Blind signal source separation using the Degenerate Unmixing Estimation Technique implemented on the ADSP-21262 SHARC digital signal processor

Jens Elofsson
adi09jel@student.lu.se

Anton Martinsen
ada09ama@student.lu.se

Alexander Pieta Theofanous
zba08ath@student.lu.se

March 2014

1 Introduction

In this report a solution for separating two audio sources using a microphone array with two microphones is presented. The method used for this purpuse is called a degenerate unmixing estimation technique, abbreviated as DUET. There are several methods and algorithms for signal separation that use similar or even completely different concepts than DUET. The DUET method is suited for audio source separation where the signal, the sound, is human speech [**4**].

The development process consisted of two implementations. The DUET-algorithm was initially implemented in MATLAB using two synthetic audio signals as input. Afterwards, when the functionality was asserted, the DUET-algorithm was implemented on the digital signal processor, hereby referred to as the DSP, with audio signals in real time.

1.1 What is audio signal separation?

Signal separation consists of separating one source from another source, contained in two or more signals that contain all the sources. If there is no information about the sources or how the signals are mixed, when the signals arrive at each input port, the task of separating those signals is called blind signal separation, commonly abbreviated BSS [**3**].

1.1.1 Applications of BSS

BSS is found in several products such as optical and medical instruments as well as cellphones. When talking on a telephone in a noisy surrounding, it is desirable to separate all the sounds coming from the speakers mouth, from the sounds of the surrounding environment.

1.1.2 Design issues for blind audio source separation systems

There are many factors to have in mind if an optimal result is to be achieved when using BSS. Different results can be achieved depending on what hardware the separation algorithm is to be implemented on as well as the quality of and the number of microphones used.

Limitations for achieving a sufficiently good separation can depend on the sampling rate of the audio signal as well as the computing speed of the system's processor.

1.2 Tools used

The DSP used in this project was the ADSP-21262 SHARC by Analog Devices. The development environment used to program the DSP was Visual DSP++ 5.0. MATLAB was used for testing the algorithm and generating plots.

The microphone array was built using two metal clips attached to a soldering holder. A J-TAG emulator interface device was used to program the flash

memory of the DSP. The emulator was connected between the DSP and the computer, programmed from the computer via USB.

2 Theory

2.1 Fast Fourier Transform and the Inverse Fast Fourier Transform

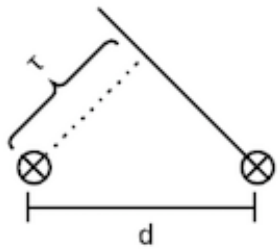
The Fourier Transform is a mathematical function which transfers a signal from the time domain to the frequency domain. This makes it possible to view the frequency content of a given signal, represented in the time domain. In order to transform a signal in the frequency domain to the time domain the Inverse Fourier Transform is applied on the signal [**2**].

When converting a signal from the time domain to the frequency domain, information about the signal is lost. The signal's varying amplitude in different time slots cannot be reconstructed again, after fourier transformation. To avoid losing information about the signals being transformed, finite discrete transforms can be used. Finite transforms set the limits of the Fourier functions to a specific value instead of infinity [**5**].

One implementation of the Fourier Transform is the Fast Fourier Transform, which requires a block size of $N = 2^k$. Fast Fourier Transform and Inverse Fast Fourier Transform is abbreviated FFT, and IFFT, respectively.

2.2 Signal separation algorithm

The foundation of source separation is the fact that if a planar wave arrives at a microphone array, it will arrive at the second microphone with a delay relative to the first.



Considering two sources s_1 and s_2, and x_1 and x_2 as the audio inputs to the microphones. τ is the time delay for the sound to reach the second microphone.

$$x_1 = s_1(t) + s_2(t + \tau_2) \tag{1}$$

$$x_2 = s_2(t) + s_1(t + \tau_1) \tag{2}$$

By using the FFT, the signals translate into the frequency domain

$$X_1(\omega) = S_1(\omega) + S_2(\omega) * e^{j\omega\tau_2} \tag{3}$$
$$X_2(\omega) = S_2(\omega) + S_1(\omega) * e^{j\omega\tau_1} \tag{4}$$

To separate the two sources from each other the cross spectrum between the two signals is calculated by

$$G(\omega) = X_1(\omega) * \overline{X_2}(\omega) \tag{5}$$

Inserting the above equations of X_1 and X_2 provides the following equation

$$G(\omega) = S_1(\omega)\overline{S_2}(\omega)+S_1(\omega)\overline{S_1}(\omega)*e^{-j\omega\tau_1}+S_2(\omega)\overline{S_2}(\omega)*e^{j\omega\tau_2}+S_2(\omega)\overline{S_1}(\omega)*e^{j\omega\tau_2-j\omega\tau_1} \tag{6}$$

