SPEECH NOISE SEPARATION USING NON-NEGATIVE MATRIX FACTORIZATION

MOHAMMAD SOLAIMAN AL-ASHKAR
A`LAA MOHAMMAD ESMAIL

Computer Science Department– Collage of Science – Cairo University

Supervised by: Dr. ABD EL-AZIZ AHMED
Assistant Professor of Information Systems, ISSR, Cairo University

Final Project Spring 2016

ABSTRACT:

In this work we focus on the single channel audio source decomposition, since many of modern technologies like visual assistance systems and many other software solutions depends on the human sound interaction there is a need to efficient techniques to human speech sound enhancement to get as possibly as pure human sound before the speech regression.

Speech regression is a technology that allows a computer to identify the words that a person speaks into a microphone or telephone.

Speech recognition can be defined as the process of converting an acoustic signal, captured by a microphone or a telephone, to a set of words [9].

In the automatic subtitling which is widely used technique in many software solutions like “Siri” visual assistance system by apple co. “Youtube” video automatic subtitling, “Google translate” system which has a speech regression ability to convert the human speech to text before translate, and many other software solutions which has a speech regression system as a major part of these systems.

The accuracy of speech regression phase depends on the technique that is used and the speech sound itself, so if the speech sounds containing a lot of noise, the regression system can’t give a good result. So to improve the accuracy of the speech regression system it is very important to improve the sound processing phase to get a pure human speech sound without any noise [5], this property require a good technique to extract the human speech sound form the mixed sound. By this work we will provide an effective technique to extract a pure human speech sounds from the mixed sounds to help the improvement of speech regression systems.

Contents

Figures

Chapter 1
Big Picture and Introduction

1.1 Introduction:

As we say that the process of remove the music or the noise from the background of the speech sound is very useful to improve the regression accuracy of speech regression system. Single channel source separation (SCSS) is very challenging problem because only one measurement of the mixed signal is available [5]. Most of ideas that mention on this problem use a prior knowledge technique that is "training date" of the signals to be separated [5]. The process of sound source separation (SSS) can be done with many algorithms. Non-negative Matrix Factorization (NMF) and (NTF) algorithms in [3] and [4] are used in many sound separation researches with good results in many of it. Our goal is to use NMF to develop an effective model to separate speech signals from mixed signals which will be used in the speech regression of automatic subtitle system in the future work.

Our work is implementing a (unsupervised*, Simi-supervised and supervised*) speech noise decomposition using NMF algorithm in the two measure phases (training phase and testing phase), using the 'winner' mask as in [5] to help in separating process. We use a dataset consists of "Speech sound sources" and "background noise sources", these two sets will be used to train our algorithm using NMF to find the Basis and the Weights vectors for each sound source from the training set.

$$S_{speech} = B_{speech} W_{speech}$$

$$S_{noise} = B_{noise} W_{noise}$$

In the training phase we will take only the basis for each sound source to make the dictionary, and in the separation phase we will retrieve the basis from the database depending on the algorithm status (supervised, unsupervised, Simi-supervised) and the spectrogram setting.

In supervised NMF we will retrieve all speech and noise basis of the specific STFT window length to make the basis dictionary

$B = [B_{s1}\, B_{s2} \ldots B_{sj}\, , B_{n1}\, B_{n2} \ldots B_{nk}]$, since in the Simi-supervised we will retrieve only speech basis B_s and we can use some equilibrium basis Beq which are a noise basis used to minimize the error of speech extracting process. Then run the NMF to detect the weights W which will be as

$W = [Ws\, ;\, Wn]$, after that we can extract the speech and noise by multiplying the basis and the weights of the speech and noise from B and W as :

$$S = B_s \times W_s \qquad N = B_n \times W_n$$

Then using 'winner' filter [5] to find speech sound from input sound STFT complex matrix.

1.2 Sound Source Separation Problem Definition:

Source separation problems are these which trying to recover the original signal sources from a mixed compound signal. The classical example for these problems is "cocktail party problem" where a number of people are talking simultaneously in a room (for example, at a cocktail party), and a listener is trying to follow one of the discussions The human brain can handle this sort of auditory source separation problem, but it is a difficult problem in digital signal processing, First analyzed by "***Colin Cherry*** ". Our problem is an single channel speech-noise separation and it can be defined as follows: let x(t) be the observed sound signal which is consists of mixed speech s(t) and noise n(t) signals the source separation problem aims to find s(t) and n(t) from observed x(t) [5]. To do that we have to use Time-Frequency Representation (TFR) methods in [11] to convert the sound signals from Time domain to Time-Frequency we can use Short-Time-Fourier-Transform (STFT) which is a Discrete Fourier Transform or we can use Constant-Q-Transform.

1.3 Sound Separation Objectives:

- Speech sound analysis need for sound separation to enhance the speech sound.
- Very useful in speech enhancement techniques.
- Many mobile applications need the speech enhancement to reduce the noise from the background of speaker sound.

1.4 Sound Source Separation Methods:

There are many methods for signal separation problem and in particular for sound source separation problems there are three main families of methods:

- **Independent Component Analysis (ICA):** depends on the ascension that the sound sources are statistically independent, stationary and at most one of them is Gaussian.
- **Sparse Component Analysis (SCA):** Sparse sources (i.e. most of the samples are null (or close to zero)).
- **Non-negative Matrix Factorization (NMF)**

1.5 Bit History of Sound Source Separation Problems:

BSS problem formulated around 1982, by Hans, **Hérault**, and **Jutten** for a biomedical problem and first papers in the mid of the 80's. Great interest from the community, mainly in **France** and later in **Europe** and in **Japan**, and then in the **USA.** The first workshop (conference) on ICA and BSS was held in **Aussois**, **France**, in January 1999. After that, these conferences have been arranged about every 18 months under slightly varying names [13]. In June 2009, 22000 scientific papers are recorded by Google Scholar. Initially, BSS addressed for LI mixtures but convolutive mixtures in the mid of the 90's and nonlinear mixtures at the end of the 90's. Until the end of 90's the BSS was asymptotic to ICA. First NMF methods in the mid of the 90's but famous contribution in 1999.

Chapter 2
System Analysis

2.1 Big Picture

The goal of our system is to provide a good technique to test all options of the speech sound source separation using Non-negative Matrix Factorization algorithm. We provide an user friendly program that can be used to separate the single channel mixed (speech, noise) sounds source in many options of the NMF algorithm, the program has the ability to learn with given pure sounds dataset, user can add sounds to dataset to help the system do well in larger range of the mixed sounds. The system give the ability to control the state parameters and give three modes of algorithm status Supervised, Simi-Supervised and Unsupervised, in addition to three filter status Winner, Linear and Hard filters, so user can try all these cases on separation state.

2.2 Program Workflow

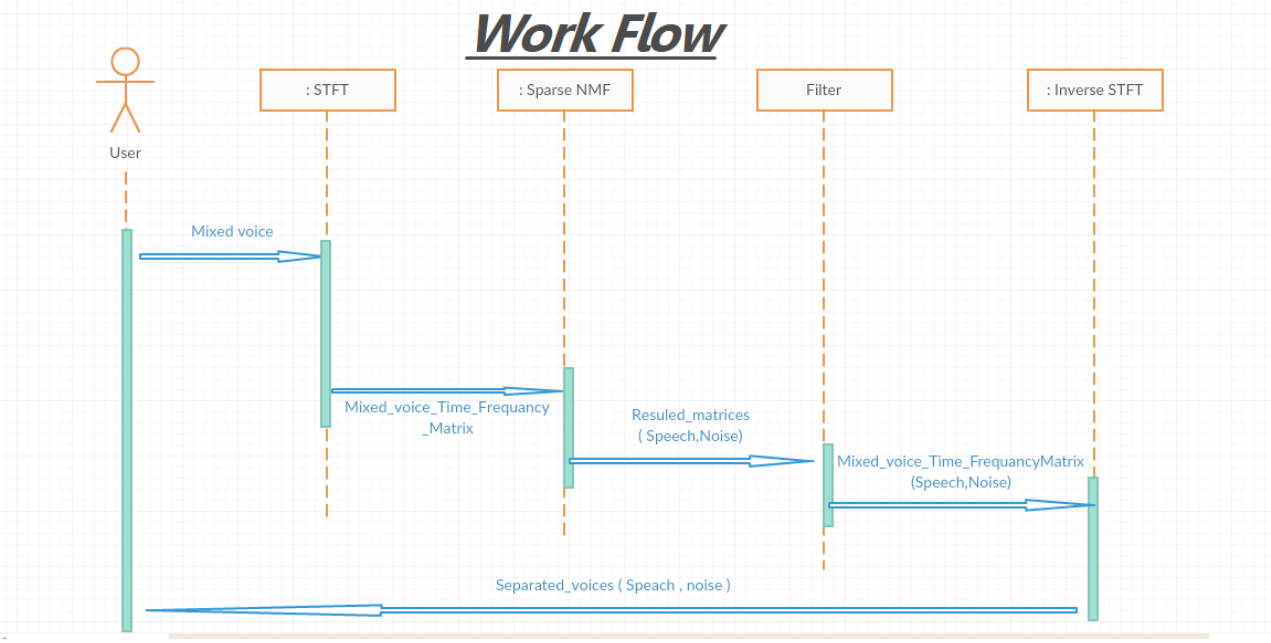

Figure 1: Source Separation Workflow Diagram

This diagram illustrates Algorithm steps Such that User Enters the mixed .wav voice and which algorithm want to use in separation (Supervised, Simi Supervised, Unsupervised) in addition to setup the parameters like (STFT window size, number of speech and noise components for the unsupervised,

the noise components for the Simi supervised, regularization parameter and filter status)

- **First Step:** the mixed Sound Pass STFT (Short-Time-Fourier-Transform) to result Time frequency matrix for the mixed voice.
- **Second Step:** the result from STFT is passed into sparse NMF (Sparse regularization for Non-negative Matrix Factorization) and that results Tow matrices N, S (Noise, Speech) Matrices
- **Third Step:** two matrices that result from Sparse NMF (Second Step) are passed to the filter step to obtain the separated voices (Speech, Noise) and are masque.
- **Forth Step:** Two Time frequency matrices results from the Filter Step passing to the inverse of STFT (Short-Time-Fourier-Transform) to obtain .wav voices (Speech, Noise).

