe[k] = rmul(max(lambda * P[k] / frame_mag[k],
sqrt((1 - (N[k] * N[k]) / (frame_mag[k] * frame_mag[k])))), inframe[k]);    break;
                        case 8: inframe[k] = rmul(max(lambda * N[k] / P[k], sqrt((1 -
(N[k] * N[k]) / (P[k] * P[k])))), inframe[k]);    break;
                        case 9: inframe[k] = rmul(max(lambda, sqrt((1 - (N[k] * N[k]) /
(P[k] * P[k])))), inframe[k]);  break;
                        ////////////////////////
                        //extra improvements //
                        ////////////////////////
                        case 10: inframe[k] = rmul(max(sqrt(lambda) * N[k], 1 - N[k] /
P[k]), inframe[k]);
                        case 11: {
                                //variation of variable alpha
                                delta = (k < 32) ? 1 : (k < 64) ? 1.5 : 2.5;
                                inframe[k] = rmul(max(lambda, 1 - sqrt((P[k] * P[k] -
alphatmp * delta * N[k] * N[k]) / (frame_mag[k] * frame_mag[k]))), inframe[k]);
                        }
                        default: inframe[k] = rmul(max(lambda, (1 - N[k] / frame_mag[k])),
inframe[k]); break;
                        }
                }
        }
        ////////////////////////////////////////////////
        //compute ifft of frame to go back to time domain //
        ////////////////////////////////////////////////
        if (RNR_flag == 0) {
                ifft(FFTLEN, inframe);
        }
        else {
                ifft(FFTLEN, RNRbuffer);
        }

}

//////////////////////////////////
//enhancement 6: variable alpha //
//////////////////////////////////
/**
 * lookup table created using excel, returns alpha value from snr
 * @param  ratio [float snr]
 * @return       [int alpha]
 */
inline int ilog10c(float ratio) {
        if (ratio <    0.316227766    ) return              20      ;
        else if (ratio <      0.398107171    ) return              19.28   ;
        else if (ratio <      0.501187234    ) return              18.56   ;
        else if (ratio <      0.630957344    ) return              17.84   ;
        else if (ratio <      0.794328235    ) return              17.12   ;
        else if (ratio <      1       ) return              16.4    ;
        else if (ratio <      1.258925412    ) return              15.68   ;
        else if (ratio <      1.584893192    ) return              14.96   ;
        else if (ratio <      1.995262315    ) return              14.24   ;
        else if (ratio <      2.511886432    ) return              13.52   ;
        else if (ratio <      3.16227766     ) return              12.8    ;
        else if (ratio <      3.981071706    ) return              12.08   ;
        else if (ratio <      5.011872336    ) return              11.36   ;
        else if (ratio <      6.309573445    ) return              10.64   ;
        else if (ratio <      7.943282347    ) return              9.92    ;
        else if (ratio <      10      ) return              9.2     ;
        else if (ratio <      12.58925412    ) return              8.48    ;
        else if (ratio <      15.84893192    ) return              7.76    ;
        else if (ratio <      19.95262315    ) return              7.04    ;
        else if (ratio <      25.11886432    ) return              6.32    ;
        else if (ratio <      31.6227766     ) return              5.6     ;
        else if (ratio <      39.81071706    ) return              4.88    ;
        else if (ratio <      50.11872336    ) return              4.16    ;
        else if (ratio <      63.09573445    ) return              3.44    ;
```

```
        else if (ratio <       79.43282347     ) return              2.72    ;
        else    100     return          2       ;
}

//try and implement faster square root, not really faster on DSK
/*
inline float my_sqrt(const float m)
{
        int j=0;
   float i=0;
   float x1,x2;
   while( (i*i) <= m )
          i+=0.1f;
   x1=i;
   for(j=0;j<10;j++)
   {
       x2=m;
      x2/=x1;
      x2+=x1;
      x2/=2;
      x1=x2;
   }
   return x2;
}  */
```