Real-Time Digital Signal Processing

Speech Enhancement Project Report

Archit Sharma: 01199766

Tomasz Bialas: 01205145

Contents

1 Introduction

This project aims to implement a speech enhancer, using the DSK6713 DSP board in real-time. The presented technique will be based on the minimum noise buffering, which estimates the noise present by finding the minimum magnitudes of frequency bins over a given time period. This is then removed from the current input signal. More details of this process are discussed in Section 2. Modifications and extra enhancements made on top of this implementation are discussed and reviewed in Section 3, after which a final implementation is built and tested.

To test the speech enhancer, an audio file from SCRIBE is edited with various noises applied, and then, the speech enhancer is used to see how closely the signal is transformed into the original recording.

2 Minimum Noise Buffer

The technique used to estimate the noise of the signal is known as *spectral subtraction* [1]. It assumes a noisy signal $X(\omega)$ is equal to a clean signal $Y(\omega)$ plus some noise $N(\omega)$. Therefore, to return $Y(\omega)$, an estimate for $N(\omega)$ must be found first and then spectral subtraction $X(\omega) - N(\omega)$ performed to obtain $Y(\omega)$.

To find $N(\omega)$, this technique takes advantage of the fact that a speech signal is being processed, and thus assumes that within a time interval of approximately 10 s, the speaking person will pause to take a breath (i.e. $Y(\omega) \approx 0$ when the speaker breathes, $X(\omega) \approx N(\omega)$ and if $N(\omega)$ is assumed to be fairly constant, then $N(\omega)$ can be approximated for the time interval at this point). Determining the point at which the speaker takes a breath is done by finding the magnitude of each frequency bin for each input frame and by finding the minimum magnitude for each frequency bin over the time interval.

While this method would work, it requires the DSP to store the magnitude spectrum of each frame over the 10 s period. A frame is processed over a 32 ms so 312.5 magnitude spectra need to be stored as 32-bit floats, and if an FFT of size 256 is used then the memory needed is 320kB. A way to reduce this memory requirement is to split the 10 s interval into four 2.5 s intervals and store the smallest recorded magnitude per frequency bin over these smaller intervals. After 2.5 s, the buffers rotate, with the oldest buffer then getting overwritten by the next input frame. $N(\omega)$ is then estimated by selecting the magnitude spectrum with the smallest total magnitude across all frequency bins from the four buffers. This method provides a decent estimation of $N(\omega)$ while storing only 4kB of data, a considerable saving.

The second part of the process is to subtract $N(\omega)$ from $X(\omega)$. However, the phase of the noise is unknown and instead, the magnitudes are subtracted only:

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

where $g(\omega) = \max\{\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\}$ to prevent negative values. It is important to note that broadband noise is random and may have peaks at random positions in the spectrum, which change location at every frame. As such, zeroing the spectrum may cause large valleys in the spectrum, causing "musical noise" with rapidly changing notes. In order to limit this effect, a small constant λ is introduced to prevent very high valleys and peaks in the spectrum. Typically, $\lambda = 0.01$ to $\lambda = 0.1$. As $N(\omega)$ is the estimated minimum noise over a period, it is likely that the noise applied to $X(\omega)$ at the current frame will be greater than the minimum being used, so $N(\omega)$ is then scaled by a constant α, initially set to 20.

The baseline algorithm was implemented and tested qualitatively in order to compare any enhancements and modifications performed. The baseline approach was found to have good a broadband noise removal, at the cost of high levels of musical noise added.

3 Enhancements

Several methods were used to improve the noise reduction of the original algorithm are described in Berouti et al., (1979) [1]. While preserving an acceptable quality of speech, only a few of those were selected for the final implementation. This was caused by the limited processing resources of the DSP and the fact that execution must be kept in real-time. Additionally, some enhancements did not improve the results and degraded the final solution.

3.1 Input filtering for noise estimation

3.1.1 Low Pass Filter, magnitude

An additional array containing the low pass filtered version of the input array was added. A pointer for the input to the noise estimation algorithm was also added, such that the estimate is performed on the low-pass version. This new array was computed based on the following formula:

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

where $k = \exp(-T/\tau)$. This method allowed for a noticeably improved noise suppression, and allowed to decrease the α value from 20 to 5. As a side effect, however, musical noise was added in the system, most likely resulting from the higher level difference between speech and the suppressed parts of the spectrum. This method was retained for the final implementation of the speech enhancement system.

3.1.2 Low Pass Filter, Power

A second version of the low pass filter enhancement was added, which instead of applied the formula to the magnitude spectrum $X(\omega)$, is applied to $X(\omega)^2$. The values are then converted back to magnitude values for further processing, as square root is computed.
Filtering in the power spectrum was expected to yield superior results to magnitude, as it should allow for more effective estimation of the noise and thus stronger suppression. During testing, better noise reduction was observed, however higher amounts of musical noise and heavy distortion were applied to the voice of the speaker. It was effectively "drowned out" by the musical noise. Due to those effects, this method was not selected for the final implementation.

3.1.3 Band Pass Filter

An attempt at bandpass filtering for more effective noise reduction was made by using the spectrum from approximately 300 Hz to 3.4 kHz, as typically used in telephone systems. As no improvement was achieved and the levels of musical noise resulted in unintelligible rendering of the speech, this method was not retained for the final implementation.

3.2 Alternative Noise Reduction Formulae

The initial noise removal is done by subtraction of the estimated noise magnitude through the formula $g(\omega) = \max\{\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\}$. In this section, several alternative formulae are implemented, and compared against the reference original implementation.

In the test code during the project development, a large `switch` statement was used to switch between different formulas, and compare their effects when integrated to the system.

$$g(\omega) = \max\left\{\lambda\frac{|N(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right\} \quad (3) \qquad g(\omega) = \max\left\{\lambda\frac{|P(\omega)|}{|X(\omega)|}, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right\} \quad (4)$$

$$g(\omega) = \max\left\{\lambda\frac{|N(\omega)|}{|P(\omega)|}, 1 - \frac{|N(\omega)|}{|P(\omega)|}\right\} \quad (5) \qquad g(\omega) = \max\left\{\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|}\right\} \quad (6)$$

An additional switch has been implemented to test those formulae, replacing noise estimates in magnitude to estimates in the power domain, as calculated in the array mentioned in section 3.1. However, due to increased musical noise, crackle and occasional voice distortion, power domain noise estimations were not selected for the final implementation.

After testing, the equation (5) was identified as ineffective and subjectively marked as worse than the baseline in terms of noise spectrum removal. This may be explained by its use of the low-pass noise estimate as the denominator of the attenuation equation, which gives a value that is a function of those quantities, and is not directly related to the original signal and the amount of noise that requires removal. This equation did not cause musical noise or distortion but low noise rejection rendered it unfit for purpose.

The equation (6), while constructed similarly to (5), was found to have a slightly higher noise removal effect on the output. Broadband noise remained high, and additional musical noise was introduced.

