App development with audio applications from m-file to app

Daniel Jankovic

September 9, 2015

1 Part 1

Voice Activity Detector (VAD) is a technique used in signal processing to detect the presence of human voice in a signal. It can be an energy detector that indicates speech when the energy of the filtered signal exceeds a predefined threshold. Considered an important technology in speech based communication, today there are various types of applications that use it. Therefore a wide variety of VAD algorithms have been developed to provide the needed features.

There are different kind of stand-alone commercial baby monitors on the market today. From the most basic, that use one-way radio communication, to advance two-way communication monitors that use signal processing to transmit audio when a predefined threshold has been reached. It is also possible to find baby video monitors that broadcast both audio and video when the sensors notice movement. Since most of the monitor applications rely on radio signals to communicate between the units there is a probability that the signal will weaken or possibly not even reach the receiver because it needs to pass through multiple walls of varying thickness. The signal could also be effected by other applications. As the stand-alone monitor focuses on reliability (among other important sale strategies), little is known about the security features. It is possible to assume that the communication is uncrypted, at least in some products, and therefore introduces a potential risk for intrusion of peoples privacy.

To resolve the issues brought up above, an application such as the *Baby Activity Detector* (BAD) can can be made more portable, versatile and secure with the help of todays smart-phone technology and VAD. There are many VAD algorithms to choose from and they all have their strengths and weaknesses. Complex algorithms such as Linear Predictive coding (LPC), mel-frequency cepstrum (MFC) are very powerful but quite difficult to grasp and also to implement, they can be considered out of scope for this course. The following VAD algorithms are easy to implement and can be, when combined, quite robust for the task of a basic BAD. The simple short-time energy algorithm calculates the energy levels for each frame to detect voice, unvoiced or silenced regions. Voiced regions will have higher energy levels, however, the algorithm does not take unwanted noise into account which means that we can have false indication of voice detection. In order to remove the noise from the signal, spectral subtraction can be preformed. In the case of BAD, the threshold needs to be adjusted so that unforeseeable sound is not interpret as the infants cry. Zero-crossing rate (ZCR), is the rate at which a signal changes from plus to minus and back. The higher the rate the higher the frequency which indicates possible voice activity. According to [1] the cry sound that an infant makes has a fundamental frequency of 250-600 Hz (pitch). To be able to use ZCR together with the information above, it is necessary to extract the pitch from the signal in order to match the frequency interval.

The main task of BAD is to detect infant activity, an alternate algorithm is proposed in [1]. It describes an cry detection algorithm that is build up by three main stages. i) *VAD*, a statistical model-based detector [5] is used for detecting sections with sufficient audio acitivity. It also helps to reduce the power consumption. ii) *Classification*, uses k-nearest neighbours (k-NN) algorithm [6] to label each frame as either 'cry' (1), close enough, or 'no cry' (0). iii) Post-processing, validation stage in order to reduce false-negative errors. The idea of having devoted algorithm to detect infant cry is a winning concept for a BAD application, according to the authors it even had promising results in low SNR. Despite simplicity of the algorithm many of the features required to implement were mentioned earlier to be out of scope for this project.

An algorithm that might be of interest [3] suggests a new approach to speech enhancement, without the help of VAD technology. The signal is divided into multiple sub-bands and an noise floor level estimate is calculated simultaneously as the short-time average. The goal is to boost the sub-bands with high Signal-to-Noise Ration (SNR) instead of to suppress the lower. This algorithm has great potential to reduce the noise levels when analyzing incoming signals to the BAD application.

A quick search on the net gives significant amount of hits for smart-phone based BAD applications. The techniques vary, from bluetooth to Wi-Fi and 3-/4G solutions. The award winning application *Baby Monitor 3G* [7] is a feature rich cross platform application that solves most the of issues brought up in this report. It supports both Wi-Fi and 3G/LTE networks, ability to transfer high quality live video, adjustable microphone sensitivity, talk-back functionality and guarantees both reliability and privacy. It can be assumed that the Android based BAD application *Dormi* [8] offers similar features as Baby Monitor 3G, even if Baby Monitor 3G's feature list provides barely any deeper information. It is noteworthy that *Domri* have both *Smart noise detection* and *Adaptive audio enhancement* as sales pitch, which from a engineering point of view is very attractive.

Following is a purposed algorithm for an BAD application

```
while(true){
    % Record audio can place it into the register %
    Get frame from the register
    Divide the signal into different sub-bands with FFT
    Calculate the total short-time energy average
    Calculate noise level for each sub-bands
    if energy average above threshold
        Calculate the gain for each sub-band
        Boost the sub-band with high SNR
        % Extract fundamental frequency (pitch)
        Count the ZCR under 1 seconds
            if the ZCR is within 250-600 Hz
                Possible infant activity detected!
                (Occurs only first time, and needs to be reseted)
% Send frames %
                Start broadcasting the sound to the receiver
            end
    else
        Reset ZCR
        Dismiss and get next frame from register
    end
}
```

2 Part 2

Three various baby and noise sounds were provided by the institute. The sample frequency for all of the recorded files are 8 kHz. In Figure 1 and Figure 3 the frequency spectrum is plotted for the baby respective noise sounds. Because the energy levels are unproportionally low for two out of the three audio recordings, an enhanced plot is displayed in Figure 2. From the plots it can be seen that the baby freqency interval is between $\sim$ 300-2500 Hz, including cry and talk, and the noise frequency interval is between $\sim$0-200 Hz and $\sim$1300-2100 Hz. One way to work with only the desired frequencies is to create a filter with a bandpass between 300-1300 Hz.