Since only one source is considered active in the time-frequency plane at once this equation can be divided into

$$G(\omega) = \begin{cases} S_1(\omega)\overline{S_1}(\omega) * e^{-j\omega\tau_1}, & \text{if } s_1 \text{ is active.} \\ S_2(\omega)\overline{S_2}(\omega) * e^{j\omega\tau_2}, & \text{if } s_2 \text{ is active.} \end{cases} \tag{7}$$

Looking at this one realizes that if the angle of the cross spectrum is negative it means that s_1 is active for this frequency, and if the angle is positive s_2 is active for this frequency. An easy way to check if the angle is negative or positive is checking the imaginary part of the cross spectrum. If the imaginary part is negative then the angle is also negative and vice versa.

Taking the imaginary part of the cross spectrum for each frequency yields

$$Im(G(\omega)) = \begin{cases} -|S_1(\omega)|^2 * sin(\omega\tau_1), & \text{if } s_1 \text{ is active.} \\ |S_2(\omega)|^2 * sin(\omega\tau_2), & \text{if } s_2 \text{ is active.} \end{cases} \tag{8}$$

2.3 Spectral leakage

If the signal, that is to be transformed by the FFT, is periodic then the FFT of that signal will produce a narrow and distinct peak at a certain frequency. A signal that is not periodic within the finite window length of the FFT, which is common for audio signals sampled at short intervals, will produce discontinuities of varying magnitudes and distributions. This unwanted spread of frequencies, the ripples, is called spectral leakage **[1]**.

2.4 Hamming window

In order to avoid spectral leakage, a Hamming Window can be applied on a signal in the frequency domain. The Hamming window is an array with values ranging from 0 to 1. When plotted, a hamming window takes on the shape of a bell curve. The Hamming window has relatively low values, close to zero, at the endpoints and values up to one, at the center of the window i.e the middle of the array. When the Hamming window is applied on a signal the magnitudes of the frequencies at the highest and lowest part of the frequency spectrum is attenuated. The magnitude of the frequencies in the middle of the spectrum stay mostly the same[**6**]. The closer to the endpoints, the more attenuation.

2.5 Overlap-add

Hamming windows solve the problem of spectral leakage but introduces a new problem: loss of information. Since the frequencies at the endpoint of the signal decrease in magnitude, an IFFT would result in a weaker and more distorted signal in the time domain than is expected. Overlap-add is a method used to preserve lost data. The method begins with saving a subset of the signal which has had a Hamming window applied to it as well as being transformed to the time domain via the IFFT. Thus, the signal to be saved has been processed and transformed into the time domain. In the next iteration, or processing phase, the signal that is currently being processed is manipulated. The manipulation consists of adding the previously saved signal with the current signal. Thereafter the processing continues whereby the manipulated signal is used in the FFT and so on. Only a segment, usually half, of the signal is saved for later iterations for the purpose of noise reduction, specifically the clicking sounds of the output, and audibility[**1**].

3 Implementation

3.1 In MATLAB

The signals processed in MATLAB were two synthetically mixed sources. This differs slightly from the implementation on the DSP in two ways. Firstly, the processing was not done in real time. This meant that the input signals had to be zero-padded and divided into blocks, in order to make it as realistic and easy as possible to implement in C later on.

The second difference was that since the two input sources in this case were synthetically mixed, the separation was near-perfect. This was not the case in the real-world implementation, which contains background noise amongst other things.

A demonstration of the results of the MATLAB implementation can be seen in figure 1, where it clearly can be seen that the algorithm filters out a lot of

the signal content. Note that the Matlab implementation used no, by us implemented, Hamming Windows and is void of the overlap-add method. The Matlab-functions that were used veiled the intricate usage of the abovementioned concepts and methods.

3.2 In VisualDSP++

The implementation on the DSP was quite different since the input signals were not synthetically produced. Instead the input came from two microphones. Thus the input signals contained a significant amount of noise. Since the nature of the project required the separation to occur in real-time there was also a time constraint.

The framework provided on the course webpage contained the necessary header-files which specified, amongst other things, the sampling rate and sampling frequency to be used by the DSP.