Finally, the equations (3) and (4) were found to give adequate wideband noise reduction with bearable amounts of musical noise introduced. They also showed lower levels of musical noise compared to the baseline. However, those implementations introduced a decrease in quality of the speech as well as crackling into the sound.

3.3 Oversubtraction

3.4 Residual Noise Reduction

Residual noise reduction technique, as described in Boll, (1979) [2], is said to reduce musical noise. The process shown by the equation (7) states that if signal after processing $X(\omega')$ is less than some threshold η, then the output is the minimum of $X(\omega')$ across three adjacent frames - the current frame, the past frame and the next frame. This means that the output must be delayed by minimum of one frame to do this process. However, the delay of at least 32 ms would not be hugely noticeable to the human ear.

$$Y(\omega)_t = \begin{cases} X(\omega')_t, & X(\omega')_t \geq \eta \\ \upsilon \quad \text{where} \quad \upsilon = \min\{X(\omega')_{t-1}, X(\omega')_t, X(\omega')_{t+1}\}, & X(\omega')_t < \eta. \end{cases} \quad (7)$$

Because phase is being ignored, υ is equal to some $X(\omega')$ which has the minimum magnitude across the three frames. Residual noise reduction works by taking advantage of the random nature of musical noise. It jumps between frequencies very quickly, so by observing adjacent frames, the musical noise will be at different frequencies. As long as the changes in the musical noise frequency are large enough to apply to different frequency bins and it does not overlap with the frequencies apparent in the clean signal, the minimum should be found and the musical noise reduced to a more consistent and less annoying hum. This decreases the previously mentioned risk of having large peaks and valleys in the spectrum, thus reducing the musical noise.

The residual noise reduction considerably reduced the amount of musical noise from the output when compared with the baseline implementation and thus will be considered for the final implementation.

3.5 Frame Length

Frame length has been set by default to 256 bins. An attempt has been made to increase the frame length to 512 as suggested to reduce musical noise, at the risk of having slurred-sounding speech. However, during testing, it has been shown that sound was heavily slurred, and that the higher size of the frame caused the CPU load to increase too much, causing skipped frames and crackling at the output.
For those reasons, the frame length was left unchanged for the final implementation.

3.6 Interval Length

In order to decrease processing time and to make the system faster responding to noise, the processing interval length has been decreased from the default value of 10 s to 5 s or 2.5 s. However, this caused increased music noise, voice distortion and crackling, and an overall heavily impacted noise reduction performance.
The processing interval was left unchanged for the final implementation.

4 Final Implementation

When testing all the enhancements, a test bench with switches that could be used in the CCS debugger was created. Only the code implementing the enhancements described in Section 3 is showcased here, the full code can be found in Appendix B.

In Figure 1, `fftframe` is the input signal after an FFT, `process_buf` points to the buffer in which processing will next occur and the other buffers store power and magnitude of the input signal before and after low pass filtering. Of note is that the `cabs` function is deconstructed here and the stages during that calculation are stored in an appropriate buffer. This is to save cycles later as some of the noise reduction formulae in Section 3.2 use these buffers to when calculating the output signal.

```
//magnitude filter
if (filtering == 1)
{
  for (k=0;k<FFTLEN;k++)
  {
    pow_buf[k] = fftframe[k].r*fftframe[k].r + fftframe[k].i*fftframe[k].i;
    mag_buf[k] = sqrtf(pow_buf[k]);
    lpmag_buf[k] = (1-filter_constant)*mag_buf[k] + filter_constant*lpmag_buf[k];
    lppow_buf[k] = lpmag_buf[k]*lpmag_buf[k];
  }
  process_buf = lpmag_buf;
}

//power filter
else if (filtering == 2)
{
  for (k=0;k<FFTLEN;k++)
  {
    pow_buf[k] = fftframe[k].r*fftframe[k].r + fftframe[k].i*fftframe[k].i;
    mag_buf[k] = sqrtf(pow_buf[k]);
    lppow_buf[k] = (1-filter_constant)*pow_buf[k] + filter_constant*lppow_buf[k];
    lpmag_buf[k] = sqrtf(lppow_buf[k]);
  }
  process_buf = lpmag_buf;
}

//no filtering *BASELINE*
else
{
  int temp = FFTLEN;
  for (k=0; k < temp; k++)
  {
    pow_buf[k] = fftframe[k].r*fftframe[k].r + fftframe[k].i*fftframe[k].i;
    mag_buf[k] = sqrtf(pow_buf[k]);
  }
  process_buf = mag_buf;
}
```

Figure 1: Low pass filter code

To implement the minimum noise buffer rotation, the code in Figure **??** is used. The buffers rotate when `frame_count` reaches 78 as with frame processing times of 32 ms and a buffer interval time of 2.5 s, 78.125

frames are needed per buffer interval, which is then rounded down to 78. When the buffers rotate, the new buffer being written to is overwritten by the spectrum of the current frame (line 18) to prevent any information from beyond the wanted time interval of 10 s impacting the speech enhancer. However, this means that the period of time from which the noise is estimated actually varies linearly between 7.5 s and 10 s.

```
1
2  if(frame_count >= 78)
3      {
4
5      /*rotates the minimum noise buffers*/
6         float *temp = m4;
7         m4 = m3;
8         m3 = m2;
9         m2 = m1;
10        m1 = temp;
11        frame_count = 0;
12
13        //shift buffer magnitude sums
14        mag4 = mag3;
15        mag3 = mag2;
16        mag2 = mag1;
17
18        //overwrites m1 with first frame in new interval
19        for (k=0;k<FFTLEN;k++)
20        {
21          m1[k] = process_buf[k];
22        }
23     }
24     //updates m1
25     else
26     {
27        for (k=0;k<FFTLEN;k++)
28        {
29          if (m1[k] > process_buf[k]) m1[k] = process_buf[k];
30        }
31     }
```

Figure 2: Buffer rotation and update code

Figure 3 shows how the minimum buffer is determined. First it calculates the magnitude sum of `m1` as it will very often change due to it updating per frame. The magnitude sums of the other buffers are fixed after the buffer is no longer being written to, and so are rotated along with the buffers themselves in Figure 2 to save clock cycles. The magnitude sums are then compared and `min_buf` then points to the minimum buffer. The power of that buffer is then calculated and stored in `noise_pow`.

```
//calculate m1 magnitude sum
mag1=0;
for (k=0;k<FFTLEN;k++)
{
  mag1 += m1[k];
}

//finds buffer with smallest magnitude sum, points min_buf to that buffer
if (mag1 < mag2 && mag1 < mag3 && mag1 < mag4) min_buf = m1;
else if (mag2 < mag3 && mag2 < mag4) min_buf = m2;
else if (mag3 < mag4) min_buf = m3;
else min_buf = m4;

//finds power buffer of that minimum noise buffer
for (k=0;k<FFTLEN;k++)
{
  noise_pow[k] = min_buf[k]*min_buf[k];
}
```