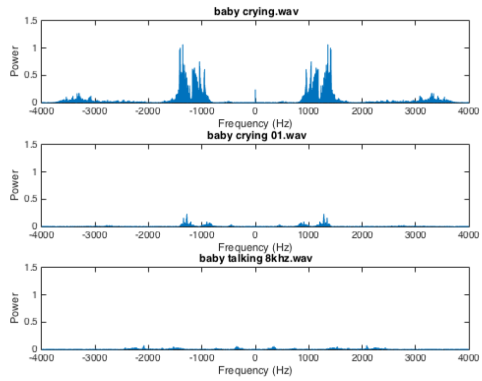

Figure 1: Frequency spectrum for baby sound

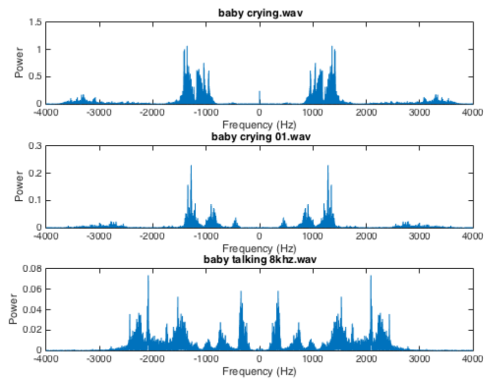

Figure 2: Frequency spectrum for baby sound, scaled y-axis

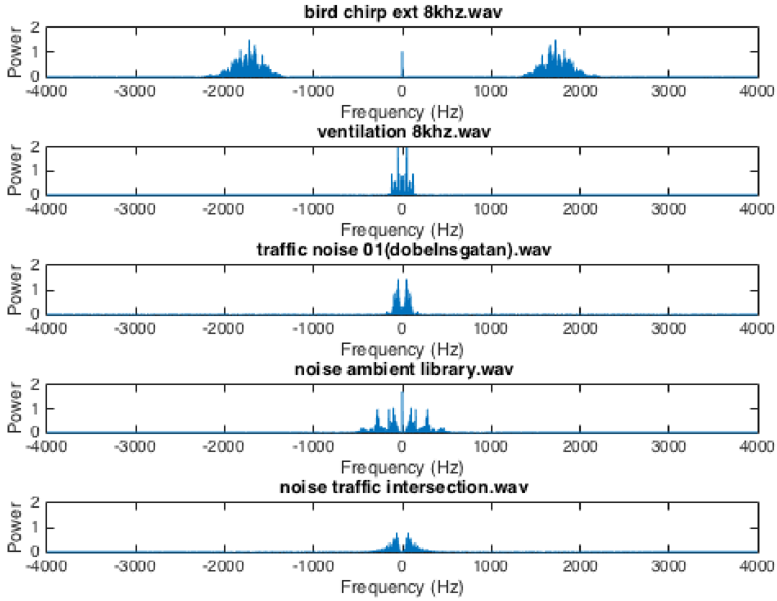

Figure 3: Frequency spectrum for noise files

2.1 The algorithms

A quick and simple algorithm for a BAD application is preferably an algorithm that measures the energy from the input sound. Another benefit of using this technique is that the implementation in Android OS may be easier for a novice application developer. Mathematically described, the power equation, which is close relationship to energy, is given by the following formula:

$$P(n) = \frac{1}{N} \sum_{k=0}^{N-1} x^2(n-k)$$

Since the BAD application will be performing these power calculations of the input sound in real-time, it is undoubtedly impossible to implement the equation above. A solution to the problem is to use the *recursive averaging* algorithm. It is small and hardware friendly algorithm that calculates the power of a given input without using too much memory. Given the equation below,

$$P(n) = \alpha P(n-1) + (1-\alpha)x^2(n)$$

instead of performing the calculation for one sample at the time, a predefined number of samples in blocks, referred to as frames, are squared and summed.

H

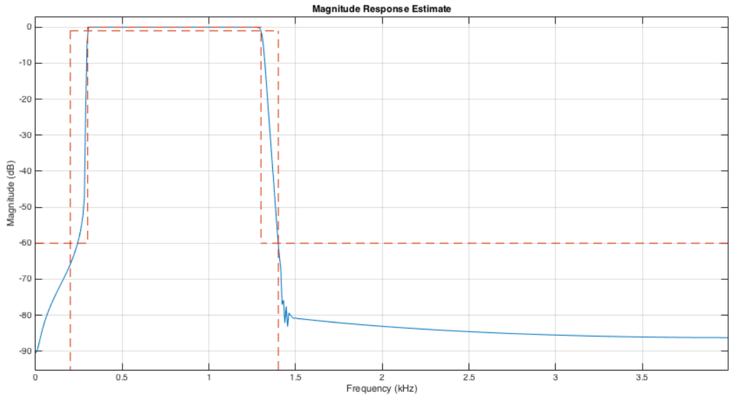

Figure 4: Butterworth 300-1300 Hz bandpass filter

Each frame represents $x^2(n)$. Every result, $P(n)$, is saved to be used as $P(n-1)$. The α is a constant between 0 and 1 and is related to the following formula:

$$\alpha = \frac{1}{T_s F_s}$$

Despite that the α can be derived from the equation, to get the better results further tweaking and testing is required. The α value used in this research, 0.5, suggests that the new $P(n)$ is equally weighted between old and new calculations. The MATLAB code for the simple algorithm can be found in Appendix B.

The advanced algorithm is based upon the same, *recursive averaging*, algorithm but before calculating the power of the input sound, the signal is first filtered through a Butterworth bandpass filter to remove unwanted frequencies. Butterworth filter, because is gives the least amount of ripple in the bandpass. By having the noise frequencies suppressed, more precision is acquired for finding baby activity. The MATLAB code for the filter can be found in Appendix A.

2.2 Evaluation and performance

To evaluate and test the performance of the simple and advance algorithm new sound files were created, the three provided baby sound files created the based. The goal was to make the alarm go off with the sound that the baby caused, in clean and noisy configurations. For each baby sound file, three new sound files were created to mislead the algorithms. The MATLAB code for this can be found in Appendix C. Giving three sets of 4 different test configurations, each set contained the following files:

1. clean baby sound without any noise

2. simulate early mornings, bird and ventilation noise added to the base sound

3. simulate daily environment, all noise files added to the base sound

4. simulate *extremely* noisy environment, all noise files amplified and added to the base sound

As can be seen from Figure 5, the green horizontal line indicates the threshold value. The red circle indicates where the alarm has been set off, when the algorithm has notified the user of baby activity. If the algorithm failed to set off the alarm, the red circle is placed in the origo. In this test environment the alarm was set off after five frames successfully breached the threshold value.