2.2.1 Database Storing

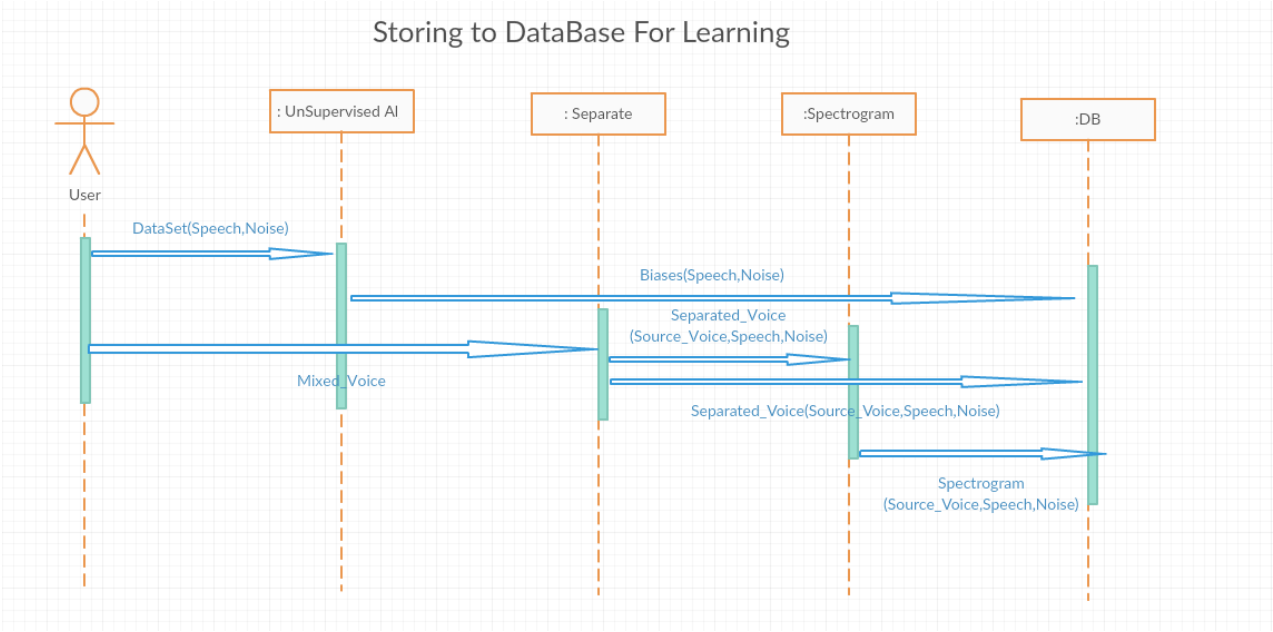

Figure 2: Learning and storing in database

Here we illustrate steps in storing database. The database contains the output of the learning process (Dataset of the algorithm) in addition to the output of separation process (speech, noise and spectrograms).

2.2.2 Learning

The user select set of speech or noise sounds and set the STFT window size for the selected sounds and the number of components K for each sound which will

results the basis B with K columns resulted from the unsupervised algorithm and store it in the database.

2.3 Database Design

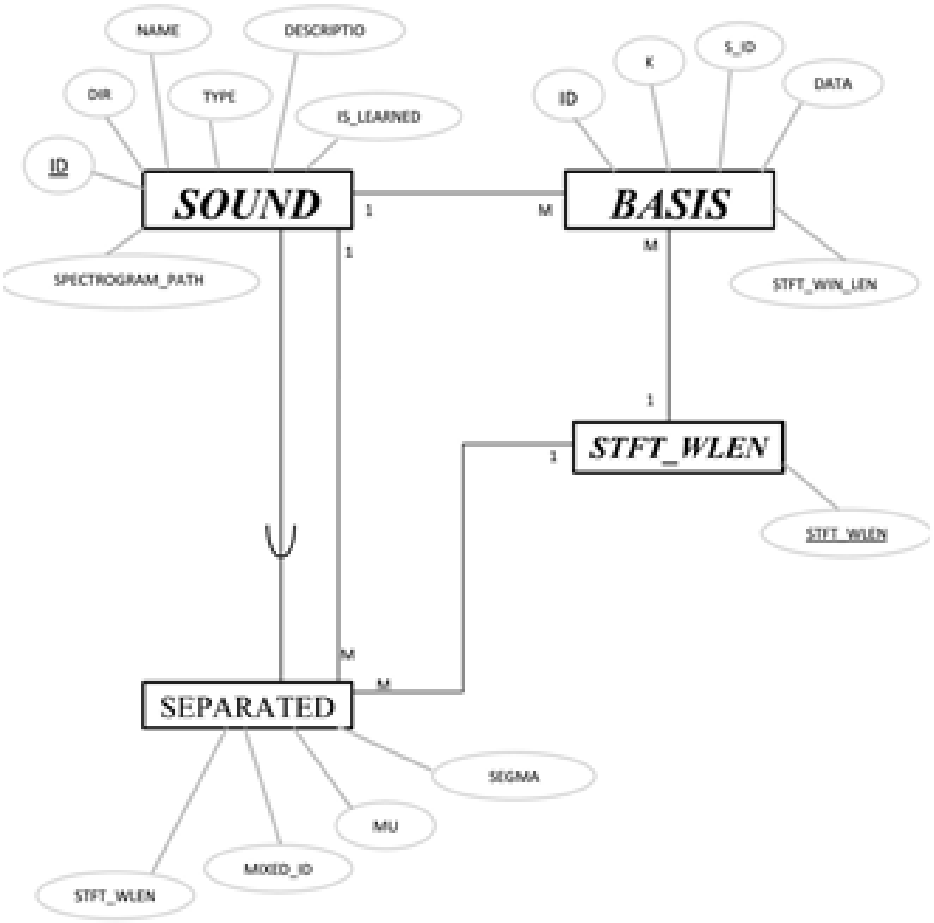

Figure 3: System Database EER Model

2.3.1 Database Relations

Between sound and basis Entities:

Every sound has more than one bias such that user may more than STFT window length for one Sound …So it will result different biases.

Between Sound and Separated:

Separated sound is a sound but it results from a mixed sound after separation. So for every one mixed sound two separated sounds (Speech and Noise) but also that depends on STFT window length such that for one STFT window length results one type of separated sounds.

Between Biases an STFT window length:

Every one base result from one type of STFT window length and also for every STFT window length one bias is resulted.

Chapter 3
Implementation Requirements

3.1 Time Frequency Representation:

In digital signal processing it is important to convert the time signal from the time domain to the frequency domain and results the time-frequency representation of the original time domain signal. The audio source in initial it is time domain array with diminutions $C \times N$ where C is the number of channels and N the number of bins, in our work we converted all sounds to mono sounds which will be an 1D-array of N bins then we used the STFT algorithm to calculate TFR as 2D-Matrix with $F \times T$ elements.

STFT is a powerful tool to obtaining TFR, it was first proposed in [12]. The STFT is used for analyzing non-stationary signals, whose frequency characteristics vary with time [11]. The following equation consumer the STFT [11]:

$$STFT(x[n]) \equiv X(l,w) = \sum_{n=-\infty}^{+\infty} x[n]w[n-l]e^{-jwl}$$

And the spectrogram which represents the spectral content of signal vs. time can be found as follow:

$$Spectrogram(x[n]) = |X(l,w)|$$

A limitation of using the STFT is that it uses a constant resolution in both frequency and time. So the product of time resolution and frequency resolution remains constant Thus, a better time resolution is obtained at a price of poorer resolution in frequency and vice versa. In other words, to analyze the frequency content accurately, we need more samples (larger window in time) in each frame [11].

The other way to get TFR is CQT by 'JC Brown' [7] which is better suited representation for musical signal due its log-frequency spectral resolution.

The CQT for discrete-time signal can be calculated by the following equation:

$$X_{cq}[k] = \sum_{n=0}^{N[k]-1} W[n,k]x[n]e^{-jwkn}$$

Where $X_{cq}[k]$ is the k_{th} component of the Constant Q transform of the input signal x[n]. W [n, k] is a window function of length N[k] for each value of k and k varies from 1, 2 . . . K which indexes the frequency bins in the Constant Q domain [11].

There are two measure drawbacks to CQT. The first one is that there is no true inverse of the transform so it is impossible to get perfect resolution of the original sound but recently ***Sch'orkhuber*** and ***Klapuri*** in [2] has proposed method to calculate CQT which allowed high quality inverse CQT to be calculated.