Figure 3: Minimum buffer code

The code in Figure 4 show cases how noise reduction formulae in 3.2 were switched between. Depending on `g_pow` and `g_omega`, `lambda_bot`, `lambda_top`, `signal_top` and `signal_bot` point to buffers for the appropriate formulae. In the case of λ not being multiplied by anything, `lambda_bot` and `lambda_top` point to a buffer filled with ones to keep the code readable. The reasoning behind this method becomes apparent in Figure 5.

```c
//points lambda and signal pointers to relevent buffers depending on g_omega and g_pow
if (g_omega == 1)
{
  lambda_top = min_buf;
  lambda_bot = mag_buf;
  if (g_pow)
  {
    signal_top = noise_pow;
    signal_bot = pow_buf;
  }
  else
  {
    signal_top = min_buf;
    signal_bot = mag_buf;
  }
}
```

Figure 4: Example of noise reduction code

Figure 5 shows how the noise reduction is implemented. By using a generic formula with `lambda_bot`, `lambda_top`, `signal_top` and `signal_bot`, the code is very readable despite whilst being able to switch between formulae. Of note is that when `residual` is on, the processed signal is also stored in `resframe0`.

```c
//calculates noise reduction in power domain
if (g_pow)
{
  for (k=0;k<FFTLEN;k++)
  {
    float temp = sqrtf(1 - alpha*(signal_top[k]/signal_bot[k]));
    float lambda_k = lambda*(lambda_top[k]/lambda_bot[k]);
    if (temp < lambda_k) temp = lambda_k;
    fftframe[k].r = fftframe[k].r*temp;
    fftframe[k].i = fftframe[k].i*temp;

    //puts processed signal in resframe0 and resframe0_mag
    if (residual)
    {
      resframe0[k] = fftframe[k];
      resframe0_mag[k] = cabs(fftframe[k]);
    }

  }
}

//calculates noise reduction in magnitude *BASELINE*
else
{
  for (k=0;k<FFTLEN;k++)
  {
    float temp = 1 - alpha*(signal_top[k]/signal_bot[k]);
    float lambda_k = lambda*(lambda_top[k]/lambda_bot[k]);
    if (temp < lambda_k) temp = lambda_k;
    fftframe[k].r = fftframe[k].r*temp;
    fftframe[k].i = fftframe[k].i*temp;

    //puts processed signal in resframe0 and resframe0_mag
    if (residual)
    {
      resframe0[k] = fftframe[k];
      resframe0_mag[k] = cabs(fftframe[k]);
    }
  }
}
```

Figure 5: Noise reduction code

Figure 6 shows how residual noise reduction is implemented. `resframe1` is the central frame, delayed by one frame with `resframe0` relatively one frame ahead and `resframe2` relatively one frame behind. The reasoning for this is described in Section 3.4.

```
//calculates and removes residual noise
  if (residual)
  {
    complex *temp = resframe2;
    float *temp_mag = resframe2_mag;
    for (k=0;k<FFTLEN;k++)
    {
      //checks frequency bin against threshold
      if (resframe1_mag[k] < res_thresh)
      {
        //finds minimum in adjacent frames
        if (resframe1_mag[k] < resframe0_mag[k] && resframe1_mag[k] < resframe2_mag[k]) fftframe[k] =
 resframe1[k];
        else if (resframe2_mag[k] < resframe0_mag[k]) fftframe[k] = resframe2[k];
      }
    }
    //rotate residual buffers
    resframe2 = resframe1;
    resframe1 = resframe0;
    resframe0 = temp;

    resframe2_mag = resframe1_mag;
    resframe1_mag = resframe0_mag;
    resframe0_mag = temp_mag;
  }
```

Figure 6: Residual noise reduction code

$$g(\omega) = \max(\lambda N(\omega), 1 - N(\omega)) \tag{8}$$

After testing using the selectable implementations, the method explained in the Berouti paper has been selected, with $\alpha = 5$, $\lambda = 0.08$, filter constant 0.7.

5 Testing

5.1 Method

Quantifying speech quality is difficult, as the perceived quality of speech can be based on many different parameters of the sound.
In order to test the speech enhancement system, two separate people would listen to the processed recording independently and then share their opinions in order to stay as objective as possible and avoid any biases. In case of conflicting opinions, judgements and justifications would be shared, and an agreement would be reached in order to match the listening test criteria as much as possible.
The design was tested against all provided sound files in order to have the widest selection of scenarios. The tested sound files were `clean`, `car1`, `factory1`, `factory2`, `lynx1`, `lynx2`, `phantom1`, `phantom2`, `phantom4`. Due to its low SNR, `phantom4` was not selected as a priority target for optimisation, as it was considered an edge case and optimisation for it caused degradation of quality in all other test cases.
During testing, a large amount of potential improvements has been rejected. Those improvements have been detailed in section 3.
Finally, an attempt to quantify the performance of the final selected design was done by recording the outputs and plotting the spectra of the different sound files.

5.2 Results

This section discusses a selected range of test results to show the system performance as well as its limits.

In the `clean` test case, the resulting spectrum was generated as expected. The spectral components of speech remained relatively unaffected, as showed by the similarity of vertical stripes in Figure 15. It can be observed that periods with no speech activity have been quieted down, even with the absence of original noise, the low background noise has been still attenuated.

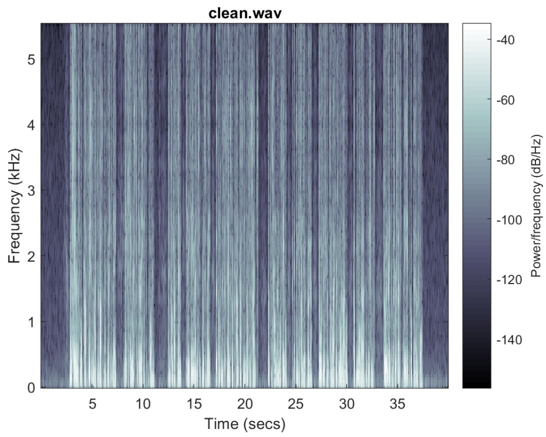

Figure 7: `clean` file, unprocessed

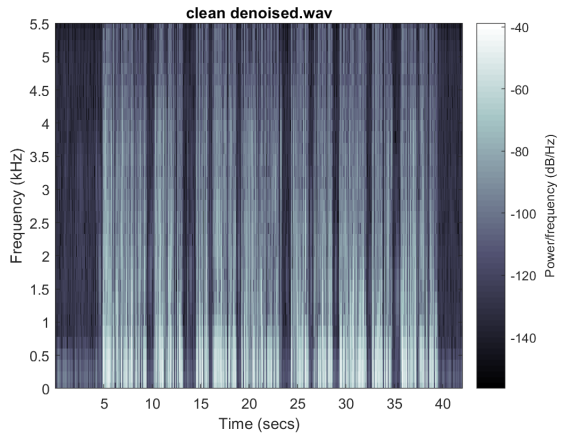

Figure 8: `clean` file, processed

The `factory1` test case shows the system operating in nominal conditions. The background noise has been mostly removed and periods of no voice activity can be observed on the diagram in Figure **??**. Initial noise can be seen before the 5 seconds mark, after which, the noise reduction algorithm starts correctly attenuate the noise.