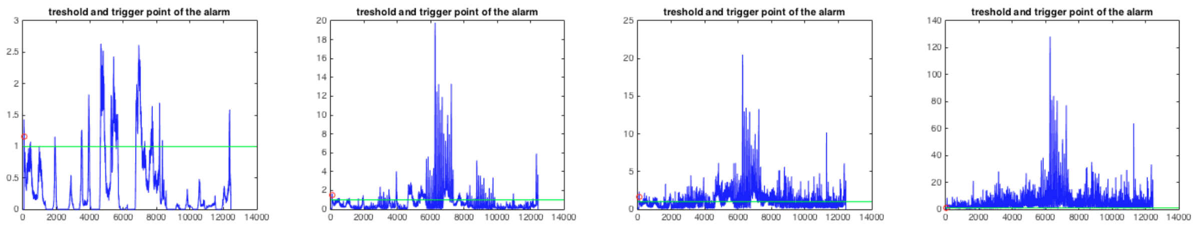

Figure 5: Baby crying.wav, simple algorithm

The test results from the *Baby crying.wav* set, Figure 6, show that the alarm was set off in approximate same time in every configuration and the reason for this could be that there is a high energy baby sound in the beginning of all of the files. The *clean* and the *bird and ventilation* configurations can be considered as plausibly successful results since the noise levels are not interfering as much as the two, to the right, configurations.

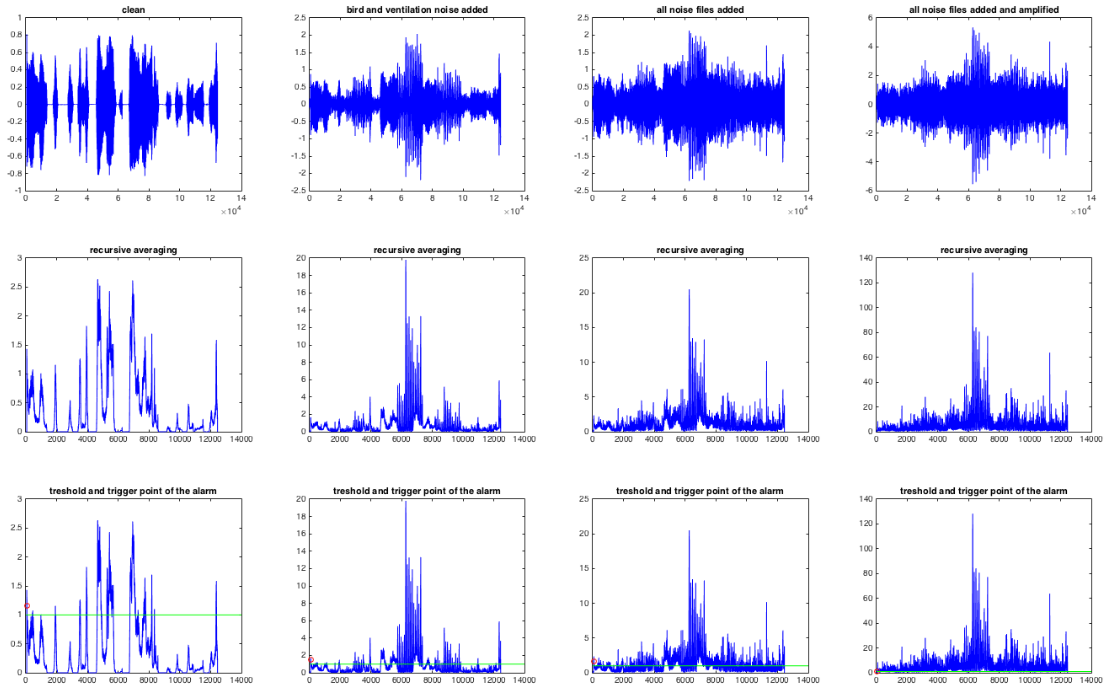

Figure 6: Baby crying.wav, simple algorithm

Both *Baby crying1.wav* and *Baby talking.wav* gave unsatisfactory results without any amplification, as can be seen in Figure 7 and Figure 8. The success of setting of the alarm in the other configurations were due to the noise.. Not remotely close to trigger the alarm in clean configuration, illustrated be the left most plots, the test were performed a second time with amplified baby sound.

