Real Time Digital Signal Processing Speech Enhancement Project

Yong Wen Chua (ywc110) & Ryan Savitski (rs5010)

23rd March 2013

Abstract

This report describes the implementation and analysis of a dynamic real-time noise filtering system for a speech signal, done primarily through spectral subtraction of estimated noise. It will describe and analyse a series of techniques to improve the filter performance, and conclude with the the specific implementation chosen.

Declaration

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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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 offences policy.

Signed: Yong Wen Chua & Ryan Savitski

Contents

List of Figures

1 Introduction

In this digital age, more communication is now happening over networks rather than in person. Driven primarily by telephony, this increase in prevalence of digital communication has made the transmission of clear and noise-free sound more important than ever before. Implementations of such a system is complicated by the fact that the implementers can never fully know what sort of noise environment users are in when they are speaking through the digital communication system. In addition, such a noise reduction system have to run in real-time, or users would not be able to communicate in real-time.

This project aims to implement such a noise-reduction system on a set of noise corrupted sound files in real-time. Various heuristics are used to estimate the noise profile. Once estimated, spectral subtraction is used to "subtract" the noise spectrum estimate to obtain a clearer version of the corrupted speech signal.

In section §2, a basic implementation of the noise filter is explored. This will also establish the basic framework for the enhancements explored in section §3, along with details of the parameters used in those enhancements. In section §4, the combination of the enhancements chosen to be used for the final filter version are described. Finally, further ideas and heuristics that could enhance the filter are discussed in section §5. The code for the project can be found in section §A.

2 Basic Implementation

Filtering of the noisy input signal is done primarily in the frequency domain. This process is illustrated in figure 2.1. The Fast Fourier Transform (DFT) algorithm is employed to perform a Discrete Fourier Transform (DFT) on the input signal. Then assuming that the noise is additive, the noise spectrum is estimated and subtracted from the noisy signal. Finally, an inverse DFT (IDFT) is performed on the input before sending it to the output.

Due to the real-time requirements of the system, it is not plausible to take a DFT of the entire noisy input. Thus, frames are passed through DFT and then processed accordingly. In order to prevent discontinuities at frame boundaries (which manifest as clicking sounds heard in the output), frames are processed in an overlapping manner, with appropriate windowing done on the frames so that the overall gain at the frame boundaries add up to unity.

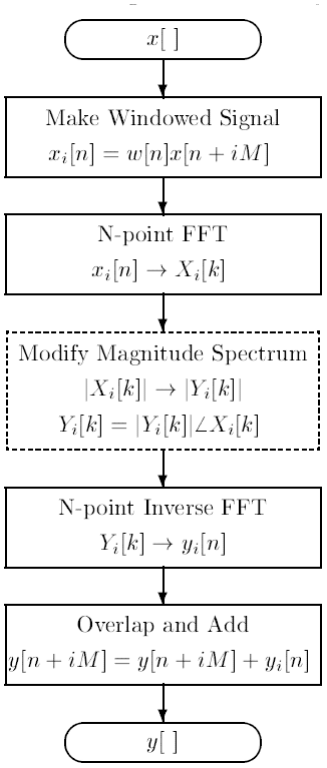


Figure 2.1: Block diagrams of how the samples are processed. (Compernolle 1992)

2.1 Frame Processing Implementation

A frame size of 256 samples was chosen for frame processing, and an over-sampling ratio of four was chosen along the the square root of the Hamming Window. This ensures that there are no sharp discontinuities at frame edges and that the overall gain of the overlapped windows sum to unity. This means that a full frame is taken for frequency domain processing every $1/4$ of a frame, known as a frame segment. Each segment is thus 64 samples long. The processing is illustrated in figure 2.2.

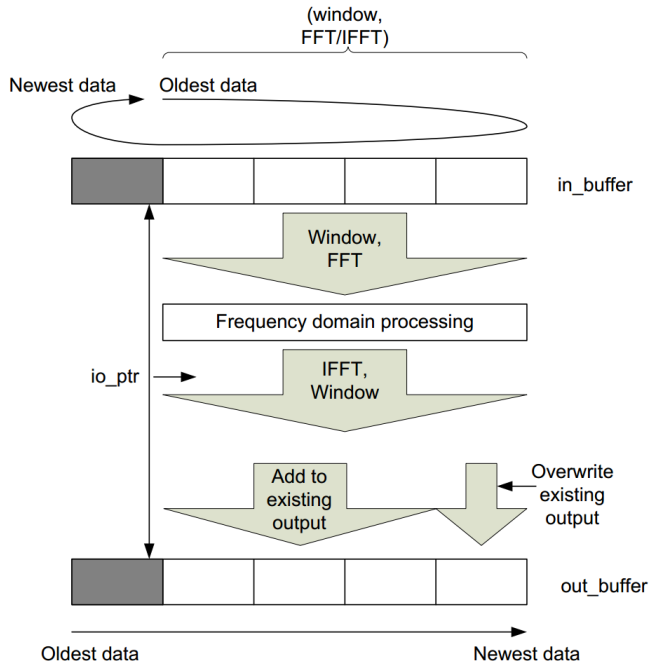


Figure 2.2: Implementation of the overlapped frame processing. (Mitcheson 2013)

The input and output buffer are implemented as circular buffers with five segments each. At any time, four of those input segments will be fully filled (and can be used for frame processing) with one being used to store the inputs coming into the system. At the same time, one of the output segments will have been fully added with its overlapped frame (and therefore ready for output), while the other four frames are being added with the current processed frame. The inputs are read from the input port via an interrupt, and the relevant output is then sent to the output port. This results in a 1.25 frame latency from the input port to the output port.

2.2 Noise Spectrum Estimation

The heuristics used in the estimation of noise assumes that within a 10 second window, the speaker would have at least paused for a fraction of the time. During this time, the spectrum of the input would simply correspond to that of the noise spectrum. Thus, the minimum of the spectrum content over a ten-second sliding window can be used to estimate the noise spectrum. Due to the fact that the phase of the noise spectrum cannot be known, and the fact that slight phase distortion is not audible to the human ears, only the absolute of the spectrum is analysed and processed.

In practice, however, it is not feasible to store and minimise over ten seconds worth of spectral content. Instead, the noise buffer is split into four, each holding a minimum over a 2.5 second period. These four separate noise minimum buffers are implemented as a single circular buffer, with the appropriate program logic to separate the segments. For the current window $M_i(\omega)$ (where $i \in [1,4]$) and an input spectrum of $X(\omega)$, $M_i(\omega) = \min\left[|X(\omega)|, M_i(\omega)\right]$.

When the noise spectrum is to be estimated for use in spectral subtraction, the minimum over the four 2.5 seconds segments are used. Noting that the nature of the naive noise estimation always underestimates the noise content, therefore an over-subtraction factor α is used for compensation. Thus, the minimum estimated noise spectrum $N(\omega)$ is given by

$$N(\omega) = \alpha \min_{i \in [1,4]} \left[M_i(\omega)\right] \tag{2.1}$$

where α was set to a value of 20 for the naive implementation, producing a very aggressive filter. Although this is higher than the suggested values by Berouti et al. (1979), it works decently in practice. However, this results in high amplitude

musical noise and some voice distortion. Lowering α would let more noise through the filter, but at the same time reduce the amount of musical noise (see section 2.3.1) present.

2.3 Noise Spectrum Subtraction

Let the output be $Y(\omega)$, and the input and noise be $X(\omega)$ and $N(\omega)$ respectively. To preserve the phase of $X(\omega)$ (and achieve a zero-phase filtering), only the magnitude should be altered, as mentioned before. Thus, the output is given by

$$|Y(\omega)| = |X(\omega)| - |N(\omega)| = |X(\omega)| \left(1 - \frac{|N(\omega)|}{|X(\omega)|}\right) \tag{2.2}$$

$$Y(\omega) = X(\omega)G(\omega) \tag{2.3}$$

$$G(\omega) = \max\left(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right) \tag{2.4}$$

where $G(\omega)$ is a gain factor that is calculated from the estimated noise spectrum derived from equation (2.2). Because of the fact that the expression $1 - \frac{|N(\omega)|}{|X(\omega)|}$ can potentially go negative (i.e. the noise is over-estimated), $G(\omega)$ should be lower bounded. The input is not simply attenuated to zero (but to the noise floor λ) to reduce the amount of perceivable "musical noise".

