Published in Image Processing On Line on YYYY–MM–DD.
ISSN 2105–1232 © YYYY IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials, software, datasets and online demo at
http://www.ipol.im/

PREPRINT July 2, 2012

Audio denoising algorithm with block thresholding

Eric Martin,
Marie de Masson d'Autume,
Christophe Varray

Abstract

This paper presents the analysis and the implementation of the method described in Guoshen Yu, Stéphane Mallat and Emmanuel Bacry *Audio Denoising by Time-Frequency Block Thresholding*. A filter of the Fourier coefficient matrix is used to denoise an audio signal. To prevent artefacts called "musical noise", a non-diagonal processing like a block thresholding is needed. Block thresholding algorithm parameters are chosen adaptively by minimizing a Stein estimation of the risk. Numerical experiments demonstrate the performance and robustness of this procedure.

Source Code

The ANSI C source code implementation of these algorithms will be soon accessible on the webpage Ipol[1].

1 Introduction

Audio signals are often contaminated by background noise and buzzing or humming noise from audio equipments. This article focuses on audio signals corrupted with Gaussian white noise, which is specially hard to remove because it is located in all frequencies. This paper describes Guoshen, Mallat and Bacry's non diagonal audio denoising algorithm [1].

Diagonal coefficient audio denoising algorithms attenuate the noise by processing each window Fourier, with empirical Wiener filter [6] or thresholding operators [7]. These algorithms create isolated time-frequency structures that are perceived as a musical noise [8].

For non stationary signals the Fourier transform is computed over widowed signals using the short time Fourier transform (STFT). It shows a time-frequency decomposition (Figures 1 and 2) like a music score.

To denoise the signal, we apply a filter by convolution. In a block thresholding, all the coefficients of a block are treated with a common threshold, depending on the block. In order to analyse the neighbourhood of a point the Stein Unbiased Risk Estimate (SURE) theorem [3] is applied. This theorem permits to minimize Stein risk estimator.

The paper first describes the STFT algorithm. Section 3 details the Wiener filter. Section 4 introduces the different steps of block thresholding algorithm. Finally, the results from different tests and some improvements on this algorithm are presented in section 5.

[1] http://www.ipol.im

2 Passage to the time-frequency field

2.1 Short Time Fourier Transform or the windowed Fourier transform (STFT)

Definitions and properties

The STFT decomposes a signal into time-frequency atoms $g_{j,k}(n) = w(n - jq)\exp\left(\frac{2i\pi kn}{W}\right)$ where j and k are respectively time and frequency indices and $(w(n))_{1\leq n\leq W}$ is a real window function [2]. The integer q characterizes the superposition factor between the windows. The STFT schema is

$$STFT : f \mapsto \langle f|\overline{g_{j,k}}\rangle = \sum_{n=1}^{W} f(n).w(n-jq)\exp\left(\frac{-2i\pi kn}{W}\right).$$

In practice, the computation of this coefficient is made through the discrete Fourier transform, computed with the algorithm of the fast Fourier transform. Denote $\phi : \mathbb{R}^W \to \mathbb{R}^W$ the discrete Fourier transform [5]. Thus, $\forall y \in \mathbb{R}^W$ and $\forall n = 1...W$

$$Y_n = [\phi(y)]_n = \sum_{k=1}^{W} y_k.exp\left(\frac{-2i\pi(k-1)(n-1)}{W}\right)$$

and the inverse discrete Fourier transform is defined by

$$y_n = \left[\phi^{-}1(Y)\right]_n = \frac{1}{N}\sum_{k=1}^{W} Y_k.exp\left(\frac{2i\pi(k-1)(n-1)}{W}\right).$$

To compute the STFT a short extract of the signal is taken, using a window function. The discrete Fourier transform is computed and the result is stored in a matrix. In Figure 1, we can see the spectrogram of a clarinet music extract by Mozart. A spectrogram is a representation of the log of the modulus of the matrix computed by STFT algorithm.

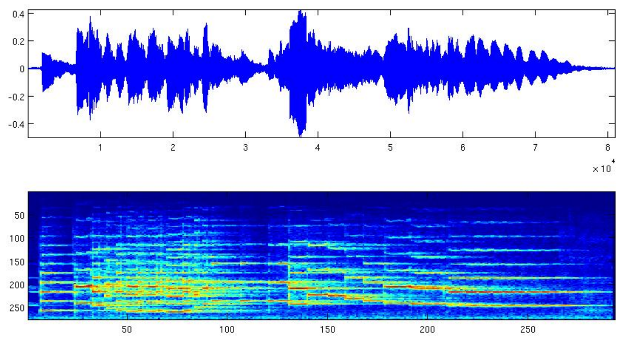


Figure 1: Spectrogram of a clarinet music extract

The Hanning window, defined on $]-1,1[$ by

$$\begin{array}{rrcl} h: & \mathbb{R} & \to & [0,1] \\ & x & \mapsto & \begin{cases} 0 & \text{if } |x| \geq 1 \\ \frac{1+\cos(\pi x)}{2} & \text{if } x \in [-1,1] \end{cases} \end{array}$$

is used in this block thresholding algorithm. The window is C^1. This is essential when the signal is reconstructed from the matrix of the coefficients. Indeed, if the window function is an rectangular

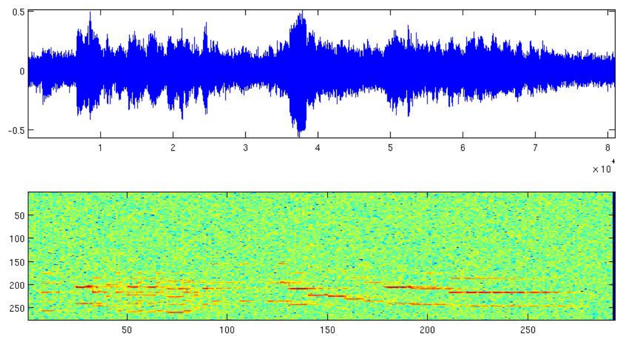


Figure 2: Spectrogram of a noisy clarinet music extract

function, after computation of the matrix, application of some filter and reconstruction, we will have a sharp at each spot where the support of the windows starts and ends. That is why C^1 windows are used and superposed.