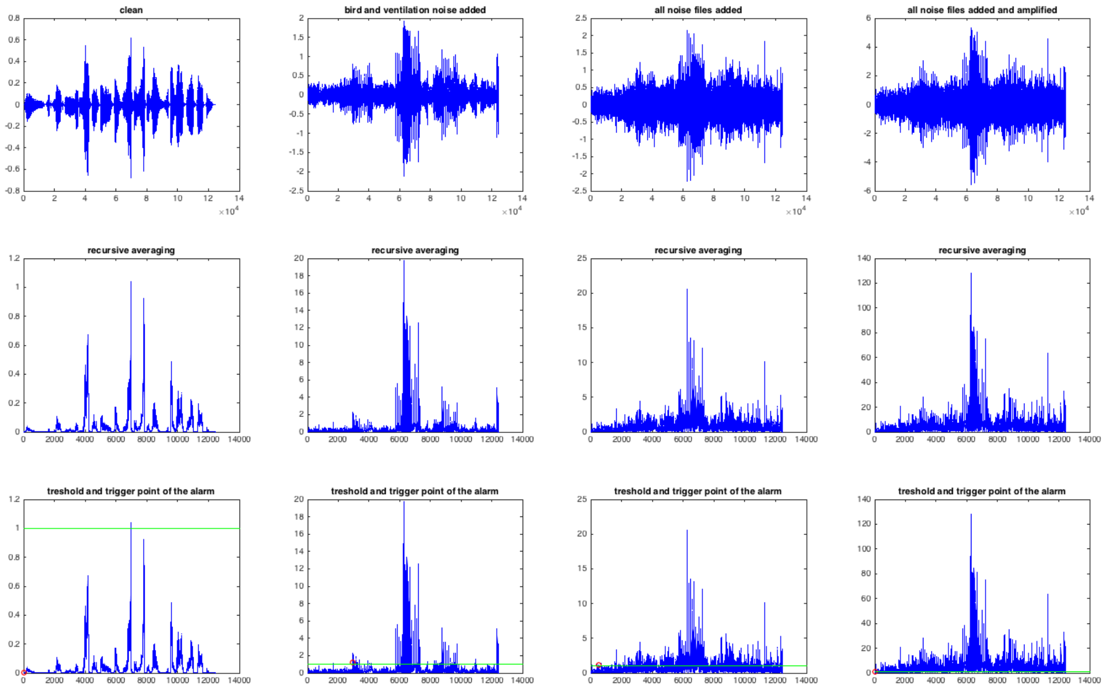

Figure 7: Baby crying1.wav, simple algorithm

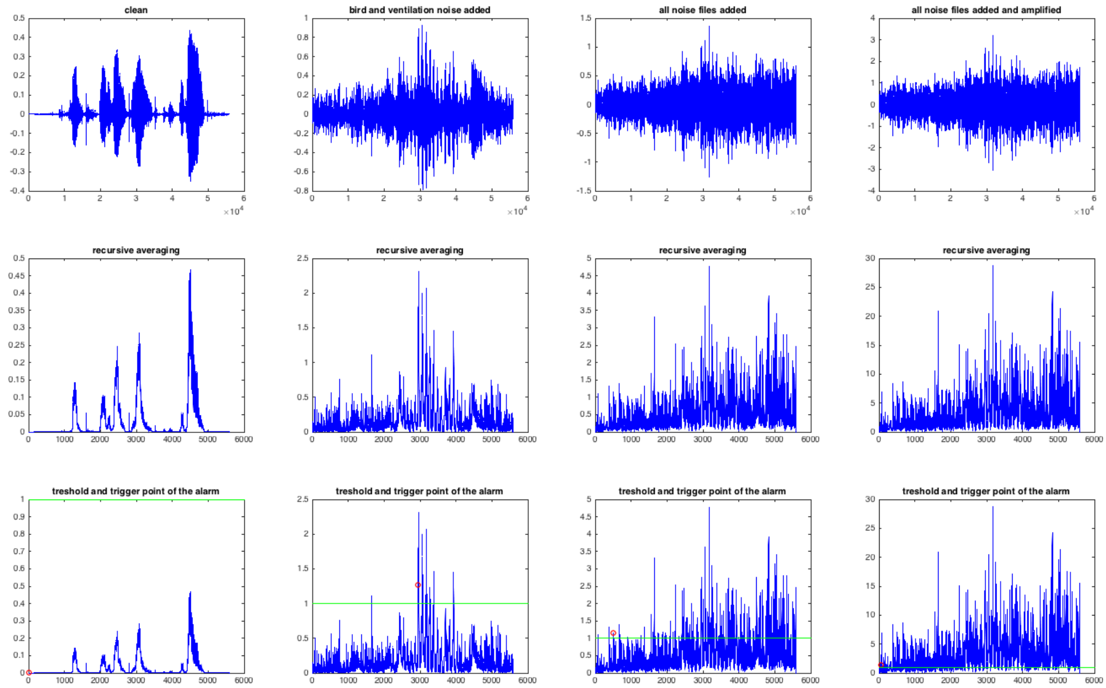

Figure 8: Baby talking.wav, simple algorithm

By having the baby sounds amplified before the noise is added, making sure that the alarm sets off in clean configuration, gives a possibility for the baby sounds to trigger the alarm in other configurations as well, see Figure 9 and Figure 10. Unfortunately, the simple algorithm performed poorly for the *Baby crying1.wav* set, Figure 9, but in the *Baby talking.wav* set, Figure 10, the first two configurations, *clean* and *bird and ventilation*, performed surprisingly well.