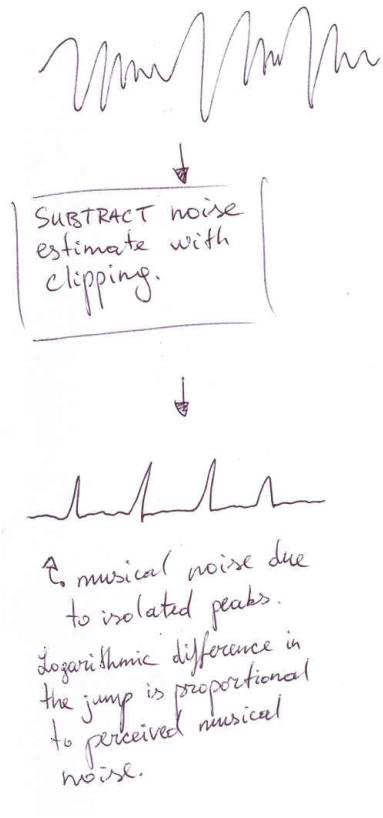
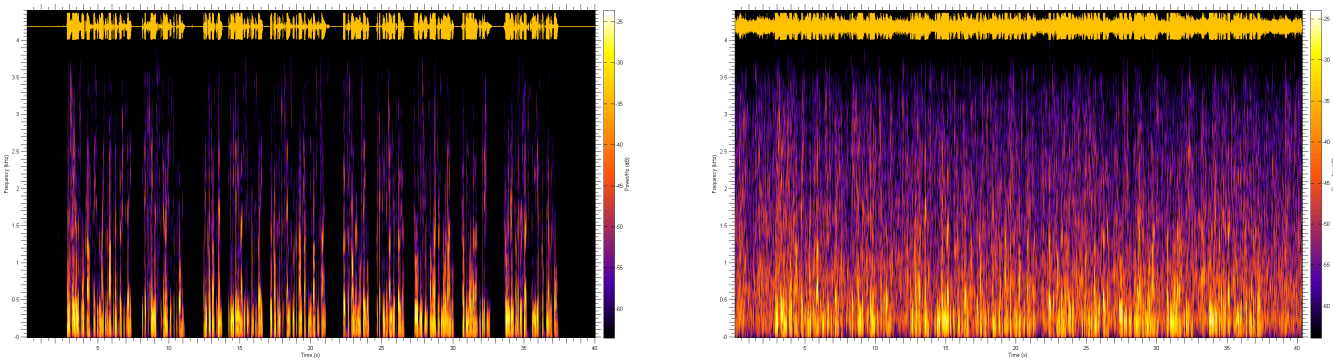


Figure 2.3: Illustration of the generation of musical noise by noise clipping.

2.3.1 Musical Noise

When subtracting an estimate of the noise with clipping, imperfect noise estimation produces isolated peaks in the spectrum of the filtered signal, which is perceived as low-tone musical notes. This is illustrated in figure 2.3. As human sound perception is logarithmic (Compernolle 1992), the difference between the zeroed floor and a sharp peak is very pronounced. Instead, by clipping to a small positive value λ, the differential step between the post-filter noise floor (λ) and the spectral peaks that are a result of imperfect noise estimation can be reduced. In this basic implementation, the spectral noise floor constant λ was chosen to be 0.05 by empirical methods, choosing from the ranges suggested by Berouti et al. (1979). Generally, a higher λ would result in a higher white noise level, but lower perceived musical noise. Lower λ results in a lower white noise output but a more pronounced musical noise effect.

2.4 Evaluation



(a) Spectrogram of the clean file

(b) Spectrogram of "factory2" file

Figure 2.4: Spectrograms of provided files, generated in Matlab

As human perception of volume, noise and in general audio signal clarity is very subjective and lacks a complete mathematical model, spectrograms are used to introduce some form of objectivity when evaluating the performance of the filter. The "factory2" corrupted sample is used for evaluation of the filter. Figure 2.4a shows the spectrogram of the provided clean sound file, and figure 2.4b shows the spectrogram of the "factory2" corrupted signal.

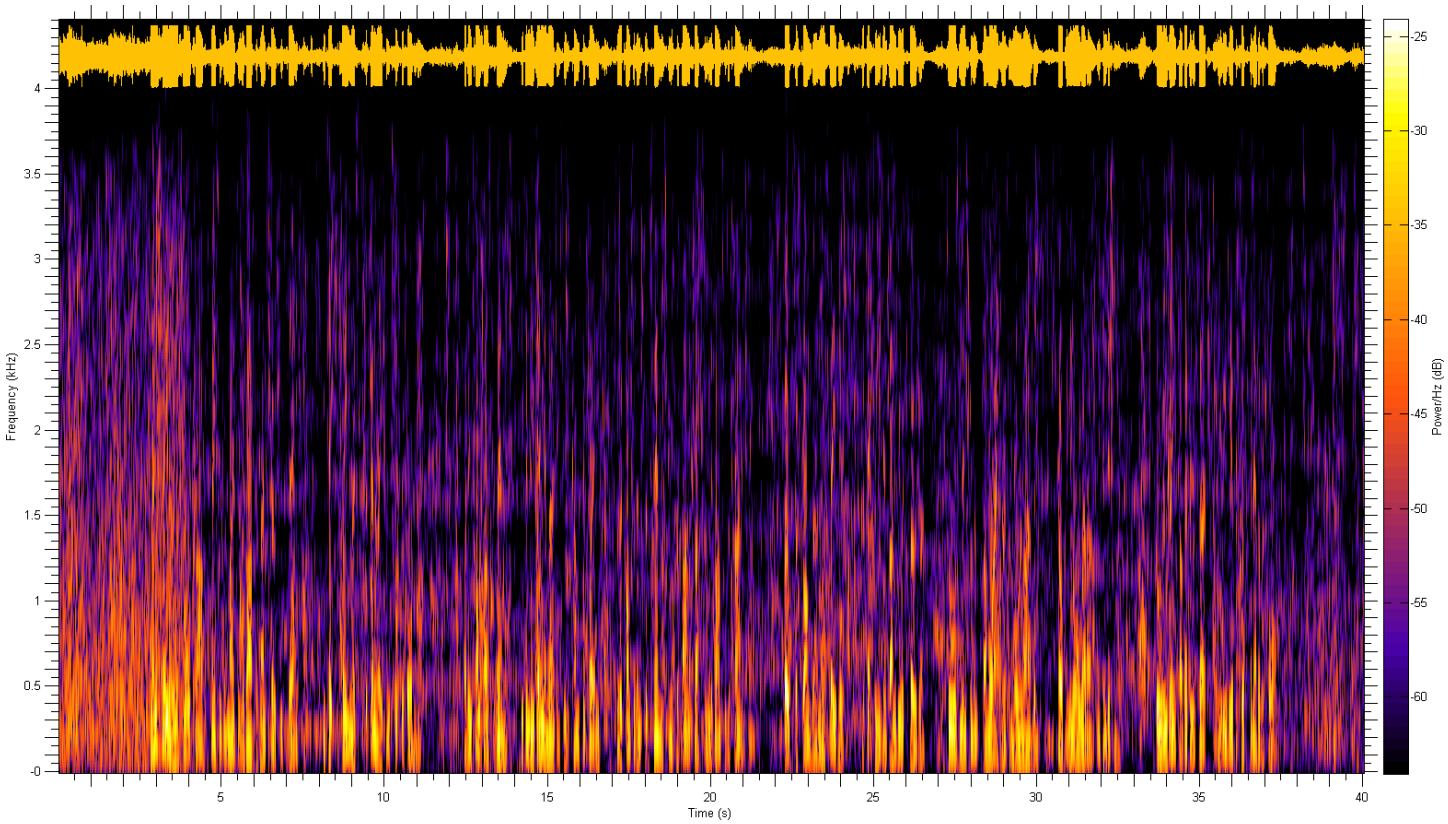


Figure 2.5: Spectrogram of the basic filtering of "factory2" at 8000 Hz sampling frequency.

After applying the basic filter, the results can be seen in figure 2.5. From listening tests, the basic implementation has managed to attenuate the background noise slightly, with darker areas seen in figure 2.5 in between speech. The filter is slow in responding to background noise changes, especially when the noise level transits from a low level to a high level.

This is due to the ten-second window used for noise estimation. As the filter is always conservative about the noise level being low (by always taking the minimum), the lower estimate before the transition is always dominating until the relevant minimum noise buffer is discarded. Due to the high over-subtraction factor α introduced in equation (2.1) to correct the underestimation of noise, some of the speech is also removed and results in the voice being slurry. There is also a very pronounced level of musical noise.

The unnatural sounding musical noise is a result of the magnitude of the frequency exhibiting strong fluctuations in the noisy area (Cappe 1994). Referring to figure 2.4a, it can be seen that the speech is paused between 11-12.5 s and the spectrogram is completely empty during that period. In the filtered version in figure 2.5, it can be seen that there is noise during this period of time, and the magnitude of the frequency bins fluctuates very rapidly, exhibiting the musical noise effect mentioned above (see section 2.3.1).

3 Enhancements

Various enhancements are implemented, and evaluated for their effectiveness in removing the noise from the signal. Some of these enhancements improve the noise removal, while some exacerbate the noise. Each are thus evaluated in turn for their effectiveness, and a final combination of these enhancements is described in section §4.

3.1 Structural Optimisation

The frequency spectrum processing can be sped up by slightly less than a factor of two simply by only processing the first half, plus one, of the frequency bins in the DFT. This is because the inputs to the DFT are real, and thus the DFT of the inputs will be complex conjugates around the middle. For our frame size of 256, where X_k denotes the k^{th} frequency bin, $X_{256-i} = X_{-i} = X_i^*$ for $i \in [1, 127]$, and X_0 and X_{128} are simply real-valued. This also allows some of the buffers used in the enhancements described below to be almost halved in size.

This helps to prevent "clicking" in the output, resulting from frame computation taking longer than one frame length.