3.1.1 Short Time Fourier Transformation Algorithm

We used the STFT for single channel signal so we convert all sounds to mono sounds at runtime before applying the STFT

Algorithm STFT_SINGLE_CHANNEL [10]

***INPUT*: A stream of (real)samples $x(n)$at sampling rate R, a buffer *buf* for storing , input samples,**
a buffer *fftbuf* for the in − place FFT, a windowing function *window* of size *N*

Output: A sequence of frames X_m at sampling rate R/H where *H* is the hop size, each frame being a complex number sampled function of frequency bin k, $X_m(k)$ where $k = 0, 1, \dots, N/2$

---

1. $n \leftarrow 0$; *input sample counter*
2. $m \leftarrow 0$; *Output frame counter*
3. $bufcounter \leftarrow 0$; *Index into buffer buf*
4. ***repeat***
5. ***while*** $bufcounter < (N-1)$***do***
6. $buf[bufcounter] \leftarrow x[n]$
7. $bufcounter \leftarrow bufcounter + 1$
8. $n \leftarrow n + 1$
9. ***end while***
10. *Copy buffer buf to buffer fftbuf*
11. $fftbuf = fftbuf * window$
    //Apply windowing function by multiplying pointwize
12. $FFT(fftbuf)$
    //Carry out FFT in place, so output is written back to fftbuf
13. $X_m \leftarrow fftbuf$
14. $m \leftarrow m + 1$
15. *//Prepare to fill next buffer for FFT*

16. $\boldsymbol{for\ g = 0\ to\ N - H - 1\ do}$
17. $\boldsymbol{buf[g] \leftarrow buf[g + H]}$
18. $\boldsymbol{end\ for}$ $\boldsymbol{//Hop\ by\ H\ samples; update\ contents\ of\ buf}$
19. $\boldsymbol{bufcointer \leftarrow N - H}$
20. $\boldsymbol{until\ there\ are\ fewer\ than\ H\ samples\ left\ in\ x}$

STFT parameters:

1. Window size N (usually a power of tow, e.g., 512, 1024, 2048);
2. We use sin window in our implementation
3. Hop size H (typically half or a quarter of N; alternatively expressed as overlap proportion or number of samples of overlap between successive window. Often chosen to match the main lobe size of the window in the time domain);
4. Frame rate = R/H; where R is sample rate

3.2 Non-negative Matrix Factorization (NMF):

After getting **TFR.** X(t, f) from the observed sound x(t) by STFT or CQT we will use it to find the basis and weights using NMF. Due the linearity of STFT and CQT we can write:

$$X(t,f) = S(t,f) + N(t,f)$$

$$Or \qquad X = S + N$$

Where ' t ' represent time frame index and ' f ' represent frequency index. We need to estimate the spectrograms $\boldsymbol{S}$ and $\boldsymbol{N}$ where $\boldsymbol{S}$ is speech spectrogram and $\boldsymbol{N}$ is the noise spectrogram.

Non-negative matrix factorization is an algorithm that is used to decompose any nonnegative matrix $\boldsymbol{V}$ into a nonnegative basis vectors matrix $\boldsymbol{B}$ and a nonnegative weights matrix $\boldsymbol{W}$ every column vector in the matrix $\boldsymbol{V}$ is approximated by a weighted linear combination of the basis vectors in the columns of $\boldsymbol{B}$, the weights for basis vectors appear in the corresponding column of the matrix $\boldsymbol{W}$. The matrix $\boldsymbol{B}$ contains nonnegative basis vectors that are optimized to allow the data in $\boldsymbol{V}$ to be approximated as a nonnegative linear combination of its constituent vectors [5].

The problem now is to find B and W by minimizing the distance between V and BW so we can represent the problem as:

$$\min_{B,W} D(V, BW) \ \ subjected\ to\ all\ elements\ of\ B\ and\ W\ are + ve$$

There are many different cost functions lead to different kinds of NMF, the Euclidean Distance and KL-divergence in [3] are a good choice cost functions [12] which yield to the following optimization problem [5]:

$$\min_{B,W} D(V||BW) \qquad \text{, where:}$$

$$D(V\,||BW) = \sum_{i,j}\left(V_{i,j}\,\log\frac{V_{i,j}}{(BW)_{i,j}} - V_{i,j} + (BW)_{i,j}\right)$$

And the multiplicative update rules are computed alternating as follows:

$$B \leftarrow B.* \frac{\left(V/_{BW}\right)W^T}{1W^T}$$

$$W \leftarrow W.* \frac{B^T\left(V/_{BW}\right)}{B^T 1}$$

Where ***.**** and all divisions is an element wise matrix multiplication and divisions and ***1*** is matrix of ones with the same size of **V** [5].

3.2.2 Sparse regularization for NMF Using KL-Divergence:

The previous NMF can drop in over fitting problem and the sparse regularization is useful for many machine learning problems to prevent the over fitting [6]. The cost functions in SNMF-KL as:

$$cost = D(V||BW) + \mu|W|_1 \quad \text{.....(*)}$$

Where $\mu > 0$ *is weight sparsity penalty* , *and* $|W|_1$ is L1-norm of weights $|W|_1 = \sum_{i,j}|W_{i,j}| = \sum_{i,j} W_{i,j}$

And the multiplicative update rules are computed alternating as follows:

$$B \leftarrow B.* \frac{\left(V/_{BW}\right)W^T + 1(1W^T .*\ B)\ .*\ B}{1W^T + 1\left(\left(V/_{BW}\right)W^T\ .*\ B\right).*\ B} \qquad (1)$$

$$W \leftarrow W .* \frac{B^T\left(V/_{BW}\right)}{B^T 1 + \mu} \qquad (2)$$

Where $.*$ and all divisions is an element wise matrix multiplication and divisions and 1 is matrix of ones with the same size of V.

Algorithm Sparse NMF-KL

INPUT: Matrix $\boldsymbol{V}$ of dimensions $\boldsymbol{F\,x\,T}$ contain real positive values at all, $(\boldsymbol{F\,x\,K})$ initial basis matrix $\boldsymbol{B_{init}}$, $(\boldsymbol{K\,x\,T})$ initial weights matrix $\boldsymbol{W_{init}}$, component indices set COMP_INDS contain the indices of each source in B and W.

Output: B ϵ $R^{F\,x\,K}$ and W ϵ $R^{K\,x\,T}$ such that the cost function (*) minimized

1. ***Repeat***
2. ***for each source , get the set of source indices c in COMP_INDS***
3. ***update the coulumns c of B using W*** & ***the update roule*** **(1)**
4. ***update the rows c of W using B*** & ***the update roule*** **(2)**
5. ***end for***
6. ***Normalize B and W to l1 or l2 unit length***
7. ***update the cost using*** $(*)$
8. ***until stoping condition is met***

Fact: better NMF results can be obtained by better initialization of B and W.

Initialization B, W

for $\boldsymbol{B_{init}}$ we calculate the row mean of $\boldsymbol{V^2}$ **(Mu)** then apply the relation:

$$\boldsymbol{B_{init}} = \frac{|\boldsymbol{rand}(\boldsymbol{F \times K})|}{\mathbf{2}} + \mathbf{1}_{\boldsymbol{F*K}} \;.*\; \boldsymbol{Mu_{F*K}} \qquad (\mathbf{3})$$

$$\boldsymbol{W_{init}} = \frac{|\boldsymbol{rand}(\boldsymbol{K \times T})|}{\mathbf{2}} + \mathbf{1}_{\boldsymbol{K*T}} \qquad (\mathbf{4})$$

3.2.3 Supervised NMF Algorithm

In supervised NMF we don't need to random initialize the basis matrix B_{init} since this matrix will be a combination of speech basis and noise basis that stored in database as $\{B_{s1}, \dots, B_{si}, B_{n1}, \dots, B_{nj}\}$, get these B_s and B_n from the database for selected STFT window size and put it in one matrix B so the

number of columns K in B is $K_{s1} + K_{s2} + \cdots + K_{si} + K_{n1} + K_{n2} + \cdots + K_{nj}$ where $K_{sl} and\ K_{nt}$ are the number of columns in $B_{sl}\ and\ B_{nt}$, after setting up the matrix B we initialize W as (4) then run the Sparse NMF Algorithm to update only W with B constant, then multiply the columns $from\ 1\ to\ (K_{s1} + K_{s2} + \cdots + K_{si})$ from B with the rows of the same indices in resulted W to get the Speech spectrogram $\tilde{S}$ and multiply the columns from $(K_{s1} + K_{s2} + \cdots + K_{si}) + 1\ to\ K$ with the corresponding rows in W to get the noise spectrogram $\tilde{N}$.

3.2.4 Simi Supervised NMF Algorithm

In Simi-Supervised we construct the matrix B as three parts; the first part is the set $\{ B_{si} \}\ where\ i = 1\ ..number\ of\ speech\ basis\ of\ STFT_{wlen}\ stored\ in\ DB$ and the second one is the set $\{ B_{eq,k} \}$ where B_{eq} is basis corresponding to equilibrium sound that we used to minimize the separation error by set the noises sounds that the system justify it as speech and store it to make the system do well on these types of sounds. The third part is B_n of user defined number of columns K_n random generated matrix with the same number of rows now $K = K_s + K_{eq} + K_n$ now run the Sparse NMF and update W and only the noise part of B then to get the speech spectrogram $\tilde{S}$ by multiplying B_s by corresponding rows of W and the noise spectrogram $\tilde{N}$ by multiplying $[\ B_{eq} \mid B_n\ ]$ by corresponding rows in W.