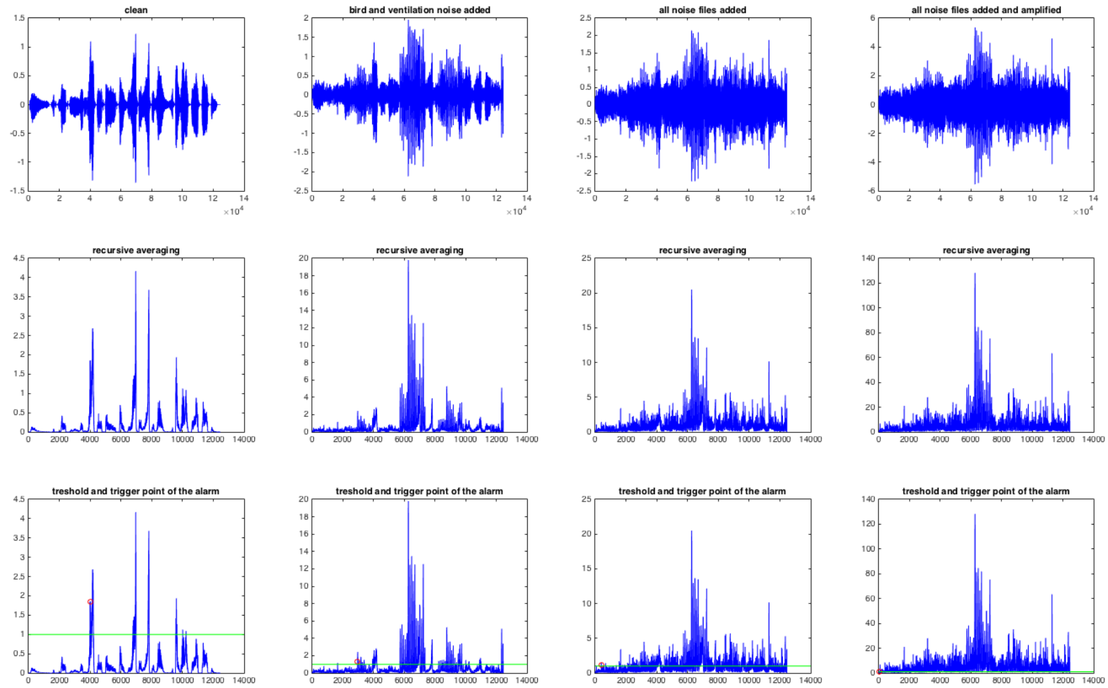

Figure 9: Amplified Baby crying1.wav, simple algorithm

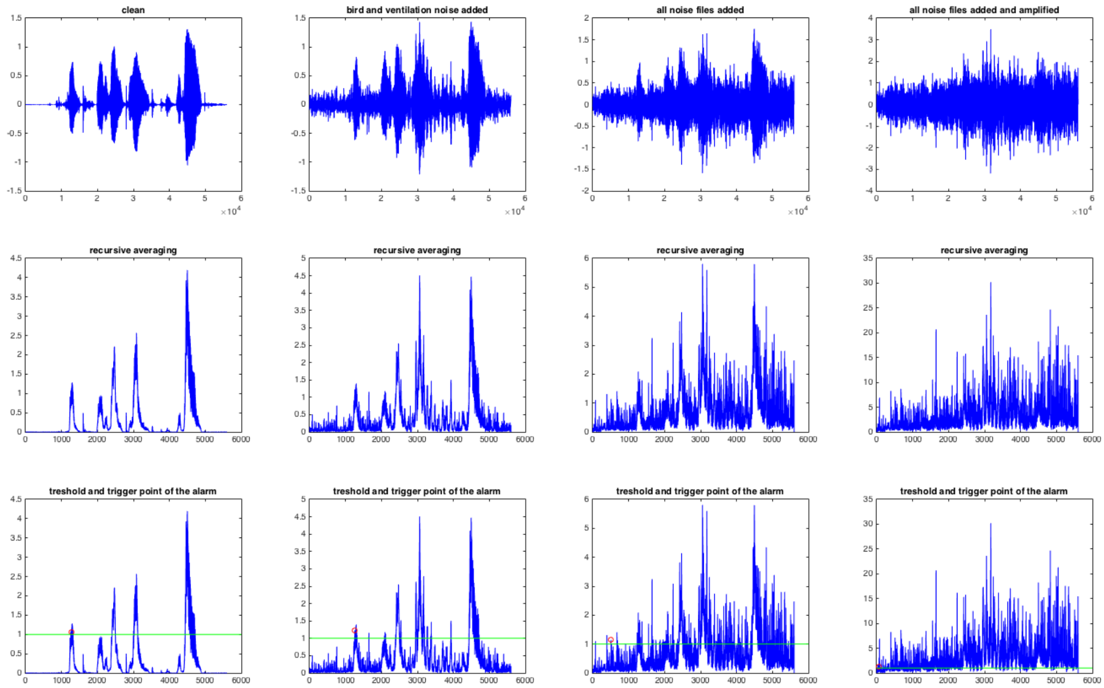

Figure 10: Amplified Baby talking.wav, simple algorithm

The advanced algorithm performed, as expected, exceptionally well due to unwanted frequencies being suppressed, not interfering with the power calculations. Figure 11, Figure 12 and Figure 13 show great results, except for the early trigger in the last configuration of the *Baby talking.wav* set. The alarm was set off earlier because the signal was infected with noise. Despite the early trigger of the alarm, it was executed during baby activity and not by the added noise. The alarm could have been triggered during the same time in other configurations of that set as well.