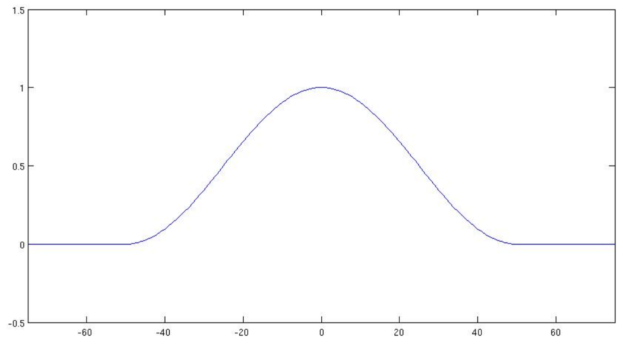


Figure 3: The Hanning window

A long function can be covered by several windows, with a superposition of half of the window's support length as the figure 4 shows.

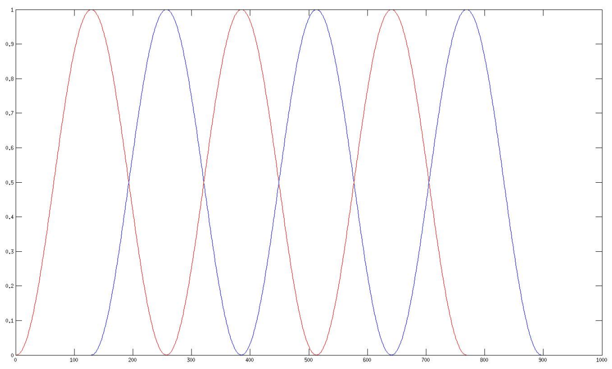


Figure 4: Superposition of half of the window's support length

Discretized windows

The size of the windows is $W = 2q + 1$ with $q \in \mathbb{N}$ (in practice, 50ms long windows are used, which corresponds to $W = 551$ samples for a 11kHz sample rate).

The discretized window covers $\{-q, ..., q\}$ and :

$$\forall n \in \{-q, ..., q\}\ ,\ w(n) = \frac{1 + \cos\left(\frac{\pi n}{q}\right)}{2}$$

Covering signal with Hanning windows

Let $y = (y_j)_{1 \leq j \leq N} \in \mathbb{R}^N$ be a discretized signal. It can be covered using K windows with a superposition of q points, ie the next window is obtain though a translation of q points to the right. Notice that $K \simeq \lfloor 2\frac{N}{W} \rfloor$.

In details :

- From the first window
$$w_1(j) = w(j - q) = \frac{1 + \cos\left(\frac{\pi}{q}(j - q)\right)}{2}$$
on the interval $\{0, 2q\}$ and 0 elsewhere. For computation considerations, a window function is considered as a signal on $\mathbb{R}^N$, with support $\{0, ..., 2q\} = \{0, ..., W - 1\}$
- Then, with a translation of q points to the right, the second window's support is in the interval $\{q, 3q\}$.
- And so on the k^{th} window's support is in the interval $\{(k-1)q, (k+1)q\}$ and the expression of the k^{th} window is :
$$w_k(j) = w(j - kq) = \frac{1 + \cos\left(\frac{\pi t}{q} - k\pi\right)}{2}$$

Proposition 1 (Reconstruction property) *For all $n \in \{q, ..., N - q\}$ (everywhere except the edges)*

$$\sum_{k=1}^{K} w_k(n) = 1.$$

Windowed signal

For $y \in \mathbb{R}^N$, we note $\tilde{y}_k$ the signal windowed by the k^{th} window. Thus, $\forall n \in \{0, ...N - 1\}\ ,\ \tilde{y}_k(n) = w_k(n)y(n)$.

Let $n \in \{q, ..., N - q\}$ (everywhere except the edge). We have $\sum_{k=1}^{K} \tilde{y}_k(n) = \sum_{k=1}^{K} y(n)w_k(n) = y(n)\sum_{k=1}^{K} w_k(n) = y(n)$.

This justifies that the inverse STFT reconstructs the right signal (cf algorithms).

2.2 Implementation [1]

STFT (cf algorithm 1)

Input :

- f : Input 1D signal, array of known length.
- $time_{win}$: window size in time (in milliseconds).

- $f_{sampling}$: signal sampling frequency in Hz.

Output :

- $STFT_{coef}$: Fourier coefficients matrix.

Algorithm 1: STFT

```
1  Size of windows in number of points : $size_{win} \leftarrow round\left(\frac{time_{win} * f_{sampling}}{1000}\right)$;
2  **if** $size_{win}$ *is even* **then**
3      $size_{win} \leftarrow size_{win} + 1$;
4  **end**
5  $halfsize_{win} \leftarrow \frac{size_{win}+1}{2}$;
6  Make a Hanning window of size $size_{win}$ : $w_{hanning} \leftarrow MakeHanning(size_{win})$;
7  Number of needed windows : $Nb_{win} \leftarrow floor\left(\frac{length(f)*2}{size_{win}}\right)$;
8  Initialize $STFT_{coef}$ : $STFT_{coef} \leftarrow zeros(size_{win}, Nb_{win} - 1))$;
9  **foreach** $k$ *from* 1 *to* $Nb_{win} - 2$ **do**
10     Compute the windowed function $f_{win}$;
11     Keep just the nonzero portion;
12     Compute the discrete Fourier transform of this signal;
13     Store it in the $k^{th}$ column of $STFT_{coef}$;
14 **end**
15 Return $SFTF_{coef}$;
```

Inverse STFT (cf algorithm 2)

Inputs :

- $STFT_{coef}$: Fourier coefficient matrix.
- $time_{win}$: window size in time (in milliseconds).
- $f_{sampling}$: signal sampling frequency in Hz.
- $length\_f$: length of the signal f.