3.2 Low-Pass Filter Input for Noise Estimation

A problem with the basic approach is that each frame's DFT is considered alone when updating the estimate of the minimum noise. Therefore, even a single frame with an instantaneous magnitude drop for an DFT bin will potentially stay as the overly conservative noise estimate for the duration of the noise buffer. Since the frame length for 256 samples at 8000 Hz sampling frequency is 32 ms, one particularly underestimated noise value will stay within the minimum noise buffer for a period that can last seconds. This is the reason for a high over-subtraction coefficient for the basic filter above, to compensate for the filter's conservatism of the noise level. Furthermore, for the same reasons of the filter being overly optimistic, any noise with small spectral pauses (that need not be aligned to one frame) will defeat the basic implementation.

3.2.1 Magnitude domain

One way of improving on the naive spectral subtracter is to low-pass filter (LPF) the DFT magnitude bins across time according to:

$$P_t(\omega) = (1-k) \times |X(\omega)| + k \times P_{t-1}(\omega) \tag{3.1}$$

$$k = e^{(-T/\tau)} \tag{3.2}$$

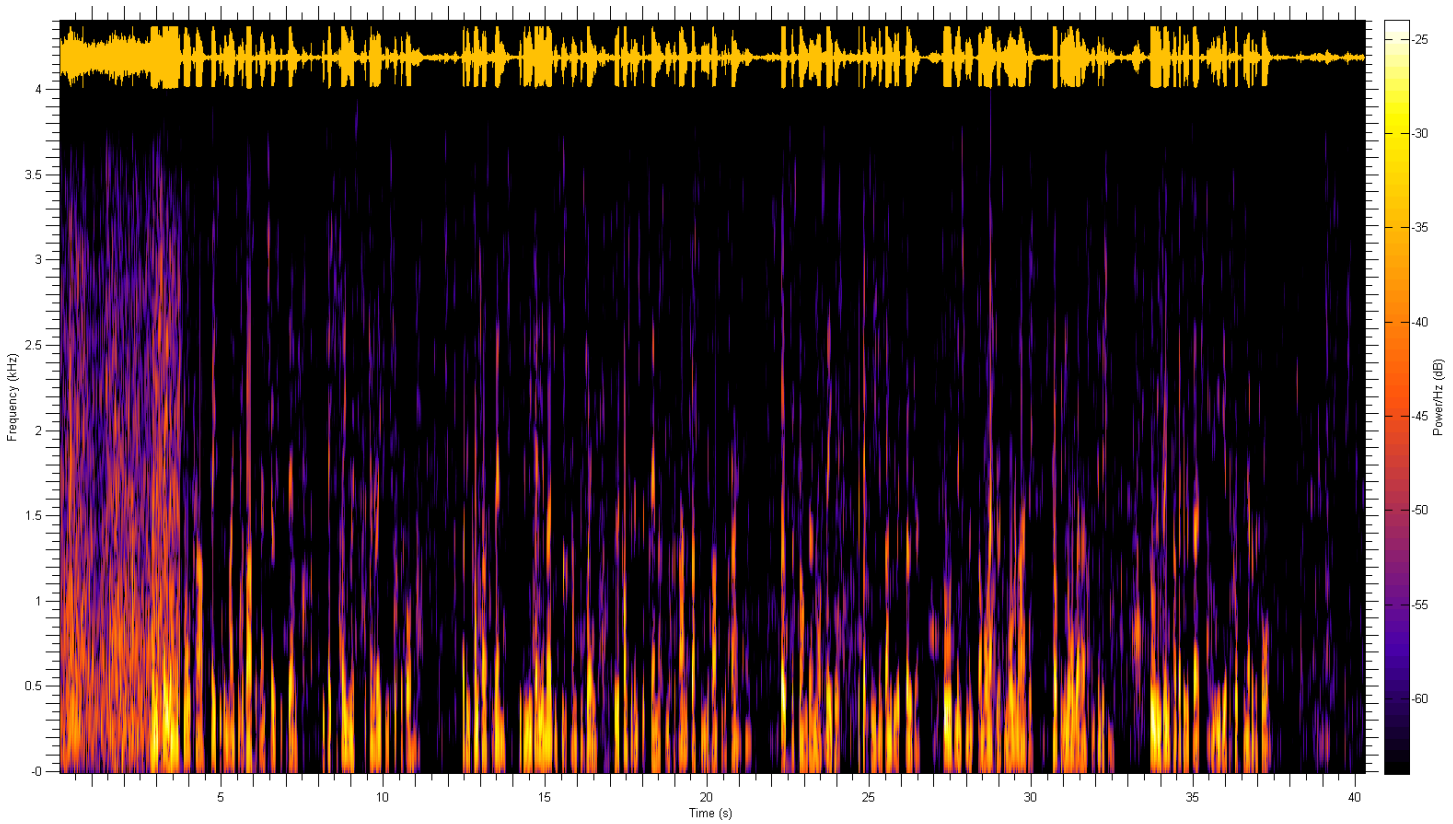


Figure 3.1: Spectrogram of the filtering with the low-pass filter input for spectral content across time

where P_t is the input estimate to be presented to the minimum noise buffer as per equation (2.1), T is the frame rate (the time between frame calculation), and τ is the time constant parameter for this filter.

$P_t(\omega)$ is the low pass filtered DFT bin that corresponds to the persistent average level of the magnitude at that frequency bin, with the smoothness of the filtering controlled by the time constant τ. By taking the minimum of the low pass filtered signal as the noise minimum (equation (2.1)), the filter will not suffer from the degenerating upon encountering sudden magnitude jumps in the noise signal's spectral content. Result is such that the noise buffer's noise estimate now more closely matches the persistent noise level. This is illustrated in figure 3.2.

The over-subtraction coefficient, α, can thus be reduced from 20 to 3.2. This also reduces the musical noise perceived by lowering the level of fluctuation in the magnitude at each frequency bin between clipped and unclipped samples. This is due to the subtracted spectrum being more closely matched to the actual noise level.

In the basic filter, when the input $|X(\omega)|$ falls sharply, this would cause a sharp fall in $|N(\omega)|$ and cause a discontinuity in the output. Low pass filtering removes this discontinuity by introducing a smoother transition. This also has the effect of lowering the responsiveness of the filter, but is worth the trade-off due to much better noise estimation.

The low pass filter can also be seen as a system where the inputs are integrated over a period of time and then averaged. Therefore, by having a longer period to average over, the pauses between speech can be used to get a much more accurate average noise estimate. Figure 3.1 shows the output of the filter being more successful in filtering out the noise when compared to figure 2.5. This is because the filter is now less susceptible to problems caused by a momentary drop in the input. The amount of musical noise is also significantly reduced as well (for example in the pause in speech from 11-12.5s).

The time constant factor τ can be adjusted in this filter, and 50 ms was chosen for the implementation. This value was obtained empirically through blind listening tests with the set of samples given in the range of 20 to 80 ms. This range maximises the integration period of noise by being matched with the inter-word pause timings. Having a higher time constant results in a longer integration period, which when bigger than the pauses between words will start integrating voice spectrum

as part of the noise estimate. A higher τ also means a slightly slower response to noise level change, although the 7.5 seconds period of the full noise buffer rotation latency has a bigger effect on the response speed (see section 2.2). A smaller τ on the other hand will be similar to the basic implementation through averaging over a shorter period.

3.2.2 Power domain

Alternatively, the LPF of the frequency bins can be done in power domain:

$$P_t(\omega) = \sqrt{(1-k) \times |X(\omega)|^2 + k \times [P_{t-1}(\omega)]^2} \quad (3.3)$$

Power domain filtering provides slightly more accurate filtering, because human hearing tracks the power of the signal closer than its magnitude. Hence, the perceivable noise could potentially be reduced.

An approach where the entire noise estimation and noise subtraction is performed in the power domain was explored, converting to magnitude domain only before output. Thus, equation (3.3) becomes

$$P_t(\omega) = (1-k) \times |X(\omega)|^2 + k \times P_{t-1}(\omega) \quad (3.4)$$

and noise subtraction originally in the form of equation (3.8) can be performed by

$$|Y(\omega)| = \sqrt{|X(\omega)|^2 - |N_p(\omega)|} \quad (3.5)$$

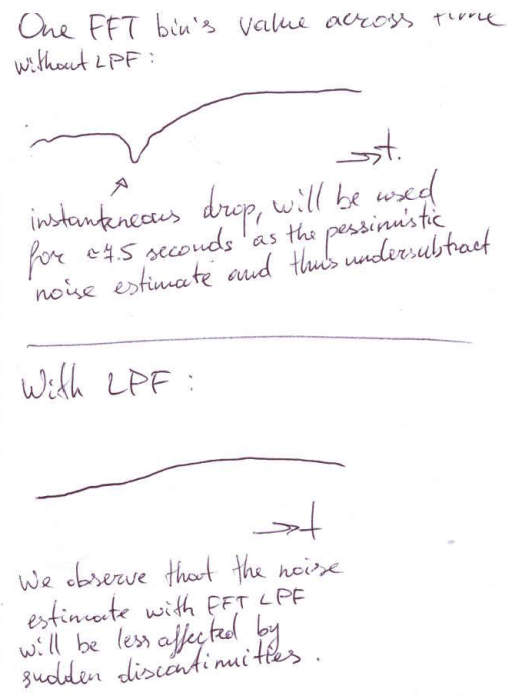


Figure 3.2: Illustration of the low pass filtering of the DFT bins.

where $|N_p(\omega)|$ is the minimum noise estimate in the power domain estimated in the same way as in equation (2.1) .