The sampling rate was 8 kHz and the sampling was done in blocks of 256 samples from each microphone input. The signals were transformed to the frequency domain with a 256-point FFT and a Hamming window was applied to the sampled block. present in the signals a threshold was introduced in order to filter the noise. The threshold was calculated as one hundredth of the mean power.

Before converting back to the time domain the resulting signal is made symmetric, in order to ensure that it is real-valued in the time domain [**6**].

4 Discussion

Since no member of the group had studied any clear cut signal processing courses, other than an introductory one, we encountered several roadblocks. Our memory had to constantly be refreshed and our knowledge was constantly appended with more new information. On the other hand, we are well versed in the usage of C and thus had no issues using the development environment.

The final product provides a somewhat coarse source separation with an acceptable audio quality. The functionality requested is provided. It was a fun project to work with and our tutor was extremely helpful when it came to distinguish what parts of the signal processing algorithm to implement "by hand" as well as which functions to use in our calculations.

4.1 Problems encountered

The MATLAB code was used as an outset for our C implementation. Since we had confirmed a near-perfect separation with good audio quality, we constantly tried to replicate the Matlab functionality in C. After translating the MATLAB code to C, to our best extent, we still could not achieve either a decent source separation or an output signal with good quality. After many weeks of trial and

error it was discovered that moving the microphones further away from each other gave better results. To ensure this was not by pure accident, we analyzed the imaginary part of the cross spectrum, which revealed that it in fact was not a coincidence but perfectly reasonable.
If the distance between the two microphones is small then τ_1 and τ_2 will be very small, close to zero.

By letting τ_1 and τ_2 go towards zero in equation (8)

$$\lim_{\tau_1,\tau_2 \to 0} Im(G(\omega)) = \begin{cases} -|S_1(\omega)|^2 * sin(\omega\tau_1) \\ |S_2(\omega)|^2 * sin(\omega\tau_2) \end{cases} \tag{9}$$

Then

$$Im(G(\omega)) \to 0 \tag{10}$$

This will be a problem for the algorithm since it separates the two sources in regard to if the imaginary part of the cross correlation is positive or negative. When calculating small values in the DSP, even small numerical errors will have a huge impact on the result. A positive imaginary value could be mistaken for a negative imaginary value and the algorithm would fail to recognize this. Thus the distance between the two microphones has to be large enough for the algorithm to work.

Another problem discovered was that the imaginary part of the cross spectrum changes sign for negative frequencies. This can be seen in equation (8). This was solved by using that a signal is symmetric around 0. Thereby only evaluating and processing the cross spectrum for the positive frequencies is needed. The complex conjugate of the signal, at a specific frequency, is placed on the mirrored location of that frequency, in the spectrum of the same signal.

$$X_1(\omega) = \overline{X_1}(-\omega) \tag{11}$$

5 References

1. http://www.physik.uni-wuerzburg.de/~praktiku/Anleitung/Fremde/ANO14.pdf
2. http://mathworld.wolfram.com/FastFourierTransform.html
3. http://cis.legacy.ics.tkk.fi/aapo/papers/IJCNN99_tutorialweb/node3.html
4. http://www.math.msu.edu/~ywang/Preprints/DUET-IEEE.pdf
5. http://stackoverflow.com/questions/5117839/understanding-overlap-and-add-for-filtering
6. http://dsp.stackexchange.com/questions/736/how-do-i-implement-cross-correlation-to-prove-two-audio-files-are-similar

6 Appendix A: Figures

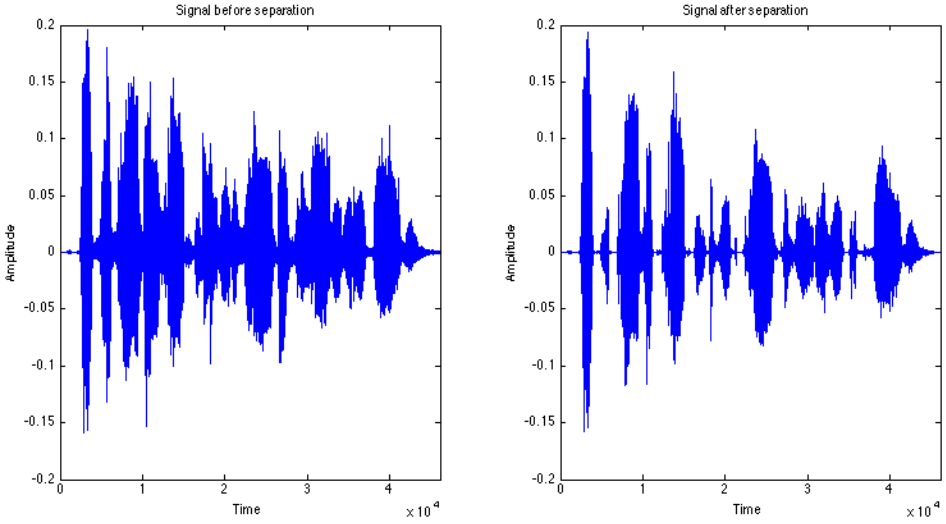


Figure 1: The signal before separation (left) compared to the signal after the separation (right) in the time domain