Output :

- f_{rec} : reconstructed signal, array of length $length\_f$.

2.3 Remarks

The superposition can be more than 2. Indeed, a logarithmic factor of redundancy can be defined. For k the factor of redundancy, the superposition is 2^k and the algorithms have to be changed a little.

- For the construction from the signal to the coefficients matrix.
  - The column number in the matrix is doubled $k-1$ times.
  - The translation in each step is smaller (translation of $\frac{1}{2^{k-1}}$ of the size of the windows' support to the right).

Algorithm 2: Inverse STFT

```
1  Size of windows in number of points : $size_{win} \leftarrow round\left(\frac{time_{win} * f_{sampling}}{1000}\right)$;
2  if $size_{win}$ is even then
3      $size_{win} \leftarrow size_{win} + 1$;
4  end
5  $halfsize_{win} \leftarrow \frac{size_{win}+1}{2}$;
6  Make a Hanning window of size $size_{win}$ : $w_{hanning} \leftarrow MakeHanning(size_{win})$;
7  Number of needed windows : $Nb_{win} \leftarrow floor\left(\frac{length\_f*2}{size_{win}}\right)$;
8  Initialize $f_{rec}$ : $f_{rec} \leftarrow zeros(length\_f)$;
9  foreach $j$ from 1 to $Nb_{win} - 1$ do
10     Compute the inverse Fourier transform of the $j^{th}$ column of $STFT_{coef}$ :
       $f_{win_{rec}} \leftarrow ift(STFT_{coef}(:, j))$;
11     Add the result to $f_{rec}$ in the right spot :
       $f_{rec}\left((j-1)halfsize_{win} + 1 : (j-1)halfsize_{win} + size_{win}\right) \longleftarrow$
       $f_{rec}\left((j-1)halfsize_{win} + 1 : (j-1)halfsize_{win} + size_{win}\right) + f_{win_{rec}}$;
12 end
13 return $f_{rec}$;
```

- For the reconstruction from the matrix of the coefficients to the signal
  - The reconstruction is the same.
  - Divide the result by 2^{k-1}

3 Wiener Filter

The ideal Wiener filter [1] [6] is the optimal time invariant linear denoising filter, using a matrix of convolution, assuming the Fourier transform modulus known.

Let f be an audio signal. Recall y the noisy signal and η the white gaussian noise. So

$$y = f + \eta \in \mathbb{R}^W$$

and in the Fourier domain

$$\hat{y}_k = \hat{f}_k + \hat{\eta}_k \in \mathbb{C}$$

with $\mathbb{E}[\eta] = 0$ and $\mathbb{E}[\eta^2] = \sigma^2$. Each Fourier coefficients is multiplied by a factor a_k. Let's find the best factors. Let

$$\hat{\tilde{f}}_k = a_k \hat{y}_k$$

be the denoised signal in Fourier domain, with the filter (a_k) minimizing Linear Minimal Mean Square Error (LMMSE). The minimum square error is given by

$$\begin{aligned} \text{MSE} &= \mathbb{E}\left[\textstyle\sum_k |\hat{\tilde{f}}_k - \hat{f}_k|^2\right] \\ &= \mathbb{E}\left[\textstyle\sum_k |a_k(\hat{f}_k + \hat{\eta}_k) - \hat{f}_k|^2\right] \\ &= \textstyle\sum_k |(a_k - 1)\hat{f}_k|^2 + |a_k|^2\sigma^2 \end{aligned}$$

The differentiation with respect to a_k gives

$$\frac{\partial \text{MSE}}{\partial a_k} = 2(a_k - 1)|\hat{f}_k|^2 + 2|a_k|\sigma^2.$$

The LMMSE is minimal for

$$a_k = \left(\frac{|\hat{f}_k|^2}{\sigma^2 + |\hat{f}_k|^2}\right)_+ .$$

This filter is ideal but impractical as it needs the original Fourier transform modulus to be known. That is why an approximation of $|\hat{f}_k|^2$, called an oracle, has to be find. The empirical Wiener filter uses the equality

$$\mathbb{E}[|\hat{y}_k|^2] = \sigma^2 + |\hat{f}_k|^2$$

to estimate $|\hat{f}_k|^2$ by $|\hat{y}_k|^2 - \sigma^2$. It gives the empirical Wiener filter [1] defined by

$$a'_k = \left(\frac{|\hat{y}_k|^2 - \sigma^2}{|\hat{y}_k|^2}\right)_+ .$$

Using this method, the denoised signal presents a lot of artefacts like musical noise (The Wiener filters are diagonal methods). The method of the block thresholding enables to reduce considerably these actefacts. The block thresholding method consists on the research of a good oracle whose parameters depend only on the noisy signal.

4 Block thresholding [1]

The STFT coefficients matrix is partitioned in blocks. The coefficients of each block are treated with a common threshold. The purpose of the method is to test different partitions and keep the best [1].

4.1 Principle of the method

4.1.1 Passage to time-frequency domain and settings

The first step consists in computing the STFT coefficients matrix. An audio signal has only real coefficients. So, this matrix has a symmetry between negative and positive frequencies.
All along the denoising, we work on two matrices which have the same size as the STFT coefficients matrix. The first one, $AttenFactorMap$ contains the attenuation factors and the second one, $flag_{depth}$ contains the subdivision representation.