Testing showed no perceivable difference regardless of whether the filtering was done in the magnitude or power domain (including both variants), agreeing with Compernolle (1992).

3.3 Low-Pass Filter Noise Estimate

Another potential for discontinuities in the output is when the minimum noise buffers are rotated. These steps can be perceivable if the noise level changes significantly. They can be smoothed out by low pass filtering over the noise estimates. This ensures a smoother transition between noise buffer rotations. The filtering is done by

$$|N_{lpf}(w)| = (1-k_n) \times |N(w)| + k_n \times |N_{lpf}(w)| \quad (3.6)$$

$$k_n = e^{(-T/\tau_n)} \quad (3.7)$$

where τ_n is the time constant for this low pass filter. The time constant determines how smooth,and less responsive the transitions will be, with higher time constant giving a smoother transition.

Experimental testing showed that $\tau_n = 100\text{ms}$ gave the best performance for the samples given.

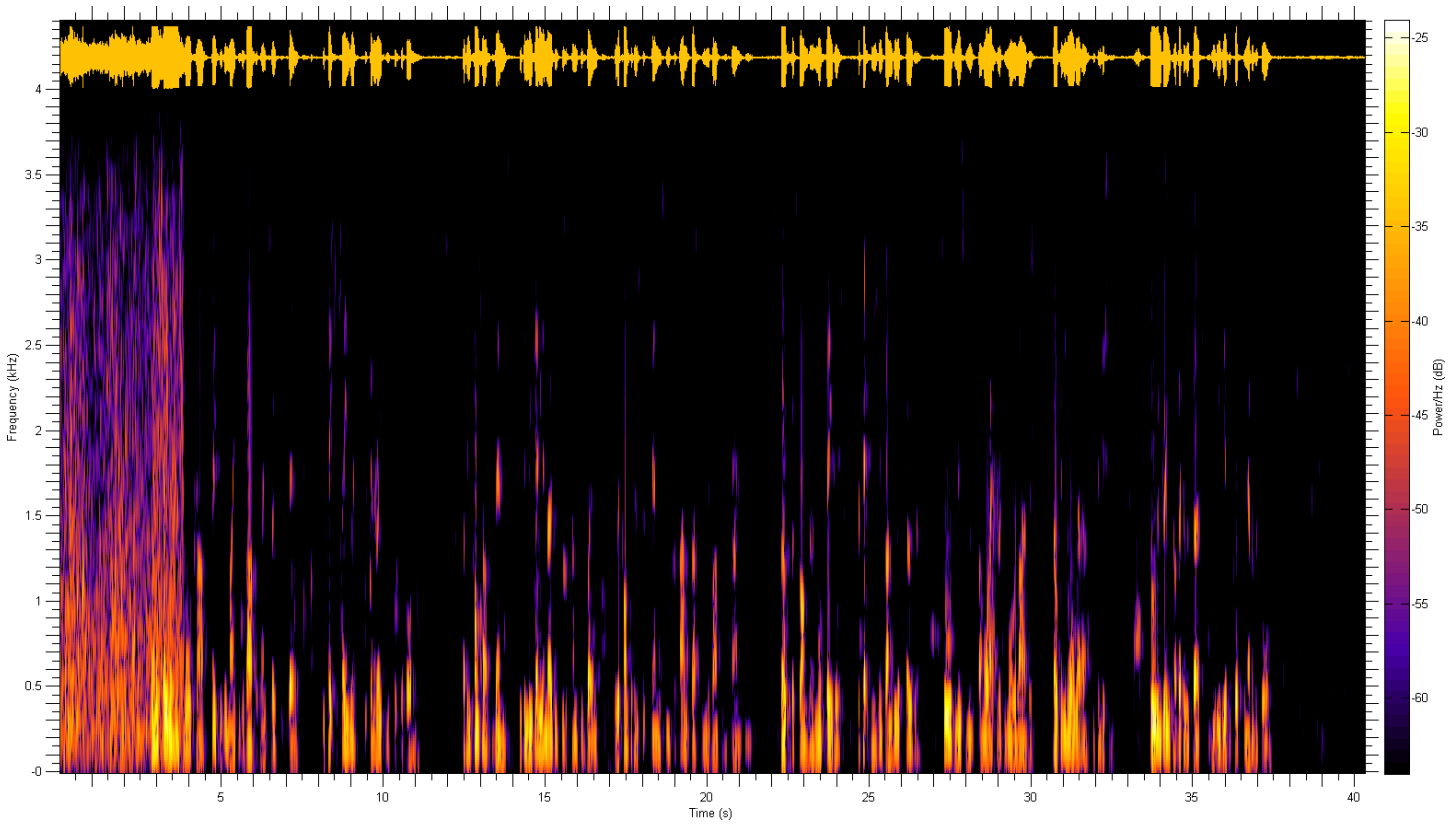


Figure 3.3: Spectrogram using an alternate calculation for $G(\omega)$, combined with input LPF for noise estimation

3.4 Alternate Calculation for $G(\omega)$

The nature of the noise estimation algorithm is such that a lower noise estimate (even momentarily) will cause an under-estimate of the noise spectrum. While this effect is moderated partly by the LPF in section 3.2, it still does not handle situations where the noise level increases. Consider that the factor $G(\omega)$ derived in equation (2.4) can be calculated using another form to perform noise subtraction form the input $X(\omega)$:

$$|Y(\omega)| = |X(\omega)| - |X(\omega)|\frac{|N(\omega)|}{|P(\omega)|} = |X(\omega)|\left(1 - \frac{|N(\omega)|}{|P(\omega)|}\right) \tag{3.8}$$

$$\therefore G(\omega) = \max\left[\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|}\right] \tag{3.9}$$

where $|N(\omega)|$ is the current minimum noise estimate over ten seconds from equation (2.1), and $|P(\omega)|$ is the low-pass filtered noise estimate for the current frame being processed from equation (3.1). equation (3.8) gives rise to different effects depending on the values of $|N(\omega)|$ and $|P(\omega)|$.

3.4.1 Noise Level Matches Minimum Estimate

Consider a situation when the value of $|N(\omega)|$ is equal, or slightly more than$|P(\omega)|$. This will signify that the current minim noise estimate over ten seconds roughly matches the low-pass filtered noise estimate for the current frame being processed. Thus, there is a high confidence that the input $X(\omega)$ consists entirely of only noise. Hence, the fraction $\frac{|N(\omega)|}{|P(\omega)|} \approx 1$ and the entirety of $X(\omega)$ will be subtracted.

3.4.2 Noise Level Increase

Consider that $|N(\omega)|$ remains constant at a previously encountered minima and that the noise level has increased via an increase in $|X(\omega)|$. Then, because of the LPF in section 3.2, $|P(\omega)|$ does not rise sharply, but slowly. Thus, $|P(\omega)|$ is slightly more than $|N(\omega)|$ and $\frac{|N(\omega)|}{|P(\omega)|}$ will have a value that is close to one. In the original form of noise subtraction in equation (2.2), $|X(\omega)|$ will continued to be subtracted by a very low $|N(\omega)|$ while the noise level has risen. This causes the increased noise to not be filtered, and can cause musical noise. In equation (3.8), this will result in a faster reduction in noise as $|X(\omega)|$ will be subtracted by a larger value. A numerical example of this happening is illustrated in table 3.1.

However, as time goes by and $P(\omega)$ increases, $\frac{|N(\omega)|}{|P(\omega)|} \rightarrow |N(\omega)|$ as $|P(\omega)| \rightarrow 1$. This means that, given sufficient time, equation (3.8) will perform as badly, or even worse as equation (2.2) in the event of a noise increase. Thus, the time constant τ in equation (3.1) can be increased to slow down this process so that eventually when the noise buffer described in section 2.2 rotates, the old value will be discarded. Alternatively, the time interval in which the minimum noise is estimated across can be reduced.

$\|N(\omega)\| = 0.1, \|X(\omega)\| = 0.9$	$\|X(\omega)\| - \|N(\omega)\| = 0.8$
$\|P(\omega)\|$	$\|X(\omega)\| - \|X(\omega)\|\frac{\|N(\omega)\|}{\|P(\omega)\|}$
0.2	0.45
0.3	0.6
0.4	0.675
0.5	0.72
0.6	0.75
0.7	0.771429
0.8	0.7875
0.9	0.8

Table 3.1: Numerical example depicting the noise subtraction during an increase in the noise level.

3.4.3 Evaluation

The caveat with this enhancement is that $|P(\omega)|$ is unable to distinguish speech from noise, and thus will cause some of the speech to be removed as well.