3.2.5 Unsupervised NMF Algorithm

In this type of the NMF we don't use the dataset at all since the basis B of size $F \times (K_s + K_n) : wher\ K_s\ and\ K_n\ are\ user\ defined\ numbers;$ also W initialized as (1) and (2) then run the algorithm and compute $\tilde{S}\ and\ \tilde{N}$ as previous sections.

3.3 Filtering:

The filtering step is where we extract the speech sound form the original mixed signal. Winner filter is a good choice for human speech extracting and it is optimal in mean squired scene [5].

In general the filter can be founded by the equation:

$$H = \frac{(B_s W_s)^\rho}{(B_s W_s)^\rho + (B_n W_n)^\rho}$$

Let: $\hat{S} = B_s W_s$ expected speech spectrogram and $\hat{N} = B_n W_n$ the expected noise spectrogram.

$$H = \frac{\tilde{S}^{\rho}}{\tilde{S}^{\rho} + \tilde{N}^{\rho}}$$

Now we have a multiple choices for the parameter> 0 , for linear mask$\rho = 1$, in winner filter we have $\rho = 2$, and for hard mask filter $\rho \to \infty$ and

$$H_{hard} = round\left(\frac{\hat{S}^2}{\hat{S}^2 + \hat{N}^2}\right).$$

Now the extracted speech signal is the inversed STFT of $\ddot{S} = H .* X$ where X is the STFT representation of the input mixed signal, and the extracted noise is the inversed STFT of the matrix $\ddot{N} = (1 - H).* X$, so:

$$s(t) = istft(\ddot{S}) \qquad and \qquad n(t) = istft(\ddot{N})$$

Chapter 4
Implementation Tools Used

First prototype done using MATLAB, which provide an easy way to make fast release to test and debug your algorithms before moving to other programming languages like JAVA which we use in our work. Our work needed to fast matrices operations technique so we choose JBLAS matrix arithmetic library which provide a fast matrices operations since it is depending on parallel technics to do the operations faster.
(For more information about JBLAS see: [14])

For the STFT algorithm implementation we need ***Fast Furrier Transform*** (FFT) we used an open source **JTransforms** library which provide parallel implementation of FFT (see:[15])

For database work we use **MySQL** database and **JDBC** on **JAVA** to handle database communications.

For user interface design we use **JavaFX** to create sample and easy to use UI using **JavaFX Scene Builder** by ORACLE (for more information about JavaFX Scene Builder see:[16])

Chapter 5
Program Using and Experiments

We provide an easy to use UI to effectively interact with system functions to allow the user to test most of test cases.

We used *JavaFX Sceene Builder* to build the user interface that allow the user to select the dataset to learn the system using chosen settings such as number of components K , STFT window size , number of iterations and the type of selected set sounds. In the separation tab user can select the input mixed sound then the system will retrieve the settings from database to let the user choose algorithms settings, when the user select mixed wav file the system will automatically display the mixed sound spectrogram using log Spectrogram, after the separation process done the system will draw the separated speech and separated noise log spectrograms. The user can play the sound by click on the sound link under the sound spectrogram.