On the same recording, lighter sections can be seen at high frequencies at the 12, 15 and 24 seconds mark. This may be caused by sudden sounds, such as shocks and hits in the factory background noise that the system did not fully attenuate as it relies on an average noise estimate. Using statistical methods to model for such noises might be a possible solution to this issue.

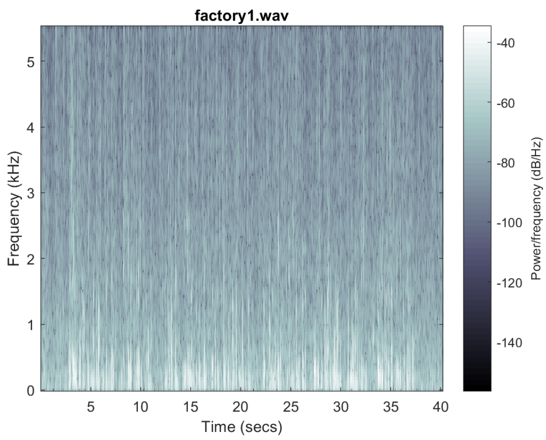

Figure 9: `factory1` file, unprocessed

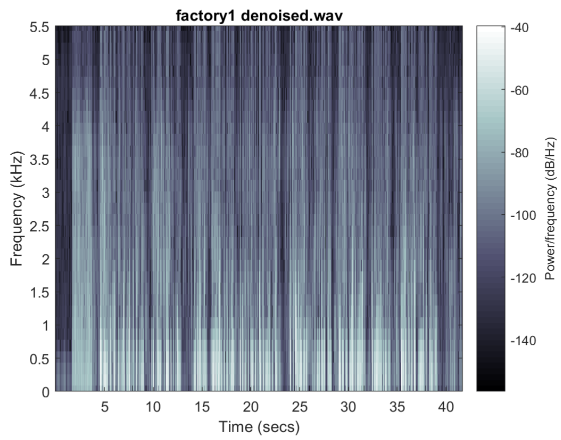

Figure 10: `factory1` file, processed

Finally, in the `phantom4` test case, a degraded performance can be observed. It can be seen on the unprocessed file in Figure 30, the low SNR of the original recording. Additionally, constant noise at harmonic frequencies can be seen, which propagates to the output of the speech enhancement system, as shown in Figure **??**. That periods of no speech activity still show audio activity on the output, as the system did not handle a very low SNR ratio.

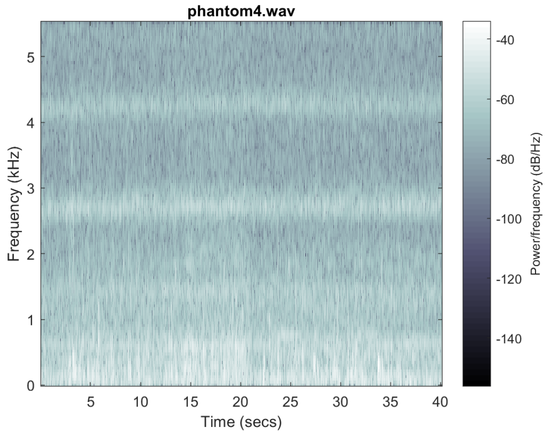

Figure 11: `phantom4` file, unprocessed

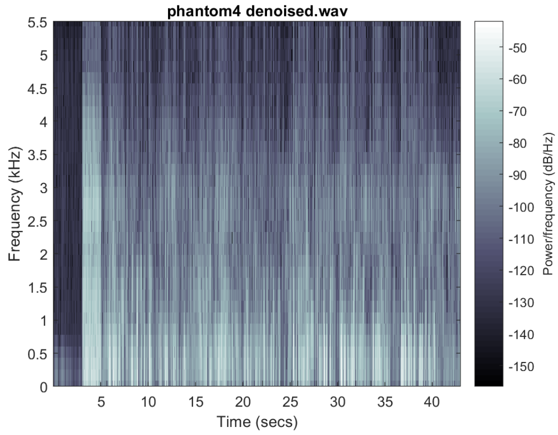

Figure 12: `phantom4` file, processed

However, in most cases the system outputted intelligible speech. Full test results are located in Appendix A.

5.3 Performance

Clock profiling of each enhancement was taken. When measuring the clock cycles, the numbers observed varied wildly so the results given in Figure 13 are approximate.

Enhancement	Cycles
Baseline	230000
Magnitude filtering	8000
Power filtering	47000
Residual	240000
Power Noise Reduction	70000

Figure 13: Approximate clock cycles of enhancements

From the results it becomes apparent why residual noise reduction would not work alongside the other enhancements. The number of clock cycles required to use it is considerably larger than other enhancements, and when used in conjunction with others would overrun the available CPU time.

6 Conclusion

The aim of the project was to implement a working speech enhancement system based on the spectral subtraction method and to implement, evaluate and select additional enhancements to provide optimal noise reduction performance while keeping high speech quality.
A functional system has been built, reducing background wideband noise. A compromise has been reached between speech quality, wideband noise subtraction, musical noise presence and processing time in order to meet optimum noise removal in most scenarios. Some edge cases, such as the low SNR signal in the `phantom4` test case, demonstrate the limitations of the chosen design. Possible additions could include processing time optimisation, implementation of oversubtraction methods for α adjustment or the addition of the omitted residual noise reduction for performance reasons and as a result these could improve noise removal process.