Sigma changing

The noise characteristics are changed during the passage from time field to time-frequency field. It is still Gaussian (for all frequencies, the noise follow a centred normal law) but σ^2 changes. Consider the discrete Fourier transform of the windowed noise. The Fourier coefficients of the noise is given by

$$\hat{\eta}_k = \frac{1}{\sqrt{W}} \sum_n w_n \eta_n \exp(\frac{-2i\pi kn}{W})$$

Thus

$$\begin{array}{rcl} \mathrm{Var}(\hat{\eta}_k) & = & \frac{1}{W}\mathrm{Var}\left[\sum_n w(n)\eta_n \exp(\frac{-2i\pi kn}{W})\right] \\ & = & \frac{1}{W}\sum_n \sigma^2 w(n)^2 \\ & = & \sigma^2(\frac{1}{W}\sum_n \frac{1}{2}(1+\cos(\frac{2\pi n}{W}))^2) \\ & = & 0.375\sigma^2. \end{array}$$

4.1.2 Finding the oracle

The coefficients matrix is partitioned into macro-blocks (Figure 5) and as the signal is real, the matrix presents a symmetry, thus it is enough to only treat the negative frequencies. The frequency 0 is treated separately.

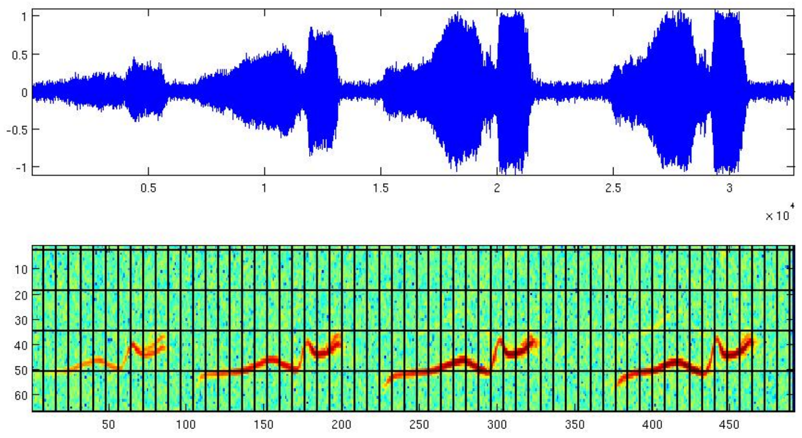


Figure 5: Cutting in macroblocks

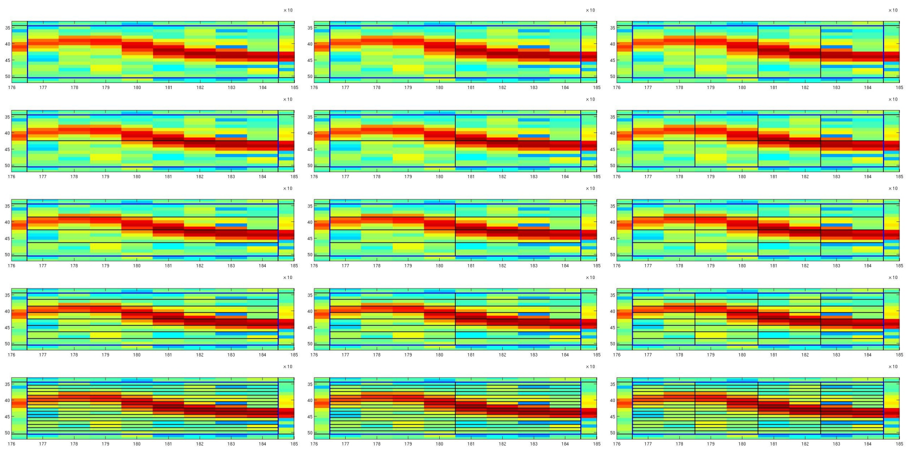


Figure 6: The 15 possible subdivisions

Case of the zero frequency - the first points

For the zero frequencies, we treat the points from the beginning to the end eight by eight (blocks 1x8):

- Computation of the attenuation coefficient for the block i of size 1x8:

$$a_i = \left(1 - \frac{\lambda}{\hat{\xi}_i + 1}\right)_+ \tag{1}$$

with

$$\hat{\xi}_i = \frac{\overline{Y_i^2}}{\sigma^2} - 1$$

where $\overline{Y_i^2}$ is the empirical mean on the block i. The real λ is a parameter depending on the block size. It comes from a generalization of the empirical Wiener filter [1].

- We filled $AttenFactorMap$ and $flag_{depth}$ in the right spot with the information computed.

Lambda matrix

As for Guoshen Yu [1], when the noise is a white Gaussian one with constant σ as a variance in a block, the computation of an upper bound of the risk in a block is split in two terms. One for the variance and another for the bias.

The real λ controls the variance term which is due to the noise variation. It is computed with the following expression:

$$P(\overline{\epsilon}^2 > \lambda\sigma^2) < \delta$$

In this expression, δ is a parameter such as, with $\delta = 10^{-3}$, musical noises are barely audible.

The blocks inside macro-blocks are rectangles. Their sizes are $L_i * W_i$ where L_i and W_i are respectively the length in time and the block width in frequency. The smallest rectangle has the size 1x2, 1 in frequency and 2 in times. With $k = 1$ (the redundancy factor), $\overline{\epsilon}^2$ is following a χ^2 distribution with the size of the block as degree of freedom. Due to discretization effects, λ takes roughly the same values for $W_i = 1$ and $W_i = 2$. So, to compute λ for $W_i = 1$, we are doing the same as if $W_i = 2$

The following matrix gives the computed values of λ for different size of blocks (computed thanks to table I) :

$$M_\lambda = \begin{pmatrix} 1.5 & 1.8 & 2 & 2.5 & 2.5 \\ 1.8 & 2 & 2.5 & 3.5 & 3.5 \\ 2 & 2.5 & 3.5 & 4.7 & 4.7 \end{pmatrix}$$

$B_i^{\#}$	4	8	16	32	64	128
λ	4.7	3.5	2.5	2.0	1.8	1.5

TABLE I

THRESHOLDING LEVEL λ CALCULATED FOR DIFFERENT BLOCK SIZE $B^{\#}$ WITH $\delta = 0.1\%$.