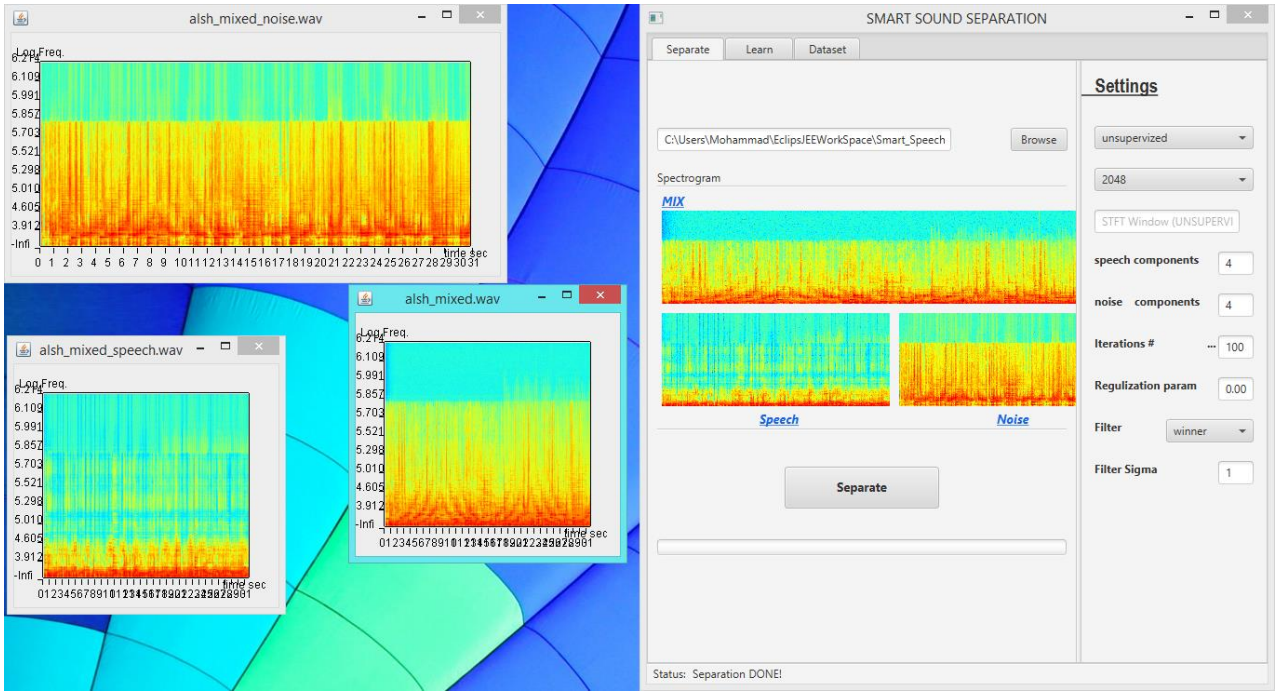

Figure 4: System UI Separation Tab

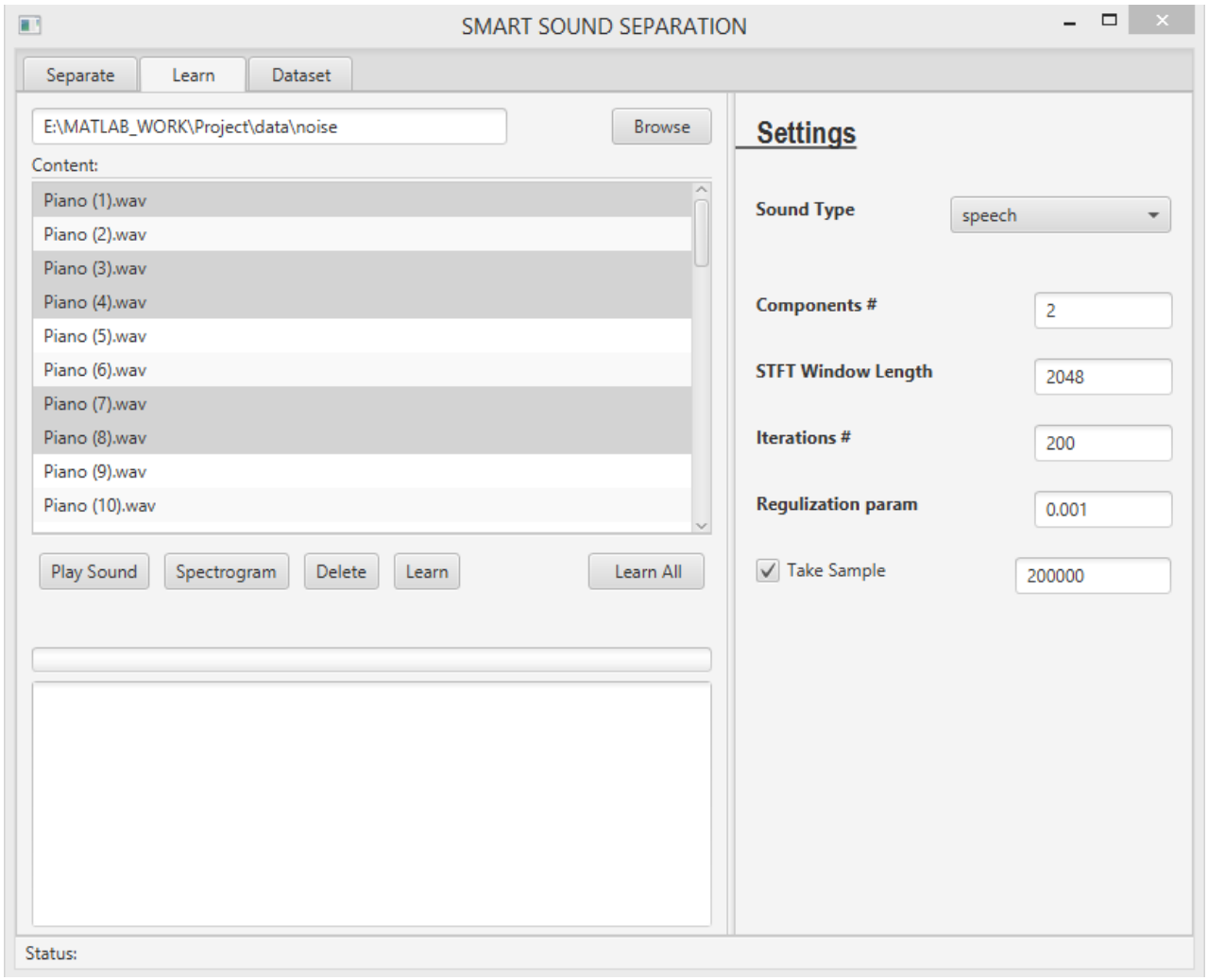

Figure 5: System UI Learn Tab

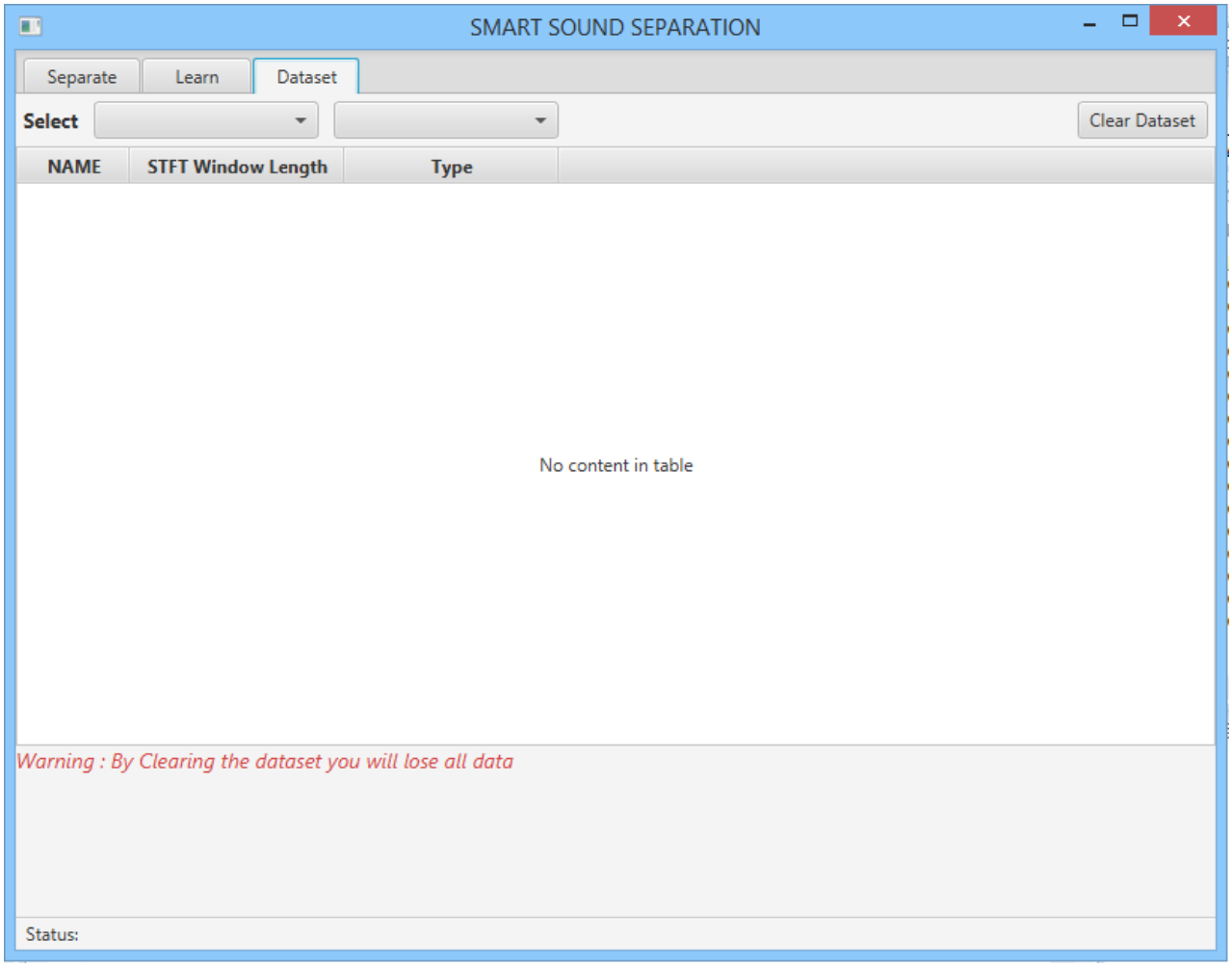

Figure 6: System UI Database Handler

APPENDIX

MATLAB Source Code

NMF Function (MATLAB CODE)

```
function [B,W,COST] = NMF_KL(V, B_init, W_init,niter,comp_ind ,switch_B,
                             switch_W , learnSource)
%%
%   NMF by multiplicative update using Kullback-Leibler divergence cost
%
%%%%%INPUT
%       V: F x T mixed spectrogram contain positive values
%       B_init: F x J basis vectors
%       W_init: J x T weights vectors
%       niter: the number of iterations of the algorithm
%       comp_ind: component distripution indecis
%       switch_B: bolean element to learn B or not
%       switch_W: bolean element to learn W or not
%       learnSource: 0 if testing phase, j learn source j (from 1:nsrc) all
%           values of other positions take 0 update only B(:,comp_ind{j})
%           and W(comp_ind{j}, :)
%
%%%%OUTPUT
%       B: out basis vectors
%       W: out wieghtes vectors
%       COST: vector of size (niter x 1) contain the cost of each iteration
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%   TEST EXAMPLE   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%     V = [1 2 3 4 5 7;3 23 4 5 8 5;5 4 2 99 8 6;  5 4 3 8 7 6;6 5 4 3 2 8]
%     B_init = abs(randn([5,2]));
%     W_init = abs(randn([2,6]));
%     ind = cell(2,1);  ind{1} = [1];   ind{2} = [2];
%
%     [B,W,COST] = NMF_KL(V,B_init,W_init,20,ind,1,1,0);
%     plot([1:20],COST,'gx')
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%   MOHAMAD SOLAIMAN ALASHKAR   Feb-27-2016
%%
if isempty(switch_B)
    switch_B = 1;
end
if isempty(switch_W)
    switch_W = 1;
end
if isempty(learnSource)
    learnSource = 0;
end

[F,T] = size(V);
num_src = size(comp_ind , 1);%comp_ind is cell element J x 1 like [ [1 2...4];
[4,3,..10]; [..].. ]

V_approx = zeros(F,T);
COST = zeros(niter,1);
```

```
W = W_init;
B = B_init;

if(learnSource ~= 0)
    for j =1:num_src
        if(j~=learnSource)
            B(:,comp_ind{j}) = B(:,comp_ind{j})*0.00000000006;
            W(comp_ind{j},:) = W(comp_ind{j},:)*0.00000000006;
        end
    end
end

%calculate V_approx
for j = 1:num_src
    S_j = B(:,comp_ind{j})*W(comp_ind{j},:);
    V_approx = V_approx(:,:) + S_j;
end

    cost = costfunction(V,V_approx);
    COST(1) = cost;
for itr = 2:niter

    %%UPDATE B
    if(switch_B)
        for j=1:num_src
            if(learnSource == 0 || (j == learnSource) )
                Kj = length(comp_ind{j});
                %Numerator and Denominator of update rule
                B_num = zeros(Kj, T);
                B_Den = zeros(Kj, T);
               B_num =(V ./ V_approx )*(W(comp_ind{j}, :))';
               B_Den = ones(size(V))* W(comp_ind{j}, :)';

               B_old = B(:,comp_ind{j});
               B(:,comp_ind{j}) = B(:,comp_ind{j}) .* (B_num ./ B_Den);

               V_approx = V_approx + ( B(:,comp_ind{j}) -
B_old)*W(comp_ind{j},:);

            end

        end
    end

    %%UPDATE W
    if(switch_W)
        for j=1:num_src
             if(learnSource == 0 || (j == learnSource) )
                Kj = length(comp_ind{j});
                %Numerator and Denominator of update rule
                W_num = zeros(F, Kj);
                W_Den = zeros(F, Kj);
               W_num =(B(:,comp_ind{j})') * (V ./ V_approx );
               W_Den =(B(:,comp_ind{j})') * ones(size(V));

               W_old = W(comp_ind{j},:);
               W(comp_ind{j},:) = W(comp_ind{j},:) .* (W_num ./ W_Den);

               V_approx = V_approx + B(:,comp_ind{j})*(W(comp_ind{j},:)-W_old);
```

```
            end
        end
    end
    cost = costfunction(V,V_approx);
    COST(itr) = cost;
    W = abs(W);
    B = abs(B);
    %%Normalization%%
    if switch_B && switch_W
        scale = sum(B,1);
        B = B ./ repmat(scale ,F,1);
        W = W .* repmat(scale',1,T);
    end

    if switch_B && ~switch_W
         scale = sum(B,1);
         B = B ./ repmat(scale ,F,1);
    end

    if ~switch_B && switch_W
        scale = sum(B,1);
        W = W .* repmat(scale',1,T);
    end

    fprintf('MU update: iteration %d of %d, cost = %f\n', itr, niter,
COST(itr));
end


end

function cost = costfunction(V,V_app)
%cost function for KL-deversion method
% cost(V || V_app) = sum{i,j}(V(i,j) * log(V(i,j)/V_app(i,j)) - V(i,j) +
%                                  V_app(i,j) )
cost = sum(sum(V .* (log( V./V_app )) - V + V_app));

end
```