Appendices

A Results/Spectrograms

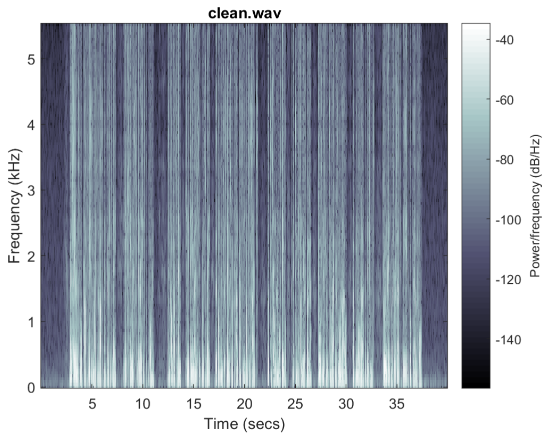

Figure 14: `clean` file, unprocessed

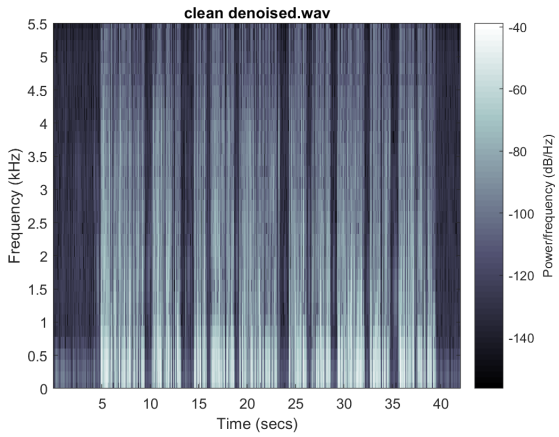

Figure 15: `clean` file, processed

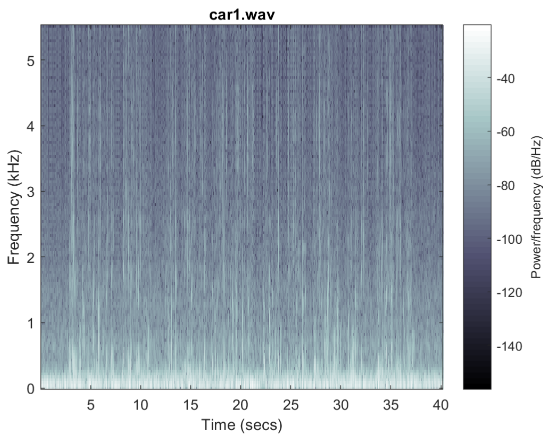

Figure 16: `car1` file, unprocessed

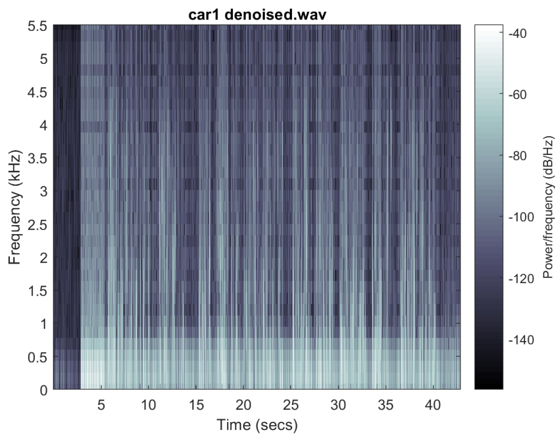

Figure 17: `car1` file, processed

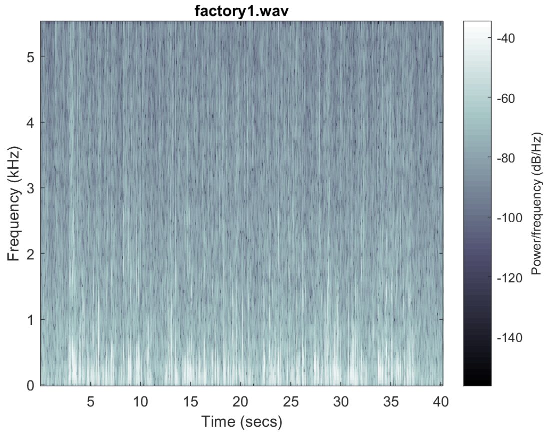

Figure 18: `factory1` file, unprocessed

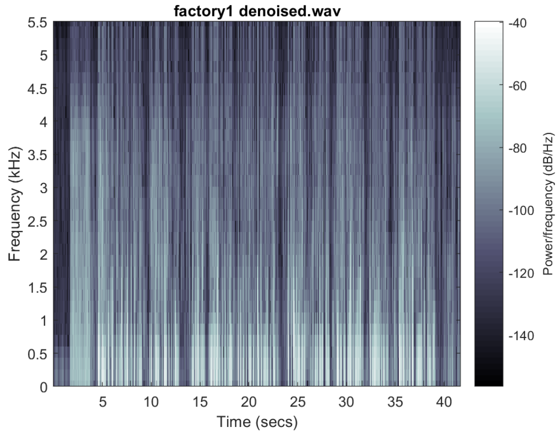

Figure 19: `factory1` file, processed

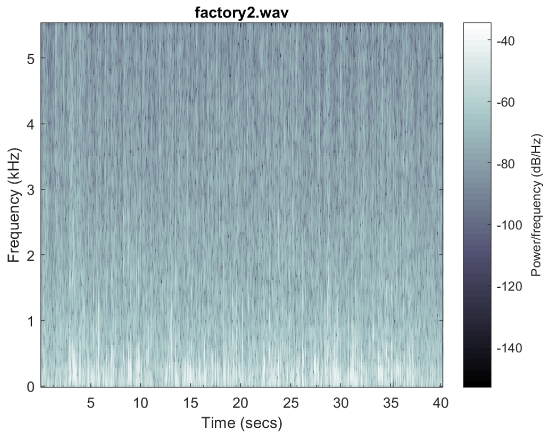

Figure 20: `factory2` file, unprocessed

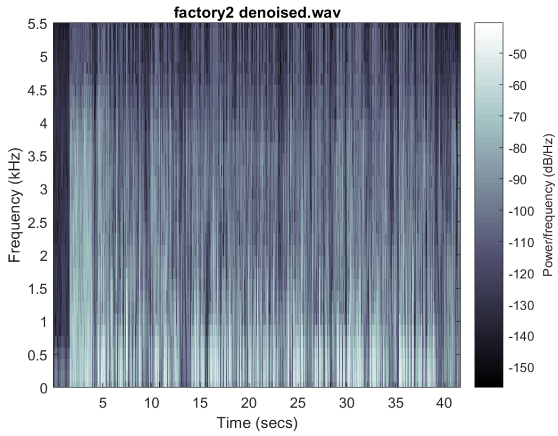

Figure 21: `factory2` file, processed

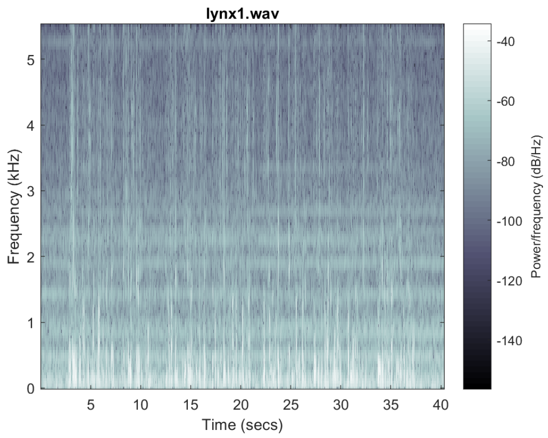

Figure 22: `lynx1` file, unprocessed

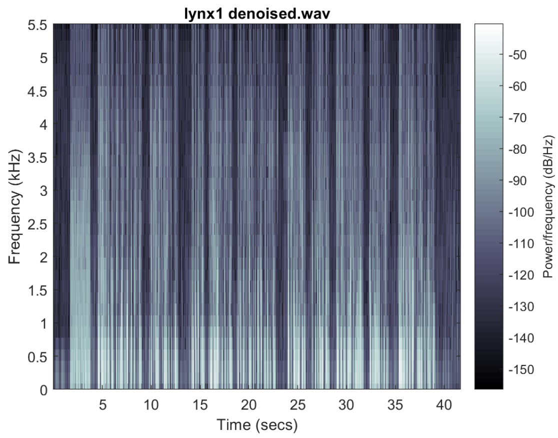

Figure 23: `lynx1` file, processed

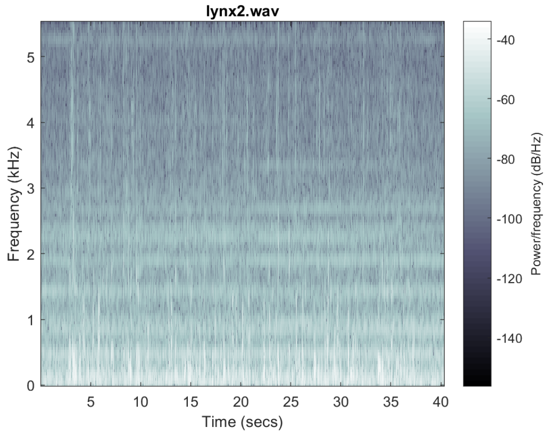

Figure 24: `lynx2` file, unprocessed

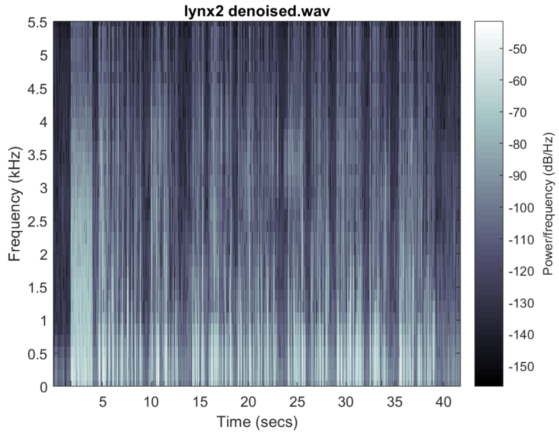

Figure 25: `lynx2` file, processed

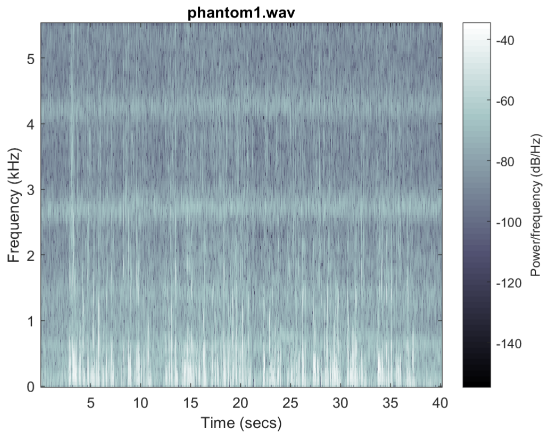

Figure 26: `phantom1` file, unprocessed

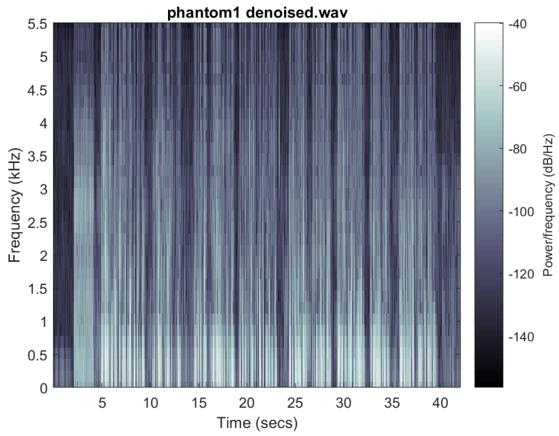

Figure 27: `phantom1` file, processed

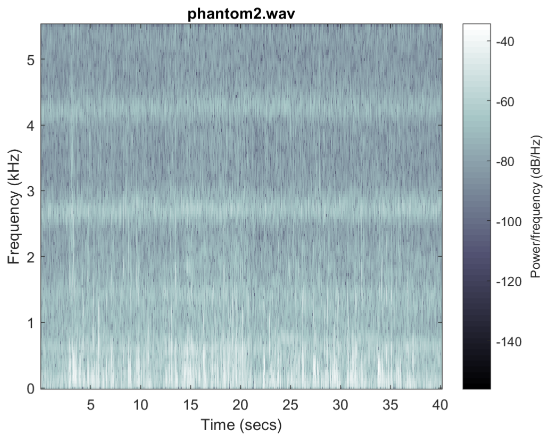

Figure 28: `phantom2` file, unprocessed