SURE theorem [3]

Even if an upper bound of the risk can be found, then it can not be computed while the signal f is unknown. That is why we use an estimator of the risk which is found with the SURE theorem. This theorem is used to find the best block shapes into a macro-block by minimising this estimated risk.

This is the SURE (Stein Unbiased Risk Estimate) theorem :

Theorem 1 *Let Y be the noisy signal. It's a normal random vector with the identity as covariance matrix and of expectation F, which is the signal searched, without noise. So, $Y = F + \epsilon$ where $\epsilon \sim \mathcal{N}(0, I_p)$. F is estimated by $Y + h(Y)$, where h is differentiable a.s. $h : \mathbb{R}^p \to \mathbb{R}^p$ and*

$$\nabla.h = \Sigma_{j=1}^{p} \frac{\partial h_j}{\partial y_j}.$$

Assuming $\mathbb{E}[\Sigma_{j=1}^{p}|\partial h_j/\partial y_j|] < \infty$,

$$R = \mathbb{E}[||Y + h(Y) - F||^2] = p + \mathbb{E}[||h(Y)||^2 + 2\nabla.h(Y)]$$

$$\hat{R} = p + ||h(Y)||_2^2 + 2\nabla h(Y)$$

$\hat{R}$ is an unbiased estimator of the risk R of $Y + h(Y)$.

Some precisions :
$Y_i + h(Y_i) = a_i.Y_i$ is an estimator of F.
Therefore, the k^{th}-block risk is :

$$\begin{array}{rcl} R_k & = & \sum_{(i,j)\in B_k} \mathbb{E}[|F[i,j] - a_k F[i,j]|^2] \\ & = & \sum_{(i,j)\in B_k} \mathbb{E}[|F[i,j] - Y[i,j] - h(Y[i,j])|^2]. \end{array}$$

Finally, the blocks are chosen to minimize $\sum_k \hat{R}_k$.

For each macroblock

A macroblock is 8 points in time (horizontally) and 16 points in frequency (vertically). The beginning is time 1 and frequency -1. Each macroblock is treated independently.

15 different subdivisions are tested (cf figure 6) and the best is kept. For a block, the risk can be computed by the following formula

$$\hat{R}_i = \sigma^2 \left(B_i^{\#} + \frac{\lambda^2 B_i^{\#} - 2\lambda(B_i^{\#} - 2)}{\frac{\overline{Y_i^2}}{\sigma^2}} 1_{\overline{Y_i^2} \geqslant \lambda\sigma^2} + B_i^{\#} \left(\frac{\overline{Y_i^2}}{\sigma^2} - 2 \right) 1_{\overline{Y_i^2} < \lambda\sigma^2} \right). \qquad (2)$$

This formula gives the estimation of the risk of the block i of size $B_i^{\#}$. It is obtains using the SURE theorem with :

- $p = B_i^{\#}$;
- $h(Y_i) = (a_i - 1)Y_i$;
- a_i is given by the formula (1).

For a given subdivision, the estimation of the risk of the macro-block, is the sum of the risk estimations of each block of the subdivision. All the 15 subdivisions are tested. The one with the minimal risk estimation is chosen. The attenuation coefficients are computed in the same way as for the zero frequency (formula (1)). Then, $AttenFactorMap$ and $flag\_depth$ are filled.

4.1.3 Handling the last samples

For the last blocks, which are not full in frequency, all the coefficients of each block are treated together like for the zero frequency. For the last few coefficients that do not make up a block, do hard thresholding.

For positive frequencies, conjugate from the negative frequencies

4.2 Algorithm (cf algorithm 3)

- Inputs :
  - f : the noisy signal.
  - $time_{win}$: the length in ms of the used windows.
  - $f_{sampling}$: the sampling frequency.
  - σ_{noise} : the sigma of the noise .
- Outputs
  - f_{rec} : the denoised signal.
  - $AttenFactorMap$: the matrix with the attenuation coefficients.
  - $flag_{depth}$: the matrix which contains the optimal subdivision.

5 Experiments and Results

5.1 Miscellaneous observations

The algorithm utilisation has been optimized at two levels. The first one is the passage to the time and frequency field. Indeed, parameters can be changed to get some minor ameliorations on the SNR.

The second is that in audio mastering it is nice to work with an time invariant method. The algorithm, in the current state, isn't time invariant. We tried to come closer to this property.

5.2 Improvements on the STFT

5.2.1 Effects of the size of the used windows

Varying the size of windows changes the SNR. It can be either better or worse. So we tried to find an optimal bracket of window size.

The first thing which can be seen is that the optimal size (in time) depends on the sampling frequency. The results show that the level of noise is not really significant on the optimal size.

Figures 7, 8 and 9 presents the variations of the SNR when the size of windows changes. These tables were computed with different signals, whose sampling frequency is given.

5.2.2 Effects of the factor of redundancy

When the factor of redundancy is increased, each point is processed several times more. The coefficients matrix, computed from the STFT algorithm with $k > 1$ gives a more precise representation in time and frequency.

As the tables show, we noticed a minor amelioration of the SNR, and this for all the size of windows. Also, this minor amelioration depends on the level of noise and increasing k is useful only when the noise is rather high.