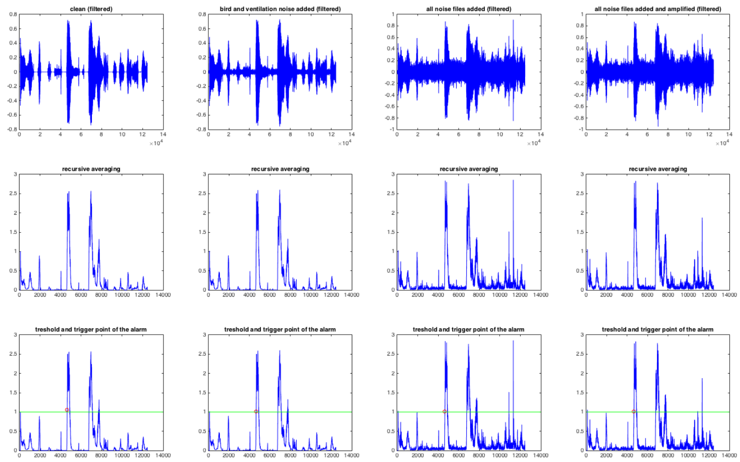

Figure 11: Baby crying.wav, advanced algorithm

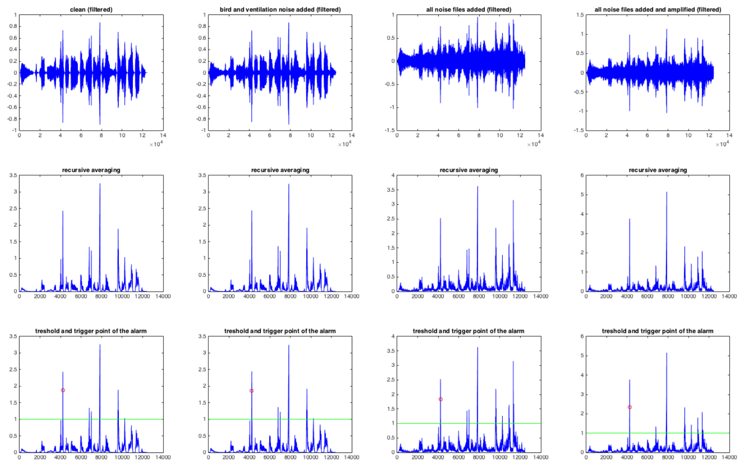

Figure 12: Amplified Baby crying1.wav, advanced algorithm

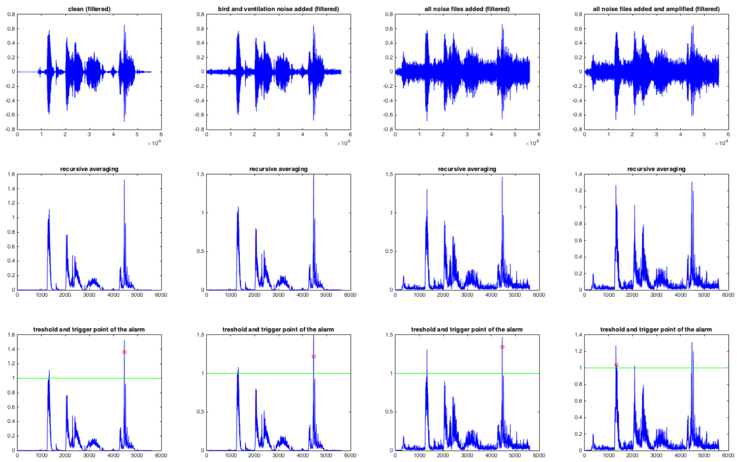

Figure 13: Amplified Baby talking.wav, advanced algorithm

2.3 Conclusion

Giving the results from the two different algorithms, it is fair to state the simple algorithm was unreliable since the noise levels could trigger the alarm. If the design goal is accuracy, reliability and good performance for the BAD application then implementing the advanced algorithm would be the best option. However, both algorithms rely on the *recursive averaging* algorithm. Implementing it must be done either way and should this be done within short amount of time, an attempt should be preformed to implement the Butterworth bandpass filter. If the filter issue can not be overcome with built-in Java classes, third-party libraries, or guidelines from the institute then the advanced algorithm will not be implemented. This should be considered as bonus task for Part 3 of the course.

3 Part 3 (4)

The last section of this report, *Part 3*, actually represents *Part 4* in the "step-by-step" instructions provided from the institute.

As mentioned in *Part 2*, the *recursive averaging* algorithm is crucial for both the simple and advanced algorithm. An comprehensive study of the framework was preformed before the implementation of the simple algorithm started.

3.1 StartFragment.java

The first modifications were performed in *startFragment*'s, *run()* method.

```
public void run()
{
    ...
    try
    {
        int N = AudioRecord.getMinBufferSize(8000,
                AudioFormat.CHANNEL_IN_MONO,
                AudioFormat.ENCODING_PCM_16BIT); /*N:640*/
        recorder = new AudioRecord(AudioSource.MIC, 8000,
                AudioFormat.CHANNEL_IN_MONO,
                AudioFormat.ENCODING_PCM_16BIT, N*10);

        recorder.getState();
        recorder.startRecording();

        /*
         * Loops until something outside of this
         * thread stops it. Reads the data from the
         * recorder and calculates an average to use
         * in the baby detector.
         */
        while(!stopped)
        {
            //was 160 now 10
            short[] buffer =  new short[10];
            N = recorder.read(buffer,0,buffer.length);
            double sum = 0;
            for (short val : buffer) {
                sum = sum + Math.abs(val);
            }

            /**
             * To scale it down, dive by buffer length
             */
            double average = sum / buffer.length;
            recursiveSum =
                (ALPHA * oldRecursiveSum) + ((1-ALPHA)*average);
            oldRecursiveSum = recursiveSum;


            final BabyState babyState =
                detector.updateState(recursiveSum);
```

```
            ...
}
```

To resemble the MATLAB implementation, the buffer length was decreased from 160 to 10. Dividing the *sum* with *buffer.length* was kept to scale down the new *P(n)*. After calculating the recursive averaging, the *recursiveSum* is sent to *updateStates()* as parameter.

3.2 StudentDetector.java

The Java class *TestDetector*, which implemented the *BabyDetector* interface, was renamed to *StudentDetector*. Since the class is more complex than the one above, more thorough explanation is provided for each method.

```
public class StudentDetector implements BabyDetector {

    // Keeps track of the values
    private short currentValue = 0;
    private int frameCounter = 0;

    // Initializing
    private boolean init = false;
    private boolean senseChange = false;
    private long sum = 0;
    private int procentage = 0;
    private int multiply;

    // Constants
    private final int maxFrames = 10000;
    private final int framesThreshold = 20;
    private final int MAX_ENERGY_CEILING = 6000;


    // Holds the values for the measurements of detector
    private short threshold = 0;
    private short baseline = 0;

...
```

The most important, in the code above, are the three constants *maxFrames*, *framesThreshold*, and *MAX ENERGY CEILING*.

- *maxFrames*

  Defines how many frames should be used to average the noise level prior to start of the main application.

- *framesThreshold*

  Defines how many frames need to breach the threshold value before setting off the alarm.

- *MAX ENERGY CEILING*

Despite the questionable variable name, this defines the highest possible threshold value.

The *MAX ENERGY CEILING* and *framesThreshold* constant were selected after evaluating the energy levels in different test environments. *framesThreshold* was given an higher value, compared to the MATLAB implementation, because it lowered the probability of false alarms. Since the application is interested in the background noise levels when the caretaker is not present in the room. Fairly long initialization time was implemented, hence why *maxFrames* was set to a high value.

```
...
@Override
public BabyState updateState(double average) {
    BabyState state = null;
    currentValue = (short) average;
    if (!init) {
        if (frameCounter < maxFrames) {
            sum += currentValue;
            frameCounter++;
            state = BabyState.SLEEPING;
        } else {
            sum = sum / maxFrames;
            baseline = (short) sum;
            init = true;
            multiply = (MAX_ENERGY_CEILING / baseline);
            threshold = getThreshold(baseline, procentage);
            frameCounter = 0;
            state = BabyState.NOISE;
        }
    } else if (init) {
        if (senseChange) {
            threshold = getThreshold(baseline, procentage);
        }
        if (frameCounter > framesThreshold) {
            frameCounter = 0;
            state = BabyState.AWAKE;
        } else if (currentValue > threshold) {
            frameCounter++;
            state = BabyState.NOISE;
        } else {
            if (frameCounter > 0) {
                frameCounter--;
                state = BabyState.NOISE;
            }

        }
    }
    return state;
}
...
```

updateState() method will be called from *StartFramgment* class and since the user has selected the student detector, the *StudentDetector* inherited method will be called. The input parameter for this method is the average sum, *recursiveSum*. The code is divided into two parts, the upper if-statement, that takes the boolean value *init*. When $init == false$, the background noise levels is averaged and a threshold value is configured. This is considered to be performed when starting the application, which is why $init = true$ when initialization is finished.