SparseNMF-KL (MATLAB CODE)

```
function [B,W,COST] = SparseNMF_KL(V, B_init, W_init,niter,comp_ind ,switch_B,
switch_W , Mu)
%%
%   Sparse NMF by multiplicative update using Kullback-Leibler divergence cost
%
%%%%%INPUT
%       V: F x T mixed spectrogram contain positive values
%       B_init: F x J basis vectors
%       W_init: J x T weights vectors
%       niter: the number of iterations of the algorithm
%       comp_ind: component distripution indecis
%       switch_B: bolean element to learn B or not
%       switch_W: bolean element to learn W or not
%       Mu: the regularization parameter can tack values like
%       0.0001,0.0003,0.001,...,0.1,0.3
%
%%%%OUTPUT
%       B: out basis vectors
```

```
%       W: out wieghtes vectors
%       COST: vector of size (niter x 1) contain the cost of each iteration
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  MOHAMMAD SOLAIMAN  ALASHKAR -2016

if isempty(switch_B)
    switch_B = 1;
end
if isempty(switch_W)
    switch_W = 1;
end

[F,T] = size(V);
num_src = size(comp_ind , 1);%comp_ind is cell element J x 1 like [ [1 2...4];
[4,3,..10]; [..].. ]

V_approx = zeros(F,T);
COST = zeros(niter,1);

W = W_init;
B = B_init;


%calculate V_approx
for j = 1:num_src
    S_j = B(:,comp_ind{j})*W(comp_ind{j},:);
    V_approx = V_approx(:,:) + S_j;
end

    cost = costfunction(V,V_approx,Mu,W);
    COST(1) = cost;
for itr = 2:niter

    %%UPDATE B
    if(switch_B)
        for j=1:num_src

                Kj = length(comp_ind{j});
                %Numerator and Denominator of update rule
                B_num = zeros(Kj, T);
                B_Den = zeros(Kj, T);
               B_num =(V ./ V_approx )*(W(comp_ind{j}, :))'...
                   +  (( ( ones(size(V)) *
W(comp_ind{j},:)').*B(:,comp_ind{j})) .* B(:,comp_ind{j}));
               B_Den = ones(size(V))* W(comp_ind{j}, :)' ...
                   + ( (V ./ V_approx )* (W(comp_ind{j}, :))' .*
B(:,comp_ind{j}) ) .* B(:,comp_ind{j});

               B_old = B(:,comp_ind{j});
               B(:,comp_ind{j}) = B(:,comp_ind{j}) .* (B_num ./ B_Den);

               V_approx = V_approx + ( B(:,comp_ind{j}) -
B_old)*W(comp_ind{j},:);


        end
    end
```

```
    %%UPDATE W
    if(switch_W)
        for j=1:num_src

                 Kj = length(comp_ind{j});
                 %Numerator and Denominator of update rule
                 W_num = zeros(F, Kj);
                 W_Den = zeros(F, Kj);
                W_num =(B(:,comp_ind{j})') * (V ./ V_approx );
                W_Den =(B(:,comp_ind{j})') * ones(size(V)) + Mu;

                W_old = W(comp_ind{j},:);
                W(comp_ind{j},:) = W(comp_ind{j},:) .* (W_num ./ W_Den);

                V_approx = V_approx + B(:,comp_ind{j})*(W(comp_ind{j},:)-W_old);

        end
    end
    cost = costfunction(V,V_approx, Mu, W);
    COST(itr) = cost;
    W = abs(W);
    B=abs(B);
    %%Normalization%%
    if switch_B && switch_W
        scale = sum(B,1);
        B = B ./ repmat(scale ,F,1);
        W = W .* repmat(scale',1,T);
    end

    if switch_B && ~switch_W
         scale = sum(B,1);
         B = B ./ repmat(scale ,F,1);
    end

    if ~switch_B && switch_W
        scale = sum(B,1);
        W = W .* repmat(scale',1,T);
    end

    fprintf('MU update: iteration %d of %d, cost = %f\n', itr, niter,
COST(itr));
end


end

function cost = costfunction(V,V_app, Mu, W)
%cost function for KL-deversion method
% cost(V || V_app) = sum{i,j}(V(i,j) * log(V(i,j)/V_app(i,j)) - V(i,j) +
%                                 V_app(i,j) )
cost = sum(sum(V .* (log( V./V_app )) - V + V_app)) + (Mu* sum(W(:)));

end
```

Separate Function Using Filter

```
function [S,N,H] = Separate(B, W,comp_ind, X)
sigma = 0.99;
J = size(comp_ind,2);%number of sources
```

```
[F,T] = size(X);
Sources = zeros(F,T,J);

for j = 1:J
     Sources(:,:,j) = B(:,comp_ind{j}) * W(comp_ind{j},:);
end
H=mask_filter(Sources(:,:,1), Sources(:,:,2),8);

S = sigma*H .* X;
N = (sigma*ones(size(H)) - H) .* X;
end
```

```
function H = mask_filter(S, N, p)

H = (S.^p)./(S.^p + N.^p);

end
```