The output with this enhancement implemented can be seen in figure 3.3. When compared with the results from figure 3.1, it can be seen that more noise (and musical noise) has been removed. It can be noted that the noise in the period 11-12.5 s when the speaker has paused his speech is almost fully removed. However, it is also observed that some of the speech has also been removed, resulting in a slightly slurry voice.

3.5 Frame lengths

In general, frame lengths should be of powers of two so that the Fast Fourier Transform (FF) used for DFT can be the most efficient. The frame length can be increased to give a better frequency resolution when doing the DFT of the samples. This would give a more granular approach in processing the frequency bins. However, this comes at the expense of temporal resolution, although it is not as important in this system. Due to the use of the frame processing structure described in section 2.1, there is a 1.25 frame delay between the input and output, and increasing the frame length will increase the time delay more. The increase in frame length will also cause more processing to be done per frame, and if the increased processing takes up more time than the frame time interval, it would cause "clicking" to be heard in the final output. The increase in frame length would also increase the heap memory used, and the limited memory would be a limiting factor.

Through experimentation, it was found that increasing the frame length did not yield much noticeable improvements. Reducing the frame length caused a degradation in the filter quality, as the frequency resolution would be reduced. It is noted that a frame length of 256 provides sufficient granularity for a sampling rate of 8 kHz.

3.6 Dynamic Over-subtraction

The over-subtraction factor α described in equation (2.1) can be dynamically adjusted based on the signal to noise ratio (SNR) given by $\frac{|X(\omega)|}{|N(\omega)|}$. When the SNR falls below a certain threshold, the over-subtraction factor α can be increased to cause more of the spectrum to be subtracted. It is known that in the frequency range of 0-50 Hz, no human speech is present. Thus, this frequency range is considered for increased over-subtraction. Alternatively, instead of over-subtracting and allowing these frequency bins to fall to the noise floor level, λ (see equation (2.4)), these frequency bins can simply be zeroed.

Through experimentation, with a low SNR threshold (approximately 2), no discernible difference could be heard and the spectrogram output did not reflect much improvement. When the SNR threshold was raised (to about 5), a degradation in the speech could be heard. This is because the noise level $|N(\omega)|$ is simply a minimum estimate and does not discern between noise and speech. When the 0-50 Hz frequency bins were simply zeroed, some of the low frequency musical noise was eliminated (e.g. in input "car"), and no degradation in speech was perceived.

3.7 Residual Musical Noise Reduction

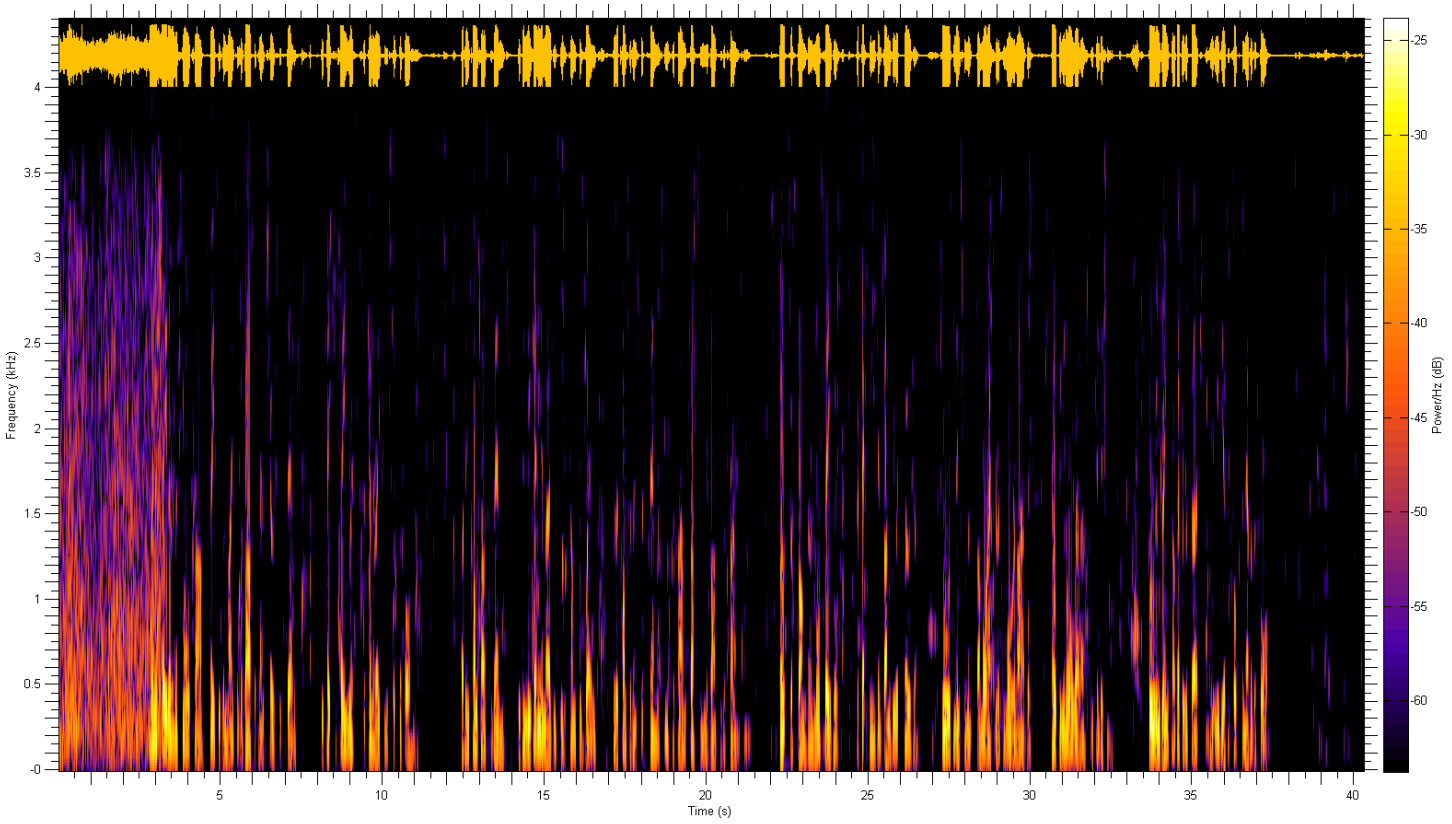


Figure 3.4: Spectrogram of the output with residual musical noise reduction.

The residual musical noise (see section 2.3.1) can be suppressed further. For a specific frequency bin, the musical noise is due to random fluctuation in its amplitude across different frames, therefore, it can be replaced with its minimum value from adjacent frames (Boll 1979).

For a current frame k, if the ratio ${}^{|N(\omega)|}/_{|X(\omega)|}$ is high, more noise is estimated than there is input. Then, take the output $|Y(\omega)| = \min_{i\in\{k-1,k,k+1\}}[|Y_i(\omega)|]$. The motivation for doing so is that if ${}^{|N(\omega)|}/_{|X(\omega)|}$ is high and $|X(\omega)|$ is rapidly

fluctuating, then there is a high probability that it is simply noise, and can be minimised. If $|X(\omega)|$ is constant, it is likely to be low energy speech. Therefore, taking the minimum will retain the speech content.

It should be noted that this would require the low-pass filtering described in section 3.2 to be implemented. Otherwise, every time $|X(\omega)|$ goes lower than $|N(\omega)|$, $|N(\omega)|$ will simply be updated with this new value. The LPF smooths out the sharp drops, thus allowing $|N(\omega)|/|X(\omega)|$ to go above one.

Generally, the threshold for $|N(\omega)|/|X(\omega)|$ should be high so that rapidly fluctuating noise can be filtered out. However, if the threshold is too high, the minimisation might not kick in often enough because $|N(\omega)|$ would simply "catch up" with the lower value of $|X(\omega)|$. Increasing the time constant τ in equation (3.1) would allow the threshold to go higher. A lower threshold would cause less fluctuating noise to be removed.

The spectrogram of the output for a threshold of 3 can be seen in figure 3.4. Compared with the output from figure 3.1, it can be seen that more of the musical noise (c.f. 11-12.5s) has been removed.

3.8 Changing the Noise Estimation Period

The noise estimation window described in section 2.2 can be reduced to allow the noise filter to react faster to changes (especially increase) in noise level. This will also allow the alternate $G(\omega)$ calculation described in section 3.4 to perform better due to the faster rotation of the noise buffers. However, if the speaker does not pause at all during the period, his voice at its lowest level will be taken as noise and then erroneously subtracted from the spectrum as a result of estimating noise over a shorter time period.

By analysing the spectrogram of the clean version of the sound file (see figure 2.4a), it can be seen that the speaker does pause at least once every four seconds (however short). Therefore the total length of all the noise buffers combined contributes to 4 seconds in our implementation.

4 Final Implementation

In the final implementation of the filter, the enhancements described in sections 3.1, 3.2, 3.3, 3.4, 3.6, and 3.8 were implemented. The structural optimisation described in section 3.1 was implemented to improve the computational performance of the filter.

Quantity	Value
Over-subtraction factor, α	3.2
Noise floor, λ	0.05
Input LPF time constant, τ	0.05s
Noise Estimate LPF time constant, τ_n	0.1s

Table 4.1: Parameters for the final implementation.