The lower part, when $init == true$, is the main activity detector that validates if the *currentValue* is bigger than threshold. However, if the sensitiivty bar has been adjusted ($senseChange == true$), a new scaled threshold value will be returned from *getThreshold()*.

```
...
/**
 * getThreshold scales the threshold in relation to the
 * MAX_ENERGY_CEILING
 * @param baseline the calculated reference point
 * @param procentage the sensitivity bar's value
 * @return returns the new threshold scaled to the
 * sensitivity level choosen by the user
 */
private short getThreshold(short baseline, int procentage) {
    short temp;
    switch (procentage) {
        case 0:
            temp = (short) (baseline * multiply);
            break;
        case 100:
            temp = (short) (baseline * (multiply*0.001));
            break;
        default:
            double denominate = (double) procentage/100;
            temp = (short) (baseline * (multiply *
                        (double)(1 - denominate)));
            break;
    }
    senseChange = false;
    return temp;
}
...
```

The *getThreshold()* method calculates the threshold value given the baseline and percentage value. Initially, the sensitivity bar will be 0 and the *case* 0 will be *MAX ENERGY CEILING* since *multiply* is extracted from

$$\frac{MAX\ ENERGY\ CEILING}{baseline}$$

The *case* 100 is suppose to return the baseline level. However, this is considered too sensitive. Instead a small fraction is added to baseline before it is returned. In the $default\ case$ a proper estimation is calculated in order to match the scaling set by the user.

```
...
@Override
public View getConfigurationView(LayoutInflater inflater) {
    View configView =
        inflater.inflate(R.layout.simple_student_detector, null);
    final TextView amplitudeLabel =
        (TextView) configView.findViewById(R.id.sensValue);

    SeekBar bar = (SeekBar) configView.findViewById(R.id.senseBar);
    bar.setProgress(0);
    bar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {

    ...

        @Override
        public void onProgressChanged(SeekBar seekBar,
            int progress,
            boolean fromUser) {

            amplitudeLabel.setText(progress + "%");
            procentage = progress;
            senseChange = true;
        }
    });
    amplitudeLabel.setText(0 + "%");
    return configView;
}
...
```

The inherited method *getConfigurationView* is inspired by the two provided classes that also implement *BabyDetector* interface. When the user adjusts the sensitivity bar, the flag *senseChange* will be set to *true* and threshold will receive a new value.

3.3 Advanced algorithm

After completing the implementation of the simple algorithm, attempts were made on implementing the advanced algorithm. However, time being a factor, the poorly documented third party DSP.jar packaged required more time and more adjustments in the framework to be able to work. The decision was made not to proceed with implementing the advance algorithm.

4 Appendix

4.1 A *butterFilter.m*

```
% Values for the bandpass
A_stop1 = 50;
F_stop1 = 200;
F_pass1 = 300;
F_pass2 = 1300;
F_stop2 = 1400;
A_stop2 = 50;
A_pass = 1;
Fs = 8000;
filtering = 1;

% Design and create Bandpass filter
BandPassSpecObj = fdesign.bandpass(F_stop1, F_pass1, ...
    F_pass2, F_stop2, A_stop1,A_pass,A_stop2, Fs);
BandPassFilt = design(BandPassSpecObj,'butter');

% Graphical representaion of the bandpass filter
% Uncomment for usage, requires DSP toolbox for MATLAB
fvtool(BandPassFilt)
```

4.2 B *recursiveAverg.m*

```
function [ P ] = recursiveAverg( buffer, alpha )
%   SIMPLE_RECURSIVEAVERG function calculates reverse averaging
%   given a squared buffer. The buffer is squared and the frames
%   are summed up and placed in a cell in vec-array. Each cell
%   represent a frame and is later used in the recursive averaging
%   algorithm. The alpha value is 0.5.
%
%   Input: buffSquared, alpha
%   Output: P-array with recursive averaging power calculations

    % The buffer-matrix is squared and summed to fit in
    % vec-array
    n=size(buffer,2);
    buffSquared=buffer.^2;
    for i=1:n
        vec(i,1)=sum(buffSquared(1:end,i));
    end

    % The vec-array is used to perform recursive averaging, which
    % is stored in P-array
    rows = size(vec,1);
    P=zeros(rows,1);
    P(1,1)=0;
    for i=2:size(vec,1)-1
        P(i,1)=alpha*P(i-1,1) + (1-alpha)*vec(i,1);
    end
end
```