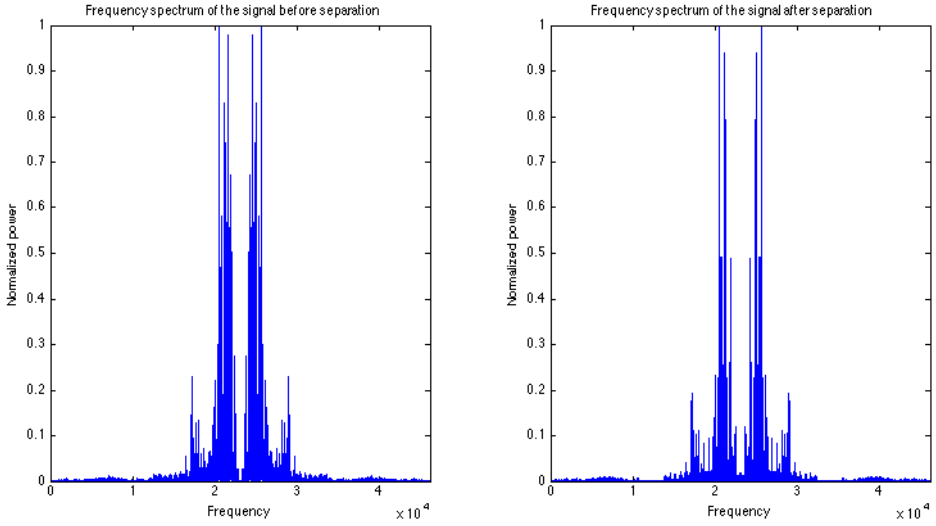


Figure 2: The signal before separation (left) compared to the signal after the separation (right) in the frequency domain

7 Appendix B: Code

7.1 MATLAB

```
load('mixing.mat')
NFFT = 128;

s1 = [s1' zeros(1,80)]';
s2 = [s2' zeros(1,80)]';

S1 = [];
S2 = [];

for i=1:length(s1)/NFFT
    temp1 = s1((i-1)*128+1:i*128);
    S1 = [S1 temp1];
    temp2 = s2((i-1)*128+1:i*128);
    S2 = [S2 temp2];
end

X1 = fft(S1);
X2 = fft(S2);

Y1 = zeros(size(X1));
Y2 = zeros(size(X2));

for i=1:size(S1,2)
   G1 = X1(1:128/2,i).*conj(X2(1:128/2,i));
   G2 = X2(1:128/2,i).*conj(X1(1:128/2,i));

   for k = 1:length(G1)
      if imag(G1(k)) > 0
         Y1(k,i) = X1(k,i);
         Y1(end+2-k,i) = conj(X1(k,i));
      end

      if imag(G2(k)) > 0
         Y2(k,i) = X2(k,i);
         Y2(end+2-k,i) = conj(X2(k,i));
      end
   end

end

y1sym = ifft(Y1,'symmetric');
y2sym = ifft(Y2,'symmetric');
y1 = ifft(Y1);
y2 = ifft(Y2);
y1r = real(ifft(Y1));
y2r = real(ifft(Y2));

out1sym = y1sym(:);
out2sym = y2sym(:);
out1 = y1(:);
out2 = y2(:);
out1r = y1r(:);
out2r = y2r(:);
```