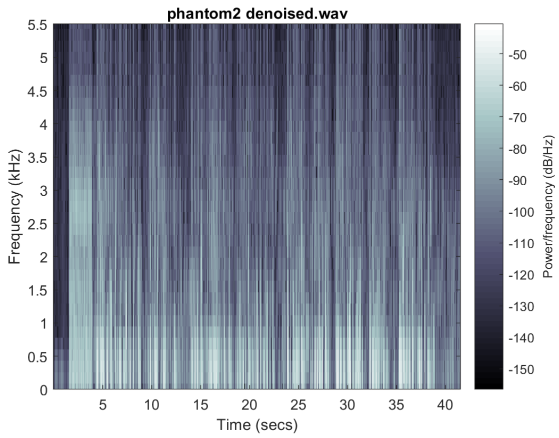

Figure 29: `phantom2` file, processed

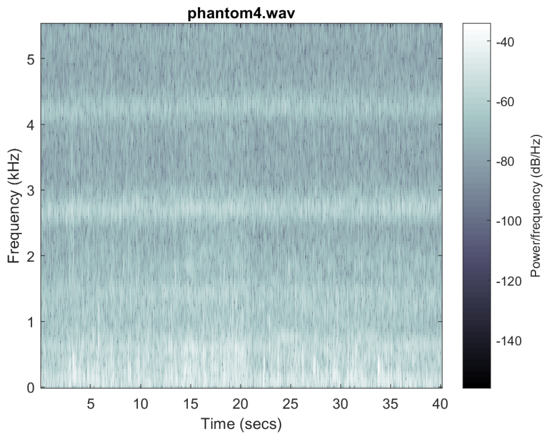

Figure 30: `phantom4` file, unprocessed

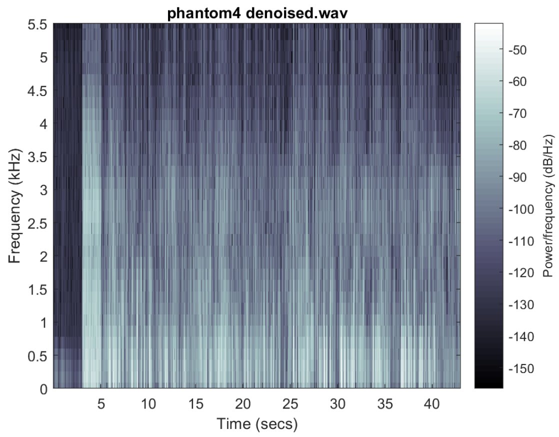

Figure 31: `phantom4` file, processed

B Code

```
enhance_a_new_hope.c

Type
Text
Size
16 KB (16,678 bytes)
Storage used
0 bytesOwned by undefined
Location
project
Owner
Tomasz Bia as
Modified
Mar 20, 2019 by Tomasz Bia as
Opened
6:57 PM by me
Created
Mar 20, 2019
Add a description
Viewers can download
/*************************************************************************************
             DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                      IMPERIAL COLLEGE LONDON

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

                      PROJECT: Frame Processing

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

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

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

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

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

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

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

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