STFT.java Class

```java
package com.finalproject.smartspeechseparator.transform;

/********** STFT CLASS *************
 * METHODS :
 * 1- stft_single   calculate the short time Fourier transformation of x
 *      INPUT: x the signal 1D-array of the sound wave, wlen the window length
 *      the function using sin window function to calculate the transformation
 *
 *      OUTPUT: X 2D-matrix of size F x T where F is the number of frequency bins
 *                  and T the number of time frames
 *
 * 2- istft_single  calculate the inverse of the STFT of spectrum X
 *      INPUT: X 2D-matrix  F x T as the output of stft_single method
 *      Output: x 1D-array containing the original signals
 *
 * */

import org.jblas.ComplexDouble;
import org.jblas.ComplexDoubleMatrix;
import org.jblas.DoubleMatrix;
import org.jblas.ranges.IndicesRange;
import org.jblas.ranges.Range;
import org.jtransforms.fft.DoubleFFT_1D;

import com.finalproject.smartspeechseparator.debug.DebugLogger;

public class STFT {

    public org.jblas.ComplexDoubleMatrix stft_single(double[] x) {
        return stft_single(x,
                1024);/**
                        * wlen: window length (default: 1024 samples or 64ms at
                        * 16 kHz, which is optimal for speech source separation
                        * via binary time-frequency masking)
                        **/
    }

    public ComplexDoubleMatrix stft_single(double[] x_, int wlen) {
        DebugLogger Debug = new DebugLogger();
        double nsampl = x_.length;
        ComplexDoubleMatrix X;/**
                                * X: nbin x nfram matrix containing the STFT
                                * coefficients with nbin frequency bins and
                                * nfram time frames
                                **/
        if (wlen != 4 * Math.floor(wlen / 4)) {
            System.err.println("ERROR: The window length must be a multiple of 4."
                    + "\nMETHOD INVOCK: stft_single with wlen = " + wlen);
            return null;
        }
        ///// Computing STFT coefficients /////
        // Defining sine window
        MatrixTools mt = new MatrixTools();

        double[][] win_v = mt.makeVector(0.5, 1, wlen - 0.5);
        for (int i = 0; i < win_v[0].length; i++) {
            win_v[0][i] = Math.sin(win_v[0][i] / wlen * Math.PI);
```

```
58          }
59          DoubleMatrix win = new DoubleMatrix(win_v);
60          DoubleMatrix x = new DoubleMatrix(x_);
61          win = win.transpose();
62          x = x.transpose();
63          // V = F x T ----.> T is nfram
64          int nfram = (int) Math.ceil(nsampl / wlen * 2);
65          // % Zero-padding
66          x = DoubleMatrix.concatHorizontally(x, DoubleMatrix.zeros(1, (int) (nfram * wlen / 2 -
    nsampl)));
67          // % Pre-processing for edges
68          x = DoubleMatrix.concatHorizontally(DoubleMatrix.zeros(1, wlen / 4), x);
69          x = DoubleMatrix.concatHorizontally(x, DoubleMatrix.zeros(1, wlen / 4));
70
71          DoubleMatrix swin = DoubleMatrix.zeros((nfram + 1) * wlen / 2, 1);
72          Range cs = new IndicesRange(mt.makeIndecesRange(0, 0));
73          for (int t = 0; t < nfram; t++) {
74              Range rs = new IndicesRange(mt.makeIndecesRange(t * wlen / 2, t * wlen / 2 + (wlen
    - 1)));
75              swin.put(rs, cs, swin.get(rs, cs).add(mt.pow(win, 2)));
76          }
77
78          swin = mt.sqrt(swin.mul(wlen));
79
80          int nbin = wlen / 2 + 1;
81
82          X = ComplexDoubleMatrix.zeros(nbin, nfram); // X = F x T
83
84          for (int t = 0; t < nfram; t++) {
85              // Framing
86              Range rs = new IndicesRange(mt.makeIndecesRange(t * wlen / 2, t * wlen / 2 + wlen
    - 1));
87              DoubleMatrix den = swin.get(rs, 0);
88              DoubleMatrix numi = x.get(0, rs);
89
90              numi = numi.transpose();
91              numi = numi.mul(win);
92
93              DoubleMatrix frame_ = numi.div(den);
94              double[] frame = frame_.toArray();
95              // FFT
96              double[] fft = new double[frame.length * 2];
97              for (int i = 0; i < frame.length; i++) {
98                  fft[i] = frame[i];
99              }
100             DoubleFFT_1D d_fft_1d = new DoubleFFT_1D(frame.length);
101
102             d_fft_1d.realForwardFull(fft);
103
104             ComplexDouble val = new ComplexDouble(0, 0);
105             for (int i = 0; i < nbin; i++) {
106                 double re = fft[2 * i], im = fft[2 * i + 1];
107                 val.set(re, im);
108                 X.put(i, t, val);
109             }
110
111         }
```

```
112         return X;
113     }
114
115     public double[] istft_single(ComplexDoubleMatrix X, int nsampl) {
116         com.finalproject.smartspeechseparator.debug.DebugLogger Debug = new DebugLogger();
117
118         int nbin = X.getRows(), nfram = X.getColumns();
119         if (nbin == 2 * Math.floor((double) nbin / 2)) {
120             System.err.println("The number of frequency bins must be odd.");
121             return null;
122         }
123
124         int wlen = 2 * (nbin - 1);
125
126         // %%% Computing inverse STFT signal %%%
127         // Defining sine window
128         MatrixTools mt = new MatrixTools();
129
130         double[][] win_v = mt.makeVector(0.5, 1, wlen - 0.5);
131         for (int i = 0; i < win_v[0].length; i++) {
132             win_v[0][i] = Math.sin(win_v[0][i] / wlen * Math.PI);
133         }
134         DoubleMatrix win = new DoubleMatrix(win_v);
135
136         // % Pre-processing for edges
137         DoubleMatrix swin = DoubleMatrix.zeros(1, (nfram + 1) * wlen / 2);
138         Range cs = new IndicesRange(mt.makeIndecesRange(0, 0));
139
140         for (int t = 0; t < nfram; t++) {
141             Range rs = new IndicesRange(mt.makeIndecesRange(t * wlen / 2, t * wlen / 2 + wlen
    - 1));
142             swin.put(rs, cs, swin.get(rs, cs).add(mt.pow(win, 2)));
143         }
144         swin = mt.sqrt(swin.div(wlen));
145         DoubleMatrix x = DoubleMatrix.zeros((nfram + 1) * wlen / 2, 1);
146
147         for (int t = 0; t < nfram; t++) {
148             // IFFT
149             ComplexDoubleMatrix fframe = X.getColumn(t);
150             fframe = ComplexDoubleMatrix.concatVertically(fframe,
151                     X.get(mt.makeIndecesRange(wlen / 2 - 1, 1, -1), t).conj());
152             double[] fframe_ = new double[fframe.length * 2];
153
154             DoubleFFT_1D d_fft_1d = new DoubleFFT_1D(fframe.length);
155             int ctr = 0;
156             for (int i = 0; i < fframe.getRows(); i++) {
157                 fframe_[ctr] = fframe.getReal(i);
158                 fframe_[ctr + 1] = fframe.getImag(i);
159                 ctr += 2;
160             }
161
162             d_fft_1d.complexInverse(fframe_, true);
163             double[] frame = new double[fframe_.length / 2];
164             ctr = 0;
165             // getting the real values
166             for (int i = 0; i < fframe_.length; i += 2) {
167                 frame[ctr] = fframe_[i];
```

```
168                 ctr++;
169             }
170             // Overlap-add
171             DoubleMatrix factorToAdd = new DoubleMatrix(frame).transpose().mul(win)
172                     .div(swin.get(mt.makeIndecesRange(t * wlen / 2, t * wlen / 2 + wlen -
    1)));
173             x.put(mt.makeIndecesRange(t * wlen / 2, t * wlen / 2 + wlen - 1),
174                     x.get(mt.makeIndecesRange(t * wlen / 2, t * wlen / 2 + wlen - 1),
    0).add(factorToAdd));
175         }
176         // Truncation
177         double[] x_ = new double[(wlen / 4 + nsampl) - (wlen / 4 + 1) + 1];
178         int ctr = 0;
179         for (int i = wlen / 4; i < wlen / 4 + nsampl; i++) {
180             x_[ctr] = x.get(i);
181             ctr++;
182         }
183         return x_;
184     }
185
186 }
187
```