The input low pass filtering enhancement described in section 3.2 gave the most significant improvement in terms of musical noise reduction. The alternate $G(\omega)$ calculation described in section 3.4 improves upon the LPF by addressing its shortcomings. This requires that τ be set to a slightly higher value. The noise estimation period (section3.8) was also reduced to four seconds to improve the performance of the alternate $|G(\omega)|$ calculation and increase the response rate of the filter. The frequency bins for 0-50 Hz were also zeroed, as described in section 3.6 because no human speech spectral content in that band is insignificant.

The enhancement described in section 3.5 was not implemented due to the adverse effects on the output. The enhancement described in section 3.7, while achieving good results, requires a longer τ to further improve the results already gotten from the other enhancements, and this will cause the noise estimate to be slower in reacting to changes.

The spectrogram of the output with enhancements combined can be found in figure 4.1, and the parameters used can be found in table 4.1. The filter has removed the musical noise present during periods when speech is paused significantly (for

example 11-12.5s) due to the filter being able to better react to changing noise levels. It has also reduced the amount of high frequency noise.

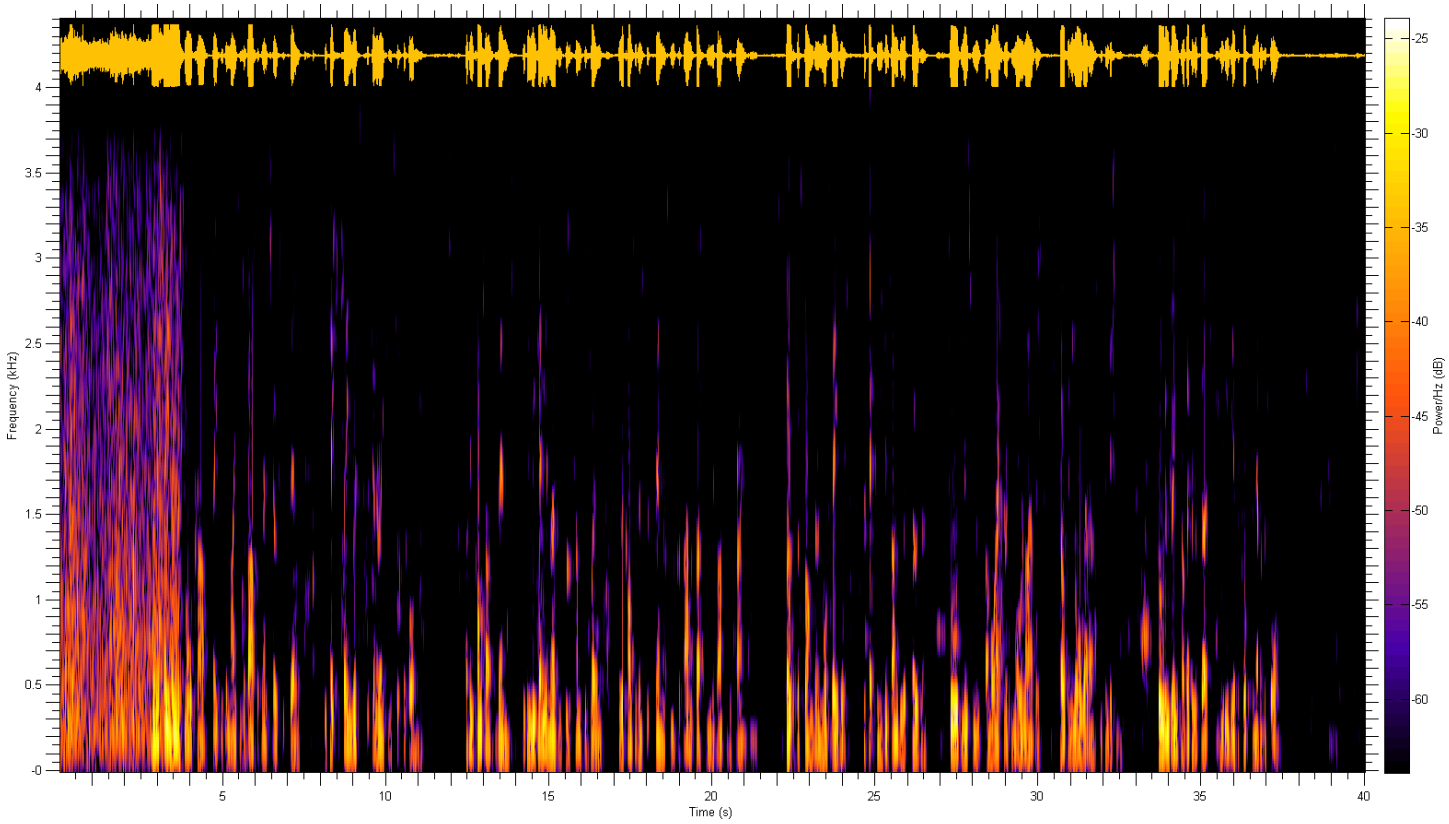


Figure 4.1: Spectrogram of the filtered signal for the final implementation.

5 Further ideas

A very appropriate application of spectral subtraction is mobile telephony due to its simplicity and low computational requirements. In such an application, when the connection for a call is initially established, the caller is not expected to immediately start speaking. As such, the first brief period after establishing a connection can be treated as being pure background noise. Noise can therefore be estimated before speech begins. This could be implemented by simply filling up the noise buffers with the initial samples and then continuing the estimation as before. This will remove the initial "warm up time" that the filter experience due to the way the noise buffers are set up in section2.2. Thus, the person on the other end of the line will receive a filtered output right from the beginning. Additionally, the filter could start tracking the noise even during the connection set-up delay period.

Another trait of mobile telephony is that usually a device will usually only have one or very few associated users. Therefore adaptive filters that are trained to perform better based on the principal traits of the owner's speech can be implemented. This could be achieved in a variety of ways including neural networks, markovian models and Wiener filters (Compernolle 1992).