4.3 C *makeSound.m*

```
% Clear and close all
close all
clear all

% Read all audio files and extract fs
% Baby sounds
baby_crying_1 = ('baby_signals/baby-crying.wav');
[x_baby_crying_1, fs_baby_crying_1] = audioread(baby_crying_1);
baby_crying_2 = ('baby_signals/baby-crying-01.wav');
[x_baby_crying_2, fs_baby_crying_2] = audioread(baby_crying_2);
baby_talking = ('baby_signals/baby-talking_8khz.wav');
[x_baby_talking, fs_baby_talking] = audioread(baby_talking);
% Noise sounds
noise_bird = ('noise_signals/bird_chirp_ext_8khz.wav');
[x_noise_bird, fs_noise_bird] = audioread(noise_bird);
noise_traffic = ('noise_signals/traffic-noise-01(dobelnsgatan).wav');
[x_noise_traffic, fs_noise_traffic] = audioread(noise_traffic);
noise_ventilation = ('noise_signals/ventilation_8khz.wav');
[x_noise_ventilation, fs_noise_ventilation] = audioread(noise_ventilation);
noise_amb_lib = ('noise_signals/noise_ambient_library_2.wav');
[x_noise_amb_lib, fs_noise_amb_lib] = audioread(noise_amb_lib);
noise_traff_inter = ('noise_signals/noise_traffic_intersection.wav');
[x_noise_traff_inter, fs_noise_traff_inter] = audioread(noise_traff_inter);

% Add noise to baby recordings
L_baby_crying_1=length(x_baby_crying_1);
L_baby_crying_2=length(x_baby_crying_2);
L_baby_talking=length(x_baby_talking);

L_noise_bird=length(x_noise_bird);
L_noise_traffic=length(x_noise_traffic);
L_noise_ventilation=length(x_noise_ventilation);
L_noise_amb_lib=length(x_noise_amb_lib);
L_noise_traff_inter=length(x_noise_traff_inter);

xL_baby_crying1=min(L_baby_crying_1,min(L_noise_bird,min(L_noise_traffic,...
    min(L_noise_ventilation,min(L_noise_amb_lib,L_noise_traff_inter)))));
xL_baby_crying2=min(L_baby_crying_2,min(L_noise_bird,min(L_noise_traffic,...
    min(L_noise_ventilation,min(L_noise_amb_lib,L_noise_traff_inter)))));
xL_baby_talking=min(L_baby_talking,min(L_noise_bird,min(L_noise_traffic,...
    min(L_noise_ventilation,min(L_noise_amb_lib,L_noise_traff_inter)))));

% Version:
% 0 - clean, without any noise
% 1 - slightly amp bird & vent noise added      Nstrength=2
% 2 - slightly amp (all) noise files added      Nstrength=2
% 3 - highly amp (all) noise files added        Nstrength=5

% Baby crying 1
x_BC10=x_baby_crying_1(1:xL_baby_crying1);
Nstrength = 2;
x_BC11=x_baby_crying_1(1:xL_baby_crying1)+...
    Nstrength*x_noise_bird(1:xL_baby_crying1)+...
    Nstrength*x_noise_ventilation(1:xL_baby_crying1);
```

```
x_BC12=x_baby_crying_1(1:xL_baby_crying1)+...
    Nstrength*x_noise_bird(1:xL_baby_crying1)+...
    Nstrength*x_noise_traffic(1:xL_baby_crying1)+...
    Nstrength*x_noise_ventilation(1:xL_baby_crying1)+...
    Nstrength*x_noise_amb_lib(1:xL_baby_crying1)+...
    Nstrength*x_noise_traff_inter(1:xL_baby_crying1);
Nstrength = 5;
x_BC13=x_baby_crying_1(1:xL_baby_crying1)+...
    Nstrength*x_noise_bird(1:xL_baby_crying1)+...
    Nstrength*x_noise_traffic(1:xL_baby_crying1)+...
    Nstrength*x_noise_ventilation(1:xL_baby_crying1)+...
    Nstrength*x_noise_amb_lib(1:xL_baby_crying1)+...
    Nstrength*x_noise_traff_inter(1:xL_baby_crying1);

% Baby Crying 2
N=2;
x_BC20=N*x_baby_crying_2(1:xL_baby_crying2);
Nstrength = 2;
x_BC21=N*x_baby_crying_2(1:xL_baby_crying2)+...
    Nstrength*x_noise_bird(1:xL_baby_crying2)+...
    Nstrength*x_noise_ventilation(1:xL_baby_crying2);
x_BC22=N*x_baby_crying_2(1:xL_baby_crying2)+...
    Nstrength*x_noise_bird(1:xL_baby_crying2)+...
    Nstrength*x_noise_traffic(1:xL_baby_crying2)+...
    Nstrength*x_noise_ventilation(1:xL_baby_crying2)+...
    Nstrength*x_noise_amb_lib(1:xL_baby_crying2)+...
    Nstrength*x_noise_traff_inter(1:xL_baby_crying2);
Nstrength = 5;
x_BC23=N*x_baby_crying_2(1:xL_baby_crying2)+...
    Nstrength*x_noise_bird(1:xL_baby_crying2)+...
    Nstrength*x_noise_traffic(1:xL_baby_crying2)+...
    Nstrength*x_noise_ventilation(1:xL_baby_crying2)+...
    Nstrength*x_noise_amb_lib(1:xL_baby_crying2)+...
    Nstrength*x_noise_traff_inter(1:xL_baby_crying2);

% Baby talking
N=3;
x_BT0=N*x_baby_talking(1:xL_baby_talking);
Nstrength = 2;
x_BT1=N*x_baby_talking(1:xL_baby_talking)+...
    Nstrength*x_noise_bird(1:xL_baby_talking)+...
    Nstrength*x_noise_ventilation(1:xL_baby_talking);
x_BT2=N*x_baby_talking(1:xL_baby_talking)+...
    Nstrength*x_noise_bird(1:xL_baby_talking)+...
    Nstrength*x_noise_traffic(1:xL_baby_talking)+...
    Nstrength*x_noise_ventilation(1:xL_baby_talking)+...
    Nstrength*x_noise_amb_lib(1:xL_baby_talking)+...
    Nstrength*x_noise_traff_inter(1:xL_baby_talking);
Nstrength = 5;
x_BT3=N*x_baby_talking(1:xL_baby_talking)+...
    Nstrength*x_noise_bird(1:xL_baby_talking)+...
    Nstrength*x_noise_traffic(1:xL_baby_talking)+...
    Nstrength*x_noise_ventilation(1:xL_baby_talking)+...
    Nstrength*x_noise_amb_lib(1:xL_baby_talking)+...
    Nstrength*x_noise_traff_inter(1:xL_baby_talking);
```

References

[1] R. Cohen, Y. Lavner, *Infant Cry Analysis and Detection*, 2012

[2] E. Verteletskaya, K. Sakhnov, *Voice Activity Detection for Speech Enhancement Applications*, 2010

[3] N. Westerlund, M. Dahl, *Speech Enhancement using an Adaptive Gain Equalizer*, 2003

[4] R. Narayanam, *An Efficient Peak Valley Detection based VAD Algorithm for Robust Detection of Speech Auditory Brainstem Responses*, 2013

[5] J. Sohn, N.S. Kim, W. Sung, *A Statistical Model-Based Voice Activity Detection*, 1999

[6] O. Sutton, *A Statistical Model-Based Voice Activity Detection*, 2012

[7] TappyTaps, *https://www.babymonitor3g.com*,

[8] Sleekbit, *http://dormi.sleekbit.com/index.html*,

[9] D. Jankovic, M. Johansson, M. Lichota, *Adaptive Gain Control in Digital Signal Processors*, 2015