```c
#define FRAMEINC (FFTLEN/OVERSAMP)  /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC) /* length of I/O buffers */

#define OUTGAIN 64000.0        /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)    /* Input gain for ADC  */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP        /* time between calculation of each frame */

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

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

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

float *inbuffer, *outbuffer;       /* Input/output circular buffers */
complex *fftframe;          /* processsing frame */
float *inwin, *outwin;                  /* Input and output windows */
float ingain, outgain;          /* ADC and DAC gains */
float cpufrac;              /* Fraction of CPU time used */
volatile int io_ptr=0;                  /* Input/ouput pointer for circular buffers */
volatile int frame_ptr=0;               /* Frame pointer */

float *m1, *m2, *m3, *m4;      /* Minimum noise intervals */
float mag1 = 0;             /* Magnitude sum of m1*/
float mag2 = 0;             /* Magnitude sum of m2*/
float mag3 = 0;             /* Magnitude sum of m3*/
float mag4 = 0;             /* Magnitude sum of m4*/

float *noise_pow;           /* Noise power for selected minimum buffer */
float *lpmag_buf;           /* Magnitude of input after low pass filter buffer*/
float *lppow_buf;           /* Power of input after low pass filter buffer*/
float *mag_buf;             /* Magnitude of input buffer*/
float *pow_buf;             /* Power of input buffer*/

complex *resframe0;           /* Stores processed signal one frame ahead of resframe1 */
complex *resframe1;           /* Stores processed signal to be output at a one frame delay from input */
complex *resframe2;           /* Stores processed signal one frame behind of resframe1 */

float *resframe0_mag;        /* magnitude buffer of resframe0 */
float *resframe1_mag;        /* magnitude buffer of resframe1 */
float *resframe2_mag;        /* magnitude buffer of resframe2 */


float *ones;             /* buffer of ones, used to switch easily between noise reductino formulae*/

float lambda = 0.08;
float alpha = 5.0;           /*scales noise reduction*/

float time_constant = 0.02;      /*time constant for low pass filter*/

float res_thresh = 0.8;        /*residual noise threshold*/

int frame_count = 0;         /*counter to see when an interval is over*/
int frames_per_interval;       /*number of frames when interval is over*/

float time_frame = 10.0;       /*time interval over which minimum noise is found*/
float filter_constant;         /*calculated within main, declared here if desired to change manually*/

/*----switches-----*/

int passthrough = 0;         /*allows signal through without processing*/
int filtering = 1;           /*switches filtering; 0 = no filtering, 1 = magnitude filtering, 2 = power filtering */
int g_omega = 5;           /*switches between noise reduction formulae*/
int g_pow = 0;             /*turns on noise reduction in power domain*/
int residual = 0;          /*turns on residual noise reduction*/


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

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

    int k; // used in various for loops

  filter_constant = 0.7;//expf(-TFRAME/time_constant);

/*  Initialize and zero fill arrays */

  inbuffer   = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
    outbuffer = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
  fftframe   = (complex *) calloc(FFTLEN, sizeof(complex));
    inwin     = (float *) calloc(FFTLEN, sizeof(float));  /* Input window */
    outwin     = (float *) calloc(FFTLEN, sizeof(float));  /* Output window */

  /*minimum noise buffers*/

```

```
    m1      = (float *) calloc(FFTLEN, sizeof(float));  /* noise buffer*/
    m2      = (float *) calloc(FFTLEN, sizeof(float));  /* noise buffer*/
    m3      = (float *) calloc(FFTLEN, sizeof(float));  /* noise buffer*/
    m4      = (float *) calloc(FFTLEN, sizeof(float));  /* noise buffer*/

  /*magnitude and power buffers of noise and signal at various stages*/

  lpmag_buf   = (float *) calloc(FFTLEN, sizeof(float));
  lppow_buf   = (float *) calloc(FFTLEN, sizeof(float));
  mag_buf   = (float *) calloc(FFTLEN, sizeof(float));  /* noise buffer*/
    pow_buf   = (float *) calloc(FFTLEN, sizeof(float));  /* noise buffer*/
    noise_pow    = (float *) calloc(FFTLEN, sizeof(float));

  /*residual noise reduction buffers*/

    resframe0    = (complex *) calloc(FFTLEN, sizeof(complex));
  resframe1    = (complex *) calloc(FFTLEN, sizeof(complex));
  resframe2 = (complex *) calloc(FFTLEN, sizeof(complex));

  resframe0_mag    = (float *) calloc(FFTLEN, sizeof(float));
  resframe1_mag    = (float *) calloc(FFTLEN, sizeof(float));
  resframe2_mag = (float *) calloc(FFTLEN, sizeof(float));

  /*array of ones used for g_omega and g_pow switching to be easier*/

  ones    = (float *) calloc(FFTLEN, sizeof(float));

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

    /* initialize hardware interrupts */
    init_HWI();

/* initialize algorithm constants */

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


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

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

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

  /* Function below sets the number of bits in word used by MSBSP (serial port) for
  receives from AIC23 (audio port). We are using a 32 bit packet containing two
  16 bit numbers hence 32BIT is set for  receive */
  MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

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

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


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

}

/******************************** process_frame() **********************************/
void process_frame(void)
{
  int k, m; //iteration variables
  int io_ptr0;
  float *min_buf; //points to minimum noise buffer
  float *process_buf; //points to buffer to be processed

  float *lambda_top;  //lambda multipplication numberator for g_omega and g_pow switching
  float *lambda_bot;  //lambda multipplication denominator for g_omega and g_pow switching
  float *signal_top;  //signal multipplication numberator for g_omega and g_pow switching
  float *signal_bot;  //signal multipplication numberator for g_omega and g_pow switching