Algorithm 3: block thresholding

```
1  $STFT_{coef} \leftarrow STFT(f, time_{win}, f_{sampling})$ matrix $W \times Z$;
2  Initialize $AttenFactorMap$, $flag_{depth}$ and $STFT_{coef_{th}}$, the last one will be the oracle ;
3  $\sigma_{hanning} \leftarrow \sigma_{noise} \cdot \sqrt{0,375}$;
4  Create the $3 \times 5$ $\lambda$-matrix : $M_\lambda \leftarrow \begin{pmatrix} 1.5 & 1.8 & 2 & 2.5 & 2.5 \\ 1.8 & 2 & 2.5 & 3.5 & 3.5 \\ 2 & 2.5 & 3.5 & 4.7 & 4.7 \end{pmatrix}$;
5  Initialize $SURE$, matrix $3 \times 5$;
6  Start at time 1 and frequency -1;
7  foreach $i$ from 1 to $floor\left(\frac{Z}{8}\right)$ (loop over time) do
8      For the zero frequency, deal with the block $1 \times 8$;
9      Compute $a_i$ for this block with the equation (1);
10     Store $a_i$ in the right spot in $AttenFactorMap$;
11     Store the subdivision (block $1 \times 8$) in $flag_{depth}$;
12     foreach Macro-block in the negative frequencies do
13         foreach Possible subdivision (cf figure 6) do
14             Compute the risk with the equation (2);
15             Store the result in the right subdivision state in $SURE$;
16         end
17         Find the Minimum of $SURE$ and the matched subdivision;
18         foreach Mini-block for this optimal subdivision do
19             Compute $a_i$ for this block with the equation (1);
20             Store $a_i$ in the right spot in $AttenFactorMap$;
21             Store this subdivision (block $1 \times 8$) in $flag_{depth}$;
22         end
23     end
24     foreach Last block not full in frequency do
25         Do the same treatment as the zero frequency;
26     end
27     foreach Last block not full in frequency and time do
28         Do hard thresholding;
29     end
30 end
31 For positive frequencies, conjugate the result from negative frequencies;
32 Product term to term : $STFT_{coef_{th}} \leftarrow STFT_{coef} .* AttenFactorMap$;
33 Wiener Filter with this oracle : $STFT_{coef_{id}} \leftarrow Wiener(STFT_{coef_{th}})$;
34 Invert the result : $f_{rec} \leftarrow inverseSTFT(STFT_{coef_{id}}, time_{win}, f_{sampling}, length(f))$;
35 Return $f_{rec}, AttenFactorMap, flag_{depth}$;
```

Algorithm 4: Amelioration : invariance

```
1  Compute the STFT of the signal : $STFT_{coef} \leftarrow STFT(f)$;
2  foreach $j$ from 0 to 7 do
3      Add $j$ empty columns at the beginning of $STFT_{coef}$;
4      Denoise with the non modified algorithm of block threholdings;
5      Store the result ;
6  end
7  Return the average of the eight denoised signal;
```