A Project Code

```c
/*************************** 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>


// min/max macros
 #define min(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a < _b ? _a : _b; })

 #define max(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })


#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/OUTGAIN)   /* Input gain for ADC  */

#define PI 3.141592653589793    // PI defined here for use in your code
#define TFRAME (FRAMEINC/FSAMP)       /* time between calculation of each frame */

// Noise Minimum Buffer Related
#define NOISE_BUFFER_NUM 4.0    // this is the number of noise buffers we are keeping
#define NOISE_TIME 4.0        // the time, in seconds for the period of time that we are keeping the buffers for
#define FRAMES_PER_NOISE_BUF ((int)(NOISE_TIME/NOISE_BUFFER_NUM/TFRAME))  // this is the number of frames processed before a noise buffer rotation happens
```

```
54
55 /****************************** Global declarations ******************************/
56
57 /* Audio port configuration settings: these values set registers in the AIC23 audio
58    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
59 DSK6713_AIC23_Config Config = { \
60       /*************************************************************************/
61       /*   REGISTER              FUNCTION            SETTINGS         */
62       /*************************************************************************/\
63     0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
64     0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
65     0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
66     0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
67     0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
68     0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
69     0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
70     0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
71     0x008d,  /* 8 SAMPLERATE Sample rate control        8 KHZ-ensure matches FSAMP */\
72     0x0001   /* 9 DIGACT     Digital interface activation    On                    */\
73       /*************************************************************************/
74 };
75
76 // Codec handle:- a variable used to identify audio interface
77 DSK6713_AIC23_CodecHandle H_Codec;
78
79 /*************** buffers ***********************/
80 float *inbuffer, *outbuffer;      // Input/output circular buffers
81 complex *frameN;               // buffer for frame N
82 complex *frameN1;          // buffer frame N-1 (enhancement 8)
83 complex *frameN2;          // buffer frame N-2 (enhancement 8)
84 complex *outFrame;           // output frame content (enhancement 8)
85 float *inwin, *outwin;              // Input and output windows coefficients
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
91 float *noiseBuffer;        // the noise circular buffer of all the M subbufs
92 int curM_offset = 0;       // noise sub-buffer pointer (which M buffer we're chosing)
93 float *previousFFTvalue;    // for LPFing the FFT bins
94
95 /* Enhancement 8 buffers */
96 float *previousFrameNXRatio;    // for storing previous |N(w)|/|X(w)|
97 float *frameN1ModY;          // for storing frame N-1 |Y(w)|
98 float *frameN2ModY;          // for storing frame N-2 |Y(w)|
99
100 float *noiseLpfBuffer;          // buffer to LPF the noise estimate to subtract (enhancement 3)
101
102 /*************** parameters *****************/
103 float noiseLambda = 0.05f;        // lower bound coefficient for spectral substractions
104 float noiseOversubtract = 3.2f;     // noise oversubtraction parameter (alpha)
105
106 /* enhancements 1 & 2 & 3 */
107 float freqLpfTimeConstant = 0.06f;    // enhancement 1/2 time constant parameter for LPF
108 float noiseLpfTimeConstant = 0.1f;  // enhancement 3 time constant parameter for LPF
109 float freqLPF_K, noiseLPF_K;        // calculated factor for enhancement 1/2 & 3 respectively
110 float prev_freqLpfTimeConstant = 0;   // previous values for the above time constant values
```

```
111 float prev_noiseLpfTimeConstant=0;  // used for tracking and seeing if the factors needs
        recalculations
112
113 /* enhancement 6 parameters */
114 float enhance6HighFreqLowerBound = 0.00625f;  // fraction of frequency bin to consider as the
        lower bound for "high freq"
115 float enhance6HighFreqUpperBound = 0.99375f;  // the former two should add to one
116 float enhance6LowFreqGain = 10.f;     // factor to multiply with alpha for low freq
117 float enhance6HighFreqGain = 1.f;     // factor to multiply with alpha for high freq
118 float enhance6LowFreqThreshold = 0.5f;    // low freq SNR threshold (NOT in dB)
119 float enhance6HighFreqThreshold = 1.f;    // high freq SNR threshold (NOT in dB)
120
121 // calculated enhancement 6 parameters
122 int enhance6HighFreqBinLowerBound, enhance6HighFreqBinUpperBound; // calculated actual frequency
        bin numbers
123 float prev_enhance6HighFreqLowerBound = 0, prev_enhance6HighFreqUpperBound=0; // previous values
        for the thresholds above to see if recalculation is needed
124
125 // enhancement 8 threshold
126 float enhancement8Threshold = 3.f;      // N/X ratio threshold for enhancement 8
127
128 /************** enhancement switches *****************/
129 short enhancement1 = 1;
130 short enhancement2 = 1;   // this overrides enhancement 1
131 short enhancement3 = 1;
132 short enhancement4Choice = 4; // choose zero to turn enhancement 4/5 off.
133 short enhancement6 = 0;
134 short enhancement8 = 0;
135
136 short enhancementZero = 1;
137
138  /****************************** Function prototypes ******************************/
139 void init_hardware(void);     /* Initialize codec */
140 void init_HWI(void);            /* Initialize hardware interrupts */
141 void ISR_AIC(void);             /* Interrupt service routine for codec */
142 void process_frame(void);       /* Frame processing routine */
143
144 /********************************* Main routine *********************************/
145 void main()
146 {
147
148     int k; // used in various for loops
149
150   /*  Initialize and zero fill arrays */
151
152   /*** buffers ***/
153   inbuffer  = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
154     outbuffer = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
155
156   frameN    = (complex *) calloc(FFTLEN, sizeof(complex));  /* FrameN for processing*/
157   frameN1   = (complex *) calloc(FFTLEN, sizeof(complex));  /* FrameN-1 for processing*/
158   frameN2   = (complex *) calloc(FFTLEN, sizeof(complex));  /* FrameN-2 for processing*/
159   outFrame  = (complex *) calloc(FFTLEN, sizeof(complex));  /* final Output Frame */
160
161     inwin   = (float *) calloc(FFTLEN, sizeof(float));  /* Input window */
162     outwin    = (float *) calloc(FFTLEN, sizeof(float));  /* Output window */
163
```

```
164     noiseBuffer     = (float *) calloc(NOISE_BUFFER_NUM*(FFTLEN/2 + 1), sizeof(float)); // noise
            estmiation buffer
165
166     previousFFTvalue  = (float *) calloc(FFTLEN/2+1, sizeof(float));    // enhancement 2 buffer
167
168     noiseLpfBuffer    = (float *) calloc(FFTLEN/2+1, sizeof(float));    // enhancement 3 buffer
169
170     /* enhancement 8 buffers */
171     previousFrameNXRatio  = (float *) calloc(FFTLEN/2+1, sizeof(float));
172     frameN1ModY       = (float *) calloc(FFTLEN/2+1, sizeof(float));
173     frameN2ModY       = (float *) calloc(FFTLEN/2+1, sizeof(float));
174
175   /* initialize board and the audio port */
176     init_hardware();
177
178     /* initialize hardware interrupts */
179     init_HWI();
180
181   /* initialize algorithm constants */
182
183     for (k=0;k<FFTLEN;k++)
184   {
185     inwin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
186     outwin[k] = inwin[k];
187   }
188     ingain=INGAIN;
189     outgain=OUTGAIN;
190
191
192     /* main loop, wait for interrupt */
193     while(1)
194     {
195       // recalculate values if necessary (for manual runtime tweaking)
196       if (prev_freqLpfTimeConstant != freqLpfTimeConstant)
197       {
198         prev_freqLpfTimeConstant = freqLpfTimeConstant;
199         freqLPF_K = exp(-1.f*TFRAME/freqLpfTimeConstant);
200       }
201
202       if (prev_noiseLpfTimeConstant != noiseLpfTimeConstant)
203       {
204         prev_noiseLpfTimeConstant = noiseLpfTimeConstant;
205         noiseLPF_K = exp(-1.f*TFRAME/noiseLpfTimeConstant);
206       }
207
208       if (prev_enhance6HighFreqLowerBound != enhance6HighFreqLowerBound)
209       {
210         prev_enhance6HighFreqLowerBound = enhance6HighFreqLowerBound;
211         enhance6HighFreqBinLowerBound = (int) (FFTLEN*enhance6HighFreqLowerBound);
212       }
213
214       if (prev_enhance6HighFreqUpperBound != enhance6HighFreqUpperBound)
215       {
216         prev_enhance6HighFreqUpperBound = enhance6HighFreqUpperBound;
217         enhance6HighFreqBinUpperBound = (int) (FFTLEN*enhance6HighFreqUpperBound);
218       }
219
```

```
220         process_frame();
221
222     }
223 }
224
225 /******************************** init_hardware() ********************************/
226 void init_hardware()
227 {
228     // Initialize the board support library, must be called first
229     DSK6713_init();
230
231     // Start the AIC23 codec using the settings defined above in config
232     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
233
234   /* Function below sets the number of bits in word used by MSBSP (serial port) for
235   receives from AIC23 (audio port). We are using a 32 bit packet containing two
236   16 bit numbers hence 32BIT is set for  receive */
237   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
238
239   /* Configures interrupt to activate on each consecutive available 32 bits
240   from Audio port hence an interrupt is generated for each L & R sample pair */
241   MCBSP_FSETS(SPCR1, RINTM, FRM);
242
243   /* These commands do the same thing as above but applied to data transfers to the
244   audio port */
245   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
246   MCBSP_FSETS(SPCR1, XINTM, FRM);
247
248
249 }
250 /******************************** init_HWI() ************************************/
251 void init_HWI(void)
252 {
253   IRQ_globalDisable();      // Globally disables interrupts
254   IRQ_nmiEnable();        // Enables the NMI interrupt (used by the debugger)
255   IRQ_map(IRQ_EVT_RINT1,4);   // Maps an event to a physical interrupt
256   IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
257   IRQ_globalEnable();       // Globally enables interrupts
258
259 }
260
261 /******************************** process_frame() ********************************/
262 void process_frame(void)
263 {