  /* 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;

```

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

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

  /************************* DO PROCESSING OF FRAME  HERE *************************/
  frames_per_interval = (int)(time_frame/TFRAME);
  fft(FFTLEN, fftframe);

  //allows signal to pass through unprocessed
  if(!passthrough)
  {

    frame_count++;

    //magnitude filter
    if (filtering == 1)
    {
      for (k=0;k<FFTLEN;k++)
      {
        pow_buf[k] = fftframe[k].r*fftframe[k].r + fftframe[k].i*fftframe[k].i;
        mag_buf[k] = sqrtf(pow_buf[k]);
        lpmag_buf[k] = (1-filter_constant)*mag_buf[k] + filter_constant*lpmag_buf[k];
        lppow_buf[k] = lpmag_buf[k]*lpmag_buf[k];
      }
      process_buf = lpmag_buf;
    }

    //power filter
    else if (filtering == 2)
    {
      for (k=0;k<FFTLEN;k++)
      {
        pow_buf[k] = fftframe[k].r*fftframe[k].r + fftframe[k].i*fftframe[k].i;
        mag_buf[k] = sqrtf(pow_buf[k]);
        lppow_buf[k] = (1-filter_constant)*pow_buf[k] + filter_constant*lppow_buf[k];
        lpmag_buf[k] = sqrtf(lppow_buf[k]);
      }
      process_buf = lpmag_buf;
    }

    //no filtering *BASELINE*
    else
    {
      int temp = FFTLEN;
      for (k=0; k < temp; k++)
      {
        pow_buf[k] = fftframe[k].r*fftframe[k].r + fftframe[k].i*fftframe[k].i;
        mag_buf[k] = sqrtf(pow_buf[k]);
      }
      process_buf = mag_buf;
    }

    if(frame_count >= 78)
    {

    /*rotates the minimum noise buffers*/
      float *temp = m4;
     m4 = m3;
     m3 = m2;
     m2 = m1;
     m1 = temp;
     frame_count = 0;

     //shift buffer magnitude sums
     mag4 = mag3;
     mag3 = mag2;
     mag2 = mag1;

     //overwrites m1 with first frame in new interval
     for (k=0;k<FFTLEN;k++)
     {
       m1[k] = process_buf[k];
     }
    }
    //updates m1
    else
    {
      for (k=0;k<FFTLEN;k++)
      {
        if (m1[k] > process_buf[k]) m1[k] = process_buf[k];
      }
    }
    //calculate m1 magnitude sum
    mag1=0;
    for (k=0;k<FFTLEN;k++)
    {
     mag1 += m1[k];
    }

    //finds buffer with smallest magnitude sum, points min_buf to that buffer
    if (mag1 < mag2 && mag1 < mag3 && mag1 < mag4) min_buf = m1;
    else if (mag2 < mag3 && mag2 < mag4) min_buf = m2;
    else if (mag3 < mag4) min_buf = m3;
    else min_buf = m4;

    //finds power buffer of that minimum noise buffer
    for (k=0;k<FFTLEN;k++)
    {
      noise_pow[k] = min_buf[k]*min_buf[k];
    }

    //points lambda and signal pointers to relevent buffers depending on g_omega and g_pow
    if (g_omega == 1)
    {
     lambda_top = min_buf;
```

```
    lambda_bot = mag_buf;
    if (g_pow)
    {
      signal_top = noise_pow;
      signal_bot = pow_buf;
    }
    else
    {
      signal_top = min_buf;
      signal_bot = mag_buf;
    }
  }

  else if (g_omega == 2)
  {
    lambda_top = lpmag_buf;
    lambda_bot = mag_buf;
    if (g_pow)
    {
      signal_top = noise_pow;
      signal_bot = pow_buf;
    }
    else
    {
      signal_top = min_buf;
      signal_bot = mag_buf;
    }
  }

  else if (g_omega == 3)
  {
    lambda_top = min_buf;
    lambda_bot = lpmag_buf;
    if (g_pow)
    {
      signal_top = noise_pow;
      signal_bot = lppow_buf;
    }
    else
    {
      signal_top = min_buf;
      signal_bot = lpmag_buf;
    }
  }

  else if (g_omega == 4)
  {
    lambda_top = ones;
    lambda_bot = ones;
    if (g_pow)
    {
      signal_top = noise_pow;
      signal_bot = lppow_buf;
    }
    else
    {
      signal_top = min_buf;
      signal_bot = lpmag_buf;
    }
  }

  else if (g_omega == 5)
  {
    lambda_top = min_buf;
    lambda_bot = ones;
    if (g_pow)
    {
      signal_top = noise_pow;
      signal_bot = pow_buf;
    }
    else
    {
      signal_top = min_buf;
      signal_bot = mag_buf;
    }
  }
  //default baseline
  else
  {
    lambda_top = ones;
    lambda_bot = ones;
    if (g_pow)
    {
      signal_top = noise_pow;
      signal_bot = pow_buf;
    }
    else
    {
      signal_top = min_buf;
      signal_bot = mag_buf;
    }
  }

  //calculates noise reduction in power domain
  if (g_pow)
  {
    for (k=0;k<FFTLEN;k++)
    {
      float temp = sqrtf(1 - alpha*(signal_top[k]/signal_bot[k]));
      float lambda_k = lambda*(lambda_top[k]/lambda_bot[k]);
      if (temp < lambda_k) temp = lambda_k;
      fftframe[k].r = fftframe[k].r*temp;
      fftframe[k].i = fftframe[k].i*temp;

      //puts processed signal in resframe0 and resframe0_mag
      if (residual)
      {
        resframe0[k] = fftframe[k];
        resframe0_mag[k] = cabs(fftframe[k]);
      }

```

```c
            }
        }

        //calculates noise reduction in magnitude *BASELINE*
        else
        {
            for (k=0;k<FFTLEN;k++)
            {
                float temp = 1 - alpha*(signal_top[k]/signal_bot[k]);
                float lambda_k = lambda*(lambda_top[k]/lambda_bot[k]);
                if (temp < lambda_k) temp = lambda_k;
                fftframe[k].r = fftframe[k].r*temp;
                fftframe[k].i = fftframe[k].i*temp;

                //puts processed signal in resframe0 and resframe0_mag
                if (residual)
                {
                    resframe0[k] = fftframe[k];
                    resframe0_mag[k] = cabs(fftframe[k]);
                }
            }
        }

        //calculates and removes residual noise
        if (residual)
        {
            complex *temp = resframe2;
            float *temp_mag = resframe2_mag;
            for (k=0;k<FFTLEN;k++)
            {
                //checks frequency bin against threshold
                if (resframe1_mag[k] < res_thresh)
                {
                    //finds minimum in adjacent frames
                    if (resframe1_mag[k] < resframe0_mag[k] && resframe1_mag[k] < resframe2_mag[k]) fftframe[k] = resframe1[k];
                    else if (resframe2_mag[k] < resframe0_mag[k]) fftframe[k] = resframe2[k];
                }
            }
            //rotate residual buffers
            resframe2 = resframe1;
            resframe1 = resframe0;
            resframe0 = temp;

            resframe2_mag = resframe1_mag;
            resframe1_mag = resframe0_mag;
            resframe0_mag = temp_mag;
        }


    }
    ifft(FFTLEN, fftframe);

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

        /* 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]+fftframe[k].r*outwin[k];
        if (++m >= CIRCBUF) m=0; /* wrap if required */
    }
        for (;k<FFTLEN;k++)
    {
        outbuffer[m] = fftframe[k].r*outwin[k];    /* this loop over-writes outbuffer */
        m++;
    }
}
/*************************** INTERRUPT SERVICE ROUTINE ****************************/

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

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

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

    /* update io_ptr and check for buffer wraparound */

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

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

References

[1] M Berouti, Richard Schwartz and J Makhoul.
"Enhancement of speech corrupted by acoustic noise".
In: *[No source information available]* 4 (May 1979), pp. 208–211.
DOI: 10.1109/ICASSP.1979.1170788.

[2] Steven F. Boll. "Suppression of Acoustic Noise in Speech Using Spectral Subtraction". In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 27 (May 1979), pp. 113–120. DOI: 10.1109/TASSP.1979.1163209.