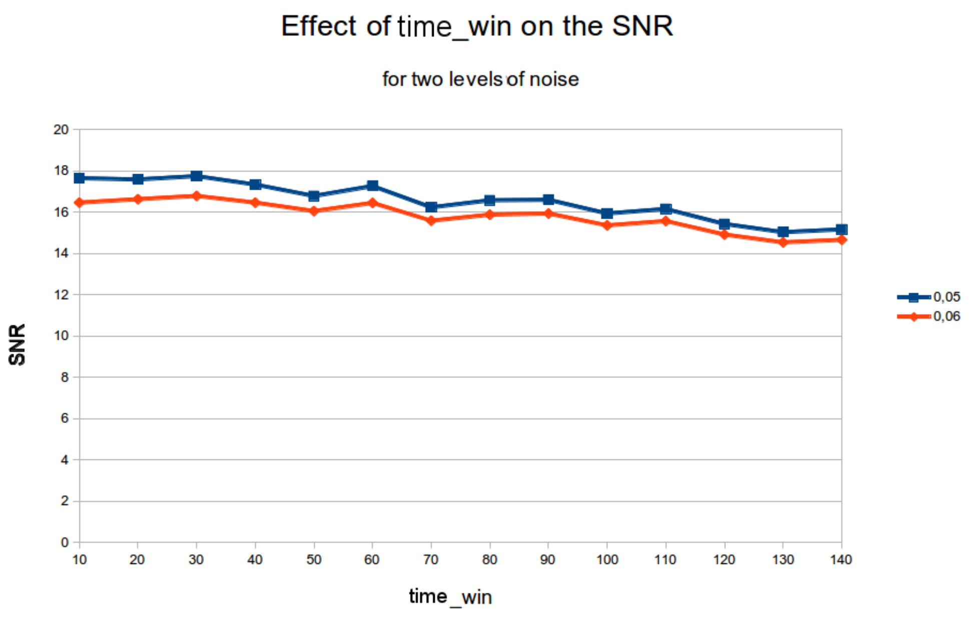


Figure 7: f_sampling = 44100 Hz

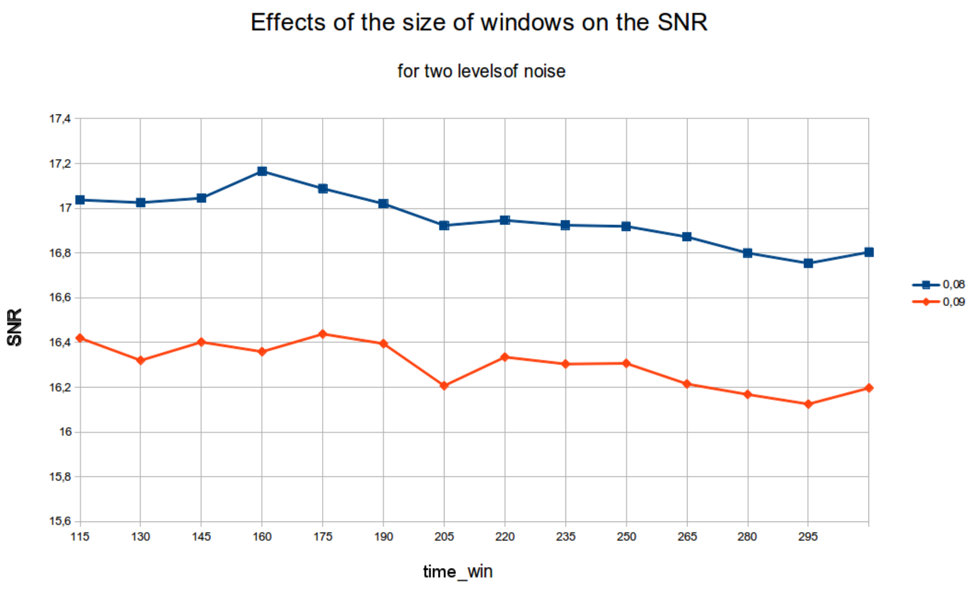


Figure 8: f_sampling = 11000 Hz

5.3 Time invariance

The main idea is to do a translation of the subdivision in macroblocks (which is arbitrary) and perform, in this way, eight different denoisings of the same signal and do the average between the eight.

This method is explicitly explained in the algorithm 4.

The figure 11 shows that the amelioration is not really good in SNR. However, it is much more pleasant to listen to. Indeed, the musical noise is hardly reduced.

This result is not really surprising. Indeed, when the signal is denoised, some artefacts appears. But they are really short in time and very localized. With this amelioration, we can rationally hope that the new artefacts will be created elsewhere, and then, each artefact will be divided by 8. That

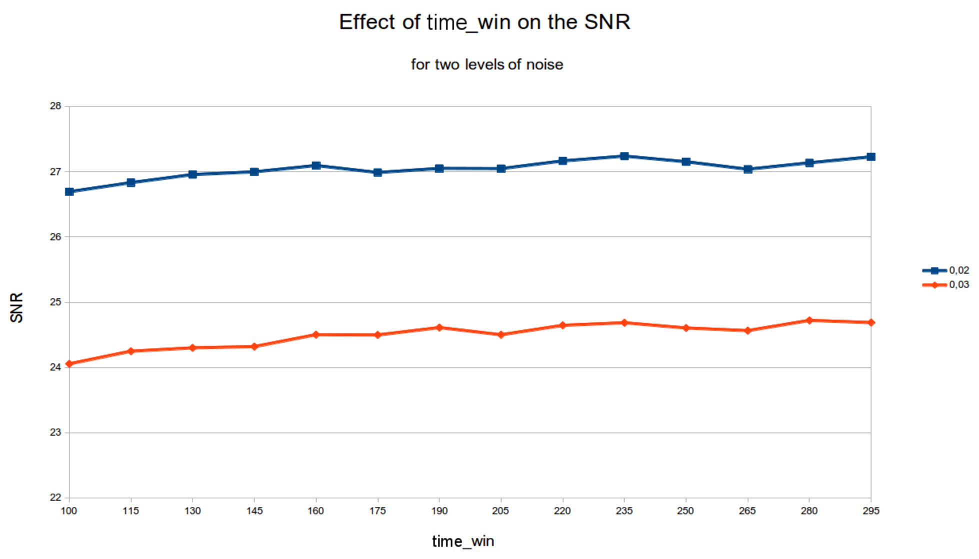
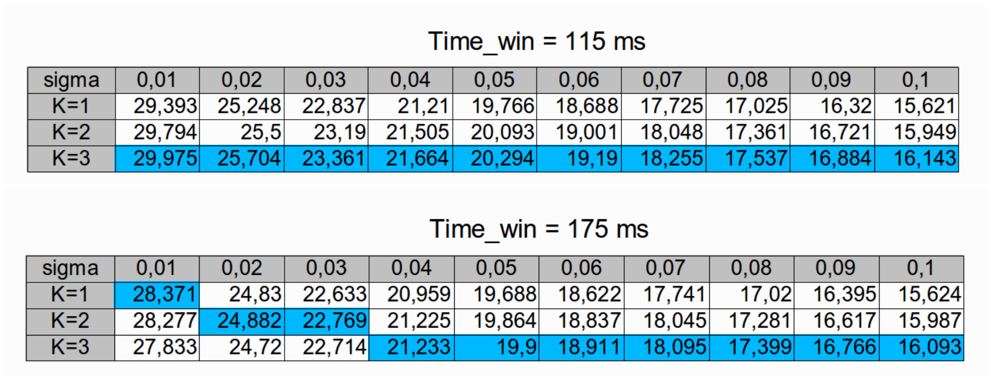


Figure 9: f_sampling = 16000 Hz



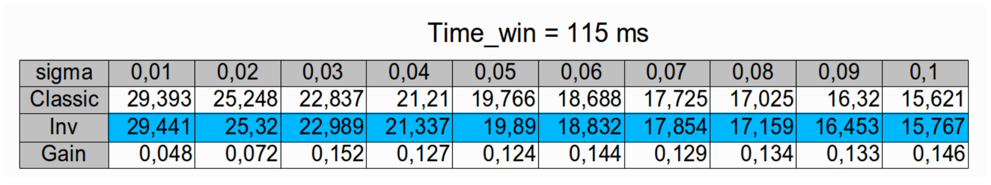
Time_win = 115 ms

sigma	0,01	0,02	0,03	0,04	0,05	0,06	0,07	0,08	0,09	0,1
K=1	29,393	25,248	22,837	21,21	19,766	18,688	17,725	17,025	16,32	15,621
K=2	29,794	25,5	23,19	21,505	20,093	19,001	18,048	17,361	16,721	15,949
K=3	29,975	25,704	23,361	21,664	20,294	19,19	18,255	17,537	16,884	16,143

Time_win = 175 ms

sigma	0,01	0,02	0,03	0,04	0,05	0,06	0,07	0,08	0,09	0,1
K=1	28,371	24,83	22,633	20,959	19,688	18,622	17,741	17,02	16,395	15,624
K=2	28,277	24,882	22,769	21,225	19,864	18,837	18,045	17,281	16,617	15,987
K=3	27,833	24,72	22,714	21,233	19,9	18,911	18,095	17,399	16,766	16,093

Figure 10: Effect of the factor of redundancy



Time_win = 115 ms

sigma	0,01	0,02	0,03	0,04	0,05	0,06	0,07	0,08	0,09	0,1
Classic	29,393	25,248	22,837	21,21	19,766	18,688	17,725	17,025	16,32	15,621
Inv	29,441	25,32	22,989	21,337	19,89	18,832	17,854	17,159	16,453	15,767
Gain	0,048	0,072	0,152	0,127	0,124	0,144	0,129	0,134	0,133	0,146

Figure 11: Effect of the invariance method

explains the previous observations.