7.2 C

```
1  /*****************************************************************************
2   * 2014-03-04
3   *****************************************************************************/
4
5
6  #include <processor_include.h>
7  #include <signal.h>
8  #include <stdfix.h>
9  #include <string.h>
10 #include <filter.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include "framework.h"
14 #include <filter.h>
15 #include <math.h>
16 #include <complex.h>
17 #include <stats.h>
18 #include <window.h>
19
20 static complex_float complex_zero, ham;
21 static unsigned int filter_left,T, size;
22 const int debug = 1;
23 static complex_float output[DSP_BLOCK_SIZE], complex_in_left[
       DSP_BLOCK_SIZE],complex_in_right[DSP_BLOCK_SIZE],fft_right[
       DSP_BLOCK_SIZE], fft_left[DSP_BLOCK_SIZE], temp[DSP_BLOCK_SIZE],
       ifft_output[DSP_BLOCK_SIZE],accumulator[DSP_BLOCK_SIZE];
24 static float dm hamming[DSP_BLOCK_SIZE];
25 static float dm threshold, amp_sum, amp_count, coeff;
26
27 void process(int sig)
28 {
29    sample_t *audio = dsp_get_audio();
30    int i;
31    float c = 1.0;
32    float d = 0.0;
33    if(filter_left){
34      c=0.0;
35      d=1.0;
36    }
37   for(i=0; i<DSP_BLOCK_SIZE; ++i) {
38      complex_in_left[i].re = (float) audio[i].left / (double)(1<<31) +
           accumulator[i].re*c;
39      complex_in_right[i].re = (float) audio[i].right / (double)(1<<31) +
            accumulator[i].re*d;
40      complex_in_left[i].im = complex_in_right[i].im = 0.0;
41    }
42      cfft256(complex_in_right, fft_right);
43   cfft256(complex_in_left, fft_left);
44   for(i=0; i<DSP_BLOCK_SIZE; i++){
45      ham.re = hamming[i];
46      fft_right[i] = cmltf(ham,fft_right[i]);
47      fft_left[i] = cmltf(ham,fft_left[i]);
48   }
```

```
49    if(filter_left){
50        memcpy(temp, fft_left, size);
51        memcpy(fft_left, fft_right, size);
52        memcpy(fft_right,temp, size);
53      }
54    if(debug==0){
55      memcpy(output, fft_left, size);
56    }else{
57        for (i = 0; i < DSP_BLOCK_SIZE/2 + 1; i++) {
58        complex_float G = cmltf(fft_left[i], conjf(fft_right[i]));
59        float amp = pow(cabs(fft_left[i]),2);
60
61        float check= (MAX_FLOAT - 100.0);
62        if(amp_count<= check && amp_sum <= check){
63          amp_sum += amp;
64          amp_count++;
65        }
66        float mean_amp = amp_sum / amp_count;
67        threshold = coeff * (mean_amp / 100.0);
68        if(G.im > 0.0 || amp < threshold){
69          output[i] = fft_left[i];
70        }else{
71          output[i] = complex_zero;
72        }
73      }
74      for(i=1; i < DSP_BLOCK_SIZE/2; i++){
75        output[DSP_BLOCK_SIZE - i] = conjf(output[i]);
76      }
77    }
78    ifft256(output, ifft_output);
79    for(i=0; i<DSP_BLOCK_SIZE/2; i++){
80        accumulator[i] = ifft_output[DSP_BLOCK_SIZE/2 + i];
81      }
82    for(i=0; i<DSP_BLOCK_SIZE; ++i) {
83      audio[i].left = audio[i].right = (ifft_output[i].re * ((float)
          (1<<31)));
84    }
85  }
86
87
88
89
90  static void keyboard(int sig)
91  {
92    if(T==0){
93      T = dsp_get_keys();
94
95      switch(T)
96       {
97        case 1:
98          filter_left = 1;
99          break;
100       case 2:
101         filter_left = 0;
102         break;
103       case 4:
104         coeff = coeff - 0.1;
```

```
105           break;
106         case 8:
107           coeff = coeff + 0.1;
108           break;
109         default:
110           printf("You can only press one key at a time!\n");
111
112       }
113    }else{
114      T=0;
115    }
116  }
117
118
119
120  static void timer(int sig)
121  {
122
123  }
124
125
126  void main( )
127  {
128    size = DSP_BLOCK_SIZE*sizeof(complex_float);
129    memset(complex_in_left, 0, size);
130    memset(output, 0, size);
131    memset(complex_in_right, 0, size);
132    memset(fft_right, 0, size);
133    memset(fft_left, 0, size);
134    memset(temp, 0, size);
135    memset(accumulator,0,size);
136    complex_zero.re = complex_zero.im = 0.0f;
137    ham = complex_zero;
138     dsp_init();
139     T=filter_left=0;
140     threshold = amp_sum=amp_count=0;
141     coeff=0.995;
142     gen_hamming (hamming,1,DSP_BLOCK_SIZE);
143     interrupt(SIG_SP1, process);
144     interrupt(SIG_USR0, keyboard);
145     dsp_start();
146     for(;;) {
147        idle();
148     }
149  }
```