264   int i, j, k, m;   // various loop counters
265   float noiseFactor, noiseMin;  // noise subtraction
266   float noiseFactorA, noiseFactorB; // enhancement 4
267   float x;      // holds the abs for the current sample
268
269   static int frameCounter = 0;    // frame counter for noise buffer rotation
270
271   short rotatedM;   // whether noise buffer rotation has been done
272   float noiseVote;  // low pass filtered noise estimate P(w) (enhancement 1/2)
273
274   int io_ptr0;   // IO pointer
275
276   /* work out fraction of available CPU time used by algorithm */
```

```
277   cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;
278
279   /* wait until io_ptr is at the start of the current frame */
280   while((io_ptr/FRAMEINC) != frame_ptr);
281
282   /* then increment the framecount (wrapping if required) */
283   if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;
284
285   /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where the
286   data should be read (inbuffer) and saved (outbuffer) for the purpose of processing */
287   io_ptr0=frame_ptr * FRAMEINC;
288
289   /* copy input data from inbuffer into inframe (starting from the pointer position) */
290
291   m=io_ptr0;
292     for (k=0;k<FFTLEN;k++)
293   {
294     frameN[k].r = inbuffer[m] * inwin[k];
295     frameN[k].i = 0.f;
296     if (++m >= CIRCBUF) m=0; /* wrap if required */
297   }
298
299   /************************ DO PROCESSING OF FRAME  HERE *************************/
300
301   fft(FFTLEN, frameN);  // FFT of this frame
302
303   // Noise minimum buffer handling
304   if (++frameCounter >= FRAMES_PER_NOISE_BUF) // rotate noise buffer if time period passed
305   {
306     frameCounter = 0;
307
308     if (++curM_offset >= NOISE_BUFFER_NUM) // circular buffer wrap for noise bufs
309       curM_offset = 0;
310
311     // M buffers have been rotated, set corresponding flag
312     rotatedM = 1;
313   }
314   else
315   { // no rotations, usual operation
316     rotatedM = 0;
317   }
318
319   // iterate over fft bins
320   /*
321     The bins are complex-conjugate symmetrical about bin 128
322     So we just process bins 0 - 128.
323    **/
324   for (i = 0; i <= FFTLEN/2; i++)
325   {
326     x = cabs(frameN[i]); // absolute of the signal's fft bin
327
328     noiseVote = x; //default
329
330     ////////////////////////////////////////////////////////////////////////
331     // Optional enhancements
332     if (enhancement1 && !enhancement2) // LPF the FFT bins
333     {
```

```
334       noiseVote = (1-freqLPF_K)*x + freqLPF_K*(previousFFTvalue[i]);
335       previousFFTvalue[i] = noiseVote;
336     }
337     if (enhancement2) // power domain LPF of FFT bins
338     {
339       noiseVote = sqrt((1-freqLPF_K)*x*x + freqLPF_K*(previousFFTvalue[i]*previousFFTvalue[i]));
340       previousFFTvalue[i] = noiseVote;
341     }
342
343     ////////////////////////////////////////////////////////////////////////
344
345     // if M buffers rotated -> overwrite bin with new vote
346     // store the minimum of the current noise value in M0 and the new bin value
347     *(noiseBuffer + curM_offset*(FFTLEN/2 + 1) + i) = (rotatedM) ? noiseVote : min(noiseVote, *(
            noiseBuffer + curM_offset*(FFTLEN/2 + 1) + i)); //TODO: []?
348
349     ////////////////////////////////////////////////////////////////////////
350     // iterate over noise min buffers and select the smallest bin value
351
352     noiseMin = *(noiseBuffer + i);
353     for (j = 1; j < NOISE_BUFFER_NUM; ++j)
354     {
355       noiseMin = min(noiseMin, *(noiseBuffer + j*(FFTLEN/2 + 1) + i));
356     }
357     // oversubstract by alpha coefficient
358     noiseMin *= noiseOversubtract;
359
360     ////////////////////////////////////////////////////////////////////////
361
362
363     /* enhancement 3 - LPF noise estimate */
364     if (enhancement3)
365     {
366       noiseLpfBuffer[i] = (1-noiseLPF_K)*noiseMin + noiseLPF_K*noiseLpfBuffer[i];
367       noiseMin = noiseLpfBuffer[i];
368     }
369
370     /* Enhancement 6 - further noise overestimation with a sharp 0/1 cutoff */
371     if (enhancement6)
372     {
373       float SNR = x/noiseMin;
374       if (i > enhance6HighFreqBinLowerBound && i < enhance6HighFreqBinUpperBound)
375       { // high frequency handling
376         if (SNR < enhance6HighFreqThreshold )
377           noiseMin *= enhance6HighFreqGain;
378       }
379       else
380       { // low frequency handling
381         if (SNR < enhance6LowFreqThreshold  )
382           noiseMin *= enhance6LowFreqGain ;
383       }
384     }
385
386     /* enhancement 4 & 5 */
387     switch (enhancement4Choice)   // calculate G(w), equation used chosen based on switch value
388     {
389       float temp;
```

```
390       case 1:
391         temp = noiseMin/x;
392         noiseFactorA = noiseLambda * temp;
393         noiseFactorB = 1.f - temp;
394         break;
395       case 2:
396         noiseFactorA = noiseLambda * noiseVote/x;
397         noiseFactorB = 1.f - noiseMin/x;
398         break;
399       case 3:
400         temp = noiseMin/noiseVote;
401         noiseFactorA = noiseLambda * temp;
402         noiseFactorB = 1.f - temp;
403         break;
404       case 4:
405         noiseFactorA =  noiseLambda;
406         noiseFactorB = 1.f -  noiseMin/noiseVote;
407         break;
408
409       /* enhancement 5 power handling */
410       case 5: // power version of no enhancement 4
411         noiseFactorA =  noiseLambda;
412         noiseFactorB =  sqrt(1.0 - noiseMin*noiseMin/(x*x));
413         break;
414       case 6: // power version of enhancement 4 - 1
415         temp = noiseMin*noiseMin/(x*x);
416         noiseFactorA = noiseLambda * sqrt(temp);
417         noiseFactorB = sqrt(1.f - temp);
418         break;
419       case 7: // power version of enhancement 4 - 2
420         noiseFactorA = noiseLambda * sqrt(noiseVote*noiseVote/(x*x));
421         noiseFactorB = sqrt(1.f - noiseMin*noiseMin/(x*x));
422         break;
423       case 8: // power version of enhancement 4 - 3
424         temp = noiseMin*noiseMin/(noiseVote*noiseVote);
425         noiseFactorA = noiseLambda * sqrt(temp);
426         noiseFactorB = sqrt(1.f - temp);
427         break;
428       case 9: // power version of enhancement 4 - 4
429         noiseFactorA =  noiseLambda;
430         noiseFactorB = sqrt(1.f -  noiseMin*noiseMin/(noiseVote*noiseVote));
431         break;
432       /* enhancement 4 & 5 off */
433       case 0:
434       default:
435         noiseFactorA = noiseLambda;
436         noiseFactorB = 1.0 - noiseMin/x;
437         break;
438     }
439
440     ////////////////////////////////////////////////////////////////////////
441     // Perform spectral substraction
442
443     noiseFactor = max(noiseFactorA, noiseFactorB);
444     frameN[i] = rmul(noiseFactor, frameN[i]);
445
446     ////////////////////////////////////////////////////////////////////////
```

```
447     /* Enhancement 8 further processing */
448     if (enhancement8)
449     {
450       x *= noiseFactor;     // absolute of Yn
451 
452       // check if previous frame N/X ratio is above a certain threshold
453       if (previousFrameNXRatio[i] > enhancement8Threshold)
454       {
455         // find the frame with the min abs(Y) and use that frame as the output
456         if (x < frameN1ModY[i] && x < frameN2ModY[i])
457           outFrame[i] = frameN[i];
458         else if (frameN1ModY[i] < x && frameN1ModY[i] < frameN2ModY[i])
459           outFrame[i] = frameN1[i];
460         else
461           outFrame[i] = frameN2[i];
462       }
463       else  // below threshold? just output the previous frame
464       {
465         outFrame[i] = frameN1[i];
466       }
467 
468       previousFrameNXRatio[i] = noiseMin/x;   // update the N/X ratio
469       frameN2ModY[i] = x;   // overwrite the value for N-2 frame
470     }
471     else // plain old no enhancement8
472     {
473       outFrame[i] = frameN[i]; // direct assignment
474     }
475 
476     if ( enhancementZero && (i == 0 || i == 1) )
477     {
478       outFrame[i] = cmplx(0,0);
479     }
480   }
481 
482   if (enhancement8) // if enhancement 8, swap relevant buffers
483   {
484     complex *tempFrame;
485 
486     tempFrame = frameN2;
487     frameN2 = frameN1; // new N-2 frame is old N-1 frame
488     frameN1 = frameN;  // new N-1 frame is old N frame
489     frameN = tempFrame; // new N frame reuses the old N-2 frame buffer for new input incoming
490 
491     // swap |Y(w)| buffers
492     float *temp;
493 
494     temp = frameN2ModY;   // note that the mod Y for frame N is already stored here
495     frameN2ModY = frameN1ModY;
496     frameN1ModY = temp;
497 
498   }
499 
500   /*
501     Bins 129 - 255 are complex conjugates of bins 127-1 respectively
502    */
503   for (i = FFTLEN/2+1 ; i < FFTLEN; i++)
```

```
504   {
505     outFrame[i] = conjg(outFrame[FFTLEN - i]);
506   }
507
508   ifft(FFTLEN, outFrame); // perform inverse FFT to return us back to time domain
509
510   /************************************************************************/
511
512     /* multiply outframe by output window and overlap-add into output buffer */
513
514   m=io_ptr0;
515
516     for (k=0;k<(FFTLEN-FRAMEINC);k++)
517   {                          /* this loop adds into outbuffer */
518       outbuffer[m] = outbuffer[m]+outFrame[k].r*outwin[k];
519     if (++m >= CIRCBUF) m=0; /* wrap if required */
520   }
521     for (;k<FFTLEN;k++)
522   {
523     outbuffer[m] = outFrame[k].r*outwin[k];    /* this loop over-writes outbuffer */
524       m++;
525   }
526 }
527 /************************** INTERRUPT SERVICE ROUTINE  ****************************/
528
529 // Map this to the appropriate interrupt in the CDB file
530
531 void ISR_AIC(void)
532 {
533   short sample;
534   /* Read and write the ADC and DAC using inbuffer and outbuffer */
535
536   sample = mono_read_16Bit();
537   inbuffer[io_ptr] = ((float)sample)*ingain;
538     /* write new output data */
539   mono_write_16Bit((int)(outbuffer[io_ptr]*outgain));
540
541   /* update io_ptr and check for buffer wraparound */
542
543   if (++io_ptr >= CIRCBUF) io_ptr=0;
544 }
545
546 /************************************************************************/
```

References

Berouti, M., Schwartz, R. & Makhoul, J. (1979), 'Enhancement of speech corrupted by acoustic noise'.

Boll, S. F. (1979), 'Supression of acoustic noise in speech using spectral subtraction'.

Cappe, O. (1994), 'Elimination of the musical noise phenomenon with the ephraim and malah noise suppressor'.

Compernolle, D. V. (1992), 'Dsp techniques for speech enhancement', p. 10.

Mitcheson, P. D. (2013), 'Real time digital signal processing – project: Speech enhancement', p. 22.