NMF-KL Java Class

NMF_KL.java

```java
package com.finlproject.smartspeechseparator.learn;

import java.io.File;
/**
 * %%
* 	%     NMF by multiplicative update using Kullback-Leibler divergence cost
* 	%
* 	%%%%%INPUT
* 	%       V: F x T mixed spectrogram contain positive values
* 	%       B_init: F x J basis vectors
* 	%       W_init: J x T weights vectors
* 	%       niter: the number of iterations of the algorithm
* 	%       comp_ind: component distripution indecis
* 	%       switch_B: bolean element to learn B or not
* 	%       switch_W: bolean element to learn W or not
* 	%       Mu: the regularization parameter can tack values like
* 	%       0.0001,0.0003,0.001,...,0.1,0.3
* 	%
* 	%%%%OUTPUT
* 	%       B: out basis vectors
* 	%       W: out wieghtes vectors
* 	%       COST: vector of size (niter x 1) contain the cost of each iteration
* 	%
* 	%
* 	%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
* 	%%      MOHAMMAD SOLAIMAN AL-ASHKAR 9-4-2016
***/
public class NMF_KL implements Runnable{


    //INPUT PARAMETERS
    private DoubleMatrix V,B_init,W_init;
    private int niter = 500;
    private Cell comp_ind;
    boolean switch_B=true,switch_W=true;
    private double Mu = 0.001;
    STATE state;
    int numOfSupervisedComponents = 1;

    //OUTPUT PARAMETERS
    DoubleMatrix B, W;
    double[] COST;

    private MatrixTools tool;
    private DebugLogger logger;
    public NMF_KL(DoubleMatrix v, DoubleMatrix b_init, DoubleMatrix w_init, int niter, Cell comp_ind, boolean switch_B,
            boolean switch_W, double mU) {
        V = v;
        B_init = b_init;
        W_init = w_init;
        this.niter = niter;
        this.comp_ind = comp_ind;
        this.switch_B = switch_B;
        this.switch_W = switch_W;
        Mu = mU;
```

```
63
64          tool = new MatrixTools();
65          logger = new DebugLogger(new File("NMF_log.txt"));
66      }
67
68
69      public NMF_KL(DoubleMatrix v, DoubleMatrix b_init, DoubleMatrix w_init, int niter, Cell
    comp_ind, boolean switch_B,
70              boolean switch_W, double mu, STATE state, int numOfSupervisedComponents,
    DoubleMatrix b, DoubleMatrix w,
71              double[] cOST, MatrixTools tool, DebugLogger logger) {
72          super();
73          V = v;
74          B_init = b_init;
75          W_init = w_init;
76          this.niter = niter;
77          this.comp_ind = comp_ind;
78          this.switch_B = switch_B;
79          this.switch_W = switch_W;
80          Mu = mu;
81          this.state = state;
82          this.numOfSupervisedComponents = numOfSupervisedComponents;
83          B = b;
84          W = w;
85          COST = cOST;
86          this.tool = tool;
87          this.logger = logger;
88      }
89
90      public NMF_KL(DoubleMatrix v, DoubleMatrix b_init, DoubleMatrix w_init, int niter, Cell
    comp_ind, boolean switch_B,
91              boolean switch_W, double mU, STATE state) {
92          V = v;
93          B_init = b_init;
94          W_init = w_init;
95          this.niter = niter;
96          this.comp_ind = comp_ind;
97          this.switch_B = switch_B;
98          this.switch_W = switch_W;
99          Mu = mU;
100         this.state = state;
101         tool = new MatrixTools();
102         logger = new DebugLogger(new File("NMF_log.txt"));
103     }
104
105     public STATE getState() {
106         return state;
107     }
108
109
110     public void setState(STATE state) {
111         this.state = state;
112     }
113
114
115     public double getMu() {
116         return Mu;
```

```
117     }
118
119
120     public void setMu(double mu) {
121         Mu = mu;
122     }
123
124
125     public NMF_KL(DoubleMatrix v, DoubleMatrix b_init, DoubleMatrix w_init, Cell comp_ind) {
126         V = v;
127         B_init = b_init;
128         W_init = w_init;
129         this.comp_ind = comp_ind;
130
131         tool = new MatrixTools();
132         logger = new DebugLogger(new File("NMF_log.txt"));
133     }
134
135     public int getNumOfSupervisedComponents() {
136         return numOfSupervisedComponents;
137     }
138
139
140     public void setNumOfSupervisedComponents(int numOfSupervisedComponents) {
141         this.numOfSupervisedComponents = numOfSupervisedComponents;
142     }
143
144
145     public DebugLogger getDebugTool() {
146         return logger;
147     }
148
149
150     public void setDebugTool(DebugLogger debugTool) {
151         this.logger = debugTool;
152     }
153
154
155
156     public DoubleMatrix getB() {
157         return B;
158     }
159
160
161     public DoubleMatrix getW() {
162         return W;
163     }
164
165
166     public double[] getCOST() {
167         return COST;
168     }
169
170
171     public int getNiter() {
172         return niter;
173     }
```

```
174
175
176     public void setNiter(int niter) {
177         this.niter = niter;
178     }
179
180
181     public void RunSupervised_NMF(){
182         run_NMF(STATE.supervised);
183     }
184
185     public void RunSimiSupervised_NMF(){
186         run_NMF(STATE.simisupervised);
187     }
188
189     public void RunUnupervised_NMF(){
190         run_NMF(STATE.unsupervised);
191     }
192
193     private void run_NMF(STATE state){
194         int F = V.getRows(), T = V.getColumns(), num_src = comp_ind.size();
195         DoubleMatrix V_approx = DoubleMatrix.zeros(F, T);
196         COST = new double[niter];
197
198         W = W_init; B = B_init;
199         //calculate V_approx
200         for(int j = 0; j<num_src;j++){
201             DoubleMatrix S_j = B.getColumns(comp_ind.get(j)).mmul(W.getRows(comp_ind.get(j)));
202             V_approx = V_approx.add(S_j);
203         }
204
205         COST[0] = costfunction(V, V_approx, Mu, W);
206
207         int j_start =0;
208         switch( state){
209             case supervised:{
210                 j_start = num_src;
211                 break;
212             }
213             case simisupervised:{
214                 j_start = numOfSupervisedComponents;
215                 break;
216             }
217             default:
218                 j_start = 0;
219                 break;
220         }
221         //Run NMF core
222         for(int itr = 1; itr<niter;itr++){
223
224             //UPDATE B
225             if(switch_B){
226                 for(int j=j_start; j<num_src;j++){
227                     int Kj = comp_ind.get(j).length;
228
229                     //Numerator and Denominator of update rule
230                     DoubleMatrix B_num = DoubleMatrix.zeros(Kj, T),
```

```
231                             B_Den = DoubleMatrix.zeros(Kj, T);
232
233                     B_num =
    V.div(V_approx).mmul(W.getRows(comp_ind.get(j)).transpose()).add(
    ((DoubleMatrix.ones(F,T).mmul(W.getRows(comp_ind.get(j)).transpose()))).mul(B.getColumns(comp_i
    nd.get(j)))).mul(B.getColumns(comp_ind.get(j))) );
234
235                     B_Den =(
    DoubleMatrix.ones(F,T).mmul(W.getRows(comp_ind.get(j)).transpose())).add(
    ((V.div(V_approx)).mmul(W.getRows(comp_ind.get(j)).transpose()).mul(B.getColumns(comp_ind.get(j
    )))).mul(B.getColumns(comp_ind.get(j))) );
236
237                     DoubleMatrix B_old = B.getColumns(comp_ind.get(j));
238
239                     //In matlab --|>
240                     // B(:,comp_ind{j}) = B(:,comp_ind{j}) .* (B_num ./ B_Den);
241                     DoubleMatrix replacement = B.getColumns(comp_ind.get(j)
    ).mul(B_num.div(B_Den));
242                     int[] inds = comp_ind.get(j);
243                     int ctr=0;
244                     for(int in : inds){
245                         B.putColumn(in,  replacement.getColumn(ctr++));
246                     }
247
248                     V_approx = V_approx.add(  ( B.getColumns(comp_ind.get(j)).sub(B_old)
    ).mmul(W.getRows(comp_ind.get(j)))    );
249                 }
250             }
251
252             //UPDATE W
253             if(switch_W){
254                 for(int j=0; j<num_src;j++){
255
256                     int Kj = comp_ind.get(j).length;
257
258                     //Numerator and Denominator of update rule
259                     DoubleMatrix W_num = DoubleMatrix.zeros(Kj, T),
260                             W_Den = DoubleMatrix.zeros(Kj, T);
261
262                     W_num = (B.getColumns(comp_ind.get(j)).transpose()
    ).mmul(V.div(V_approx));
263                     W_Den =
    ((B.getColumns(comp_ind.get(j)).transpose()).mmul(DoubleMatrix.ones(F, T))).add(Mu);
264
265                     DoubleMatrix W_old = W.getRows(comp_ind.get(j));
266
267
268                     //In Matlab --|>
269                     //W(comp_ind{j},:) = W(comp_ind{j},:) .* (W_num ./ W_Den);
270                     DoubleMatrix replacement =
    W.getRows(comp_ind.get(j)).mul(W_num.div(W_Den));
271                     int[] inds = comp_ind.get(j);
272                     int ctr=0;
273                     for(int in : inds){
274                         W.putRow(in, replacement.getRow(ctr++));
275                     }
276
```

```
277                     V_approx = V_approx.add(    B.getColumns(comp_ind.get(j)).mmul(
    W.getRows(comp_ind.get(j)).sub(W_old) )    );
278                 }
279             }
280
281             COST[itr] = costfunction(V, V_approx, Mu, W);
282
283             //    %%Normalization%%
284             DoubleMatrix scale = B.columnSums();
285             B = B.div(scale.repmat(F, 1));
286             W = W.mul(scale.transpose().repmat(1, T));
287
288             if(Double.isNaN(COST[itr])){
289                 B = null;
290                 W = null;
291                 COST = null;
292                 return;
293             }
294
295             logger.log("MU update: iteration  "+(itr + 1)+"  of  "+niter+",  cost =
    "+COST[itr]);
296         }
297     }
298
299     private double costfunction(DoubleMatrix V, DoubleMatrix V_approx, double Mu, DoubleMatrix
    W){
300
301         //System.out.println( V_approx.get(10));
302         return V.mul(    tool.log(V.div(V_approx))).sub(V).add(V_approx).sum() + (Mu *
    W.sum());
303     }
304
305
306     @Override
307     public void run() {
308         this.run_NMF(this.state);
309     }
310 }
311
312
```

You can find the full source on GetHub:

https://github.com/mohammadsolaiman/Smart_Sound_Separation_System.git

SSH: git@github.com:mohammadsolaiman/Smart_Sound_Separation_System.git

References:

[1]. *Bhiksha Raj*[1,3], *Tuomas Virtanen*[2], *Sourish Chaudhuri*[3], *Rita Singh*[3] [1]Walt Disney Research, Pittsburgh PA, USA, [2]Department of Signal Processing, Tampere University of Technology, Finland [3]Carnegie Mellon University, Pittsburgh PA, USA, **"NON-NEGATIVE MATRIX FACTORIZATION BASED COMPENSATION OF MUSIC FOR AUTOMATIC SPEECH RECOGNITION"**

[2]. *Barcelona, Spain, 2010 7th Sound and Music Computing Conference,.* **C. Sch¨orkhuber and A. Klapuri, "Constant-Q Transform toolbox for music processing,"**

[3]. Daniel D. Lee* *BelJ Laboratories Lucent Technologies Murray Hill, NJ 07974, H. Sebastian Seung*t tDept. of Brain and Cog. Sci. Massachusetts Institute of Technology Cambridge, MA 02138 , URL: http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

Algorithms-For-Non-negative-Matrix-Factorization

[4]. *Derry Fitzgerald Dublin Institute of Technology, derry.fitzgerald@dit.ie Matt Cranitch Cork Institute of Technology, matt.cranitch@dit.ie Eugene Coyle Dublin Institute of Technology, eugene.coyle@dit.e* **"Non-negative Tensor Factorisation for Sound Source Separation"**

[5]. *Emad M. Grais and Hakan Erdogan Faculty of Engineering and Natural Sciences, Sabanci University, Orhanli Tuzla, 34956, Istanbul. Email: grais@su.sabanciuniv.edu , haerdogan@sabanciuniv.edu* **"SINGLE CHANNEL SPEECH MUSIC SEPARATION USING NONNEGATIVE MATRIX FACTORIZATION AND SPECTRAL MASKS"**

[6]. *Felix Johannes Weninger* **Intelligent Single-Channel Methods for Multi-Source Audio Analysis**

[7]. **J. C. Brown,** *Journal of the Acoustic Society of America, vol. 90, pp. 60,66, 1991.*", **"Calculation of a Constant Q spectral transform**,"

[8]. *Kevin W. Wilson∗ , Bhiksha Raj∗ , Paris Smaragdis† , Ajay Divakaran∗ ∗ Mitsubishi Electric Research Lab, Cambridge, MA † Adobe Systems, Newton, MA* **SPEECH DENOISING USING NONNEGATIVE MATRIX FACTORIZATION WITH PRIORS**

[9]. Mengjie, Z., (2001) *Overview of speech recognition and related machine learning techniques*, Technical report. Retrieved December 10, 2004 from http://www.mcs.vuw.ac.nz/comp/Publications/archive/CS-TR-01/CS-TR-01-15.pdf

[10]. ***Nick Collins*** **Introduction to Computer Music**

[11]. *PhD Thesis,Rajesh Jaiswal,Audio Research Group,School of Electrical Engineering Systems,Dublin Institute of Technology, Dublin, Ireland, October 2013* " **Non-negative Matrix Factorization based Algorithms to cluster Frequency Basis Functions for Monaural Sound Source Separation"**

[12]. **SM. R** *Journal Acoustic Society America, vol. 34 pp. 1679,1683, 1962.*"**. Schroeder and B. S. Atal, "Generalized short-time power spectra and autocorrelation functions,"**

[13]. http://research.ics.aalto.fi/ica/links.shtml

[14]. Jblas.org/javadoc/index.html

[15]. https://sites.google.com/site/piotrwendykier/software/jtransforms

[16]. https://docs.oracle.com/javafx/scenebuilder/1/user_guide/jsbpub-user_guide.htm