5.4 Assessment

For each previous improvement, the improvements are not very relevant. But when we put these three methods together, the result is quite interesting. Indeed, a gain of 0,7 in the SNR can be reached compared to the examples given by Guoshen Yu [1].

6 Conclusion

This paper introduces an adaptive audio block thresholding algorithm. The denoising parameters are computed according to the time-frequency regularity of the audio signal using the SURE (Stein Unbiased Risk Estimate) theorem.

Unlike the diagonal estimators, this algorithm based on a non-diagonal estimator is really effective with white noise. However there are some defects. The sounds which are like a white Gaussian noise will be deleted. For instance, its impossible to hear cymbals from a drum kit after a denoising.

Acting on the above parameters, the SNR can be improved. Moreover, the implementation in ANSI C code improves highly the algorithm running time. Actually, there is a room for improvement, for instance, there are different possible shapes of the blocks inside the macro-blocks. In this algorithm, the macro-blocks are only split into rectangles. Another possibility would be to find an algorithm choosing the best macro-blocks for each time-frequency coefficient. This algorithm could be generalized for every noise.

References

[1] Guoshen Yu, Stéphane Mallat and Emmanuel Bacry *Audio Denoising by Time-Frequency Block Thresholding*, IEEE Transactions On Signal Processing, Vol. 56, No. 5, May 2008,
http://www.cmap.polytechnique.fr/~yu/publications/IEEEaudioblock.pdf

[2] Stéphane Mallat, *a Wavelet tour of signal processing - The Sparse Way*, 3^{rd} edition, December 2009,
Website.[2]

[3] Charles M. Stein, *Estimation of the Mean of a Multivariate normal distribution*, from *the Annals of Statistics*, 1981, Vol. 9, No.6, 1135-1151,
http://projecteuclid.org/DPubS/Repository/1.0/Disseminate?view=body&id=pdf_1&handle=euclid.aos/1176345632

[4] Gabriel Peyré, *Advanced Signal, Image and Surface Processing*, Jannuary 14, 2010,
http://www.ceremade.dauphine.fr/~peyre/numerical-tour/book/AdvancedSignalProcessing.pdf
Website.[3]

[5] Sylvie Fabre, Jean-Michel Morel, Yann Gousseau, *Analyse hilbertienne et analyse de Fourier*, October 2011,
http://dev.ipol.im/~morel/PolyAnalsye170702/PolyHilbertFourier.pdf

[6] R. J. McAulay and M. L. Malpass, *Speech enhancement using soft decisions noise suppression filter*, IEEE Trans. Acsoust. Speech, Signal process, ASSP-28, pp. 137-145, 1980.

[7] D. Donoho and I. Johnstone, *Idea spatial adaptation via wavelet shrinkage*, Biometrika, vol. 81, pp. 425-455, 1994.

[8] O. Cappé, *Elimination oft the musical noise phenimenon with the Ephraim and Malah noise suppressor*, IEEE Trans. Speech, Audio Process., vol. 2, pp. 345-349, Apr. 1994.

[2] http://www.ceremade.dauphine.fr/~peyre/wavelet-tour/
[3] http://www.ceremade.dauphine.fr/~peyre/numerical-tour/

7 Annexe

7.1 Matlab source code

Benchmark

```
%%%%%%%%%%%%%%%% Data Tables bench mark For moz1_11kHz %%%%%%%%%%%%%%%%%%

%
% Eric Martin, Marie de Masson d'Autume and Varray Christophe
%
%
% This little script compute some denoising For different methods
% denoising with different parameters. One table corresponds to one used
% method.
%
%
%
% k1 : factor of redundancy k=1
% k2 : factor of redundancy k=2
% k3 : factor of redundancy k=3
% inv : k=1 and time invariant
% inv_k2 : k=2 and time invariant
% inv_k3 : k=3 and time invariant
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%% Loading signal and initialising results tables

clear
close all

[f,f_sampling] = wavread('moz1_11kHz.wav');

res_k1=[[0,[0:0.01:0.1]] ;...
0 zeros(size([0:0.01:0.1])); ...
50 zeros(size([0:0.01:0.1])); ...
65 zeros(size([0:0.01:0.1])); ...
80 zeros(size([0:0.01:0.1])); ...
95 zeros(size([0:0.01:0.1])); ...
110 zeros(size([0:0.01:0.1])); ...
125 zeros(size([0:0.01:0.1])); ...
140 zeros(size([0:0.01:0.1])); ...
155 zeros(size([0:0.01:0.1])); ...
170 zeros(size([0:0.01:0.1])); ...
185 zeros(size([0:0.01:0.1])); ...
200 zeros(size([0:0.01:0.1])); ...
215 zeros(size([0:0.01:0.1])); ...
230 zeros(size([0:0.01:0.1])); ...
245 zeros(size([0:0.01:0.1]))];
```

```
res_k2=res_k1;
res_k3=res_k1;
res_inv=res_k1;
res_k2_inv=res_k1;
res_k3_inv=res_k1;

[I,J]=size(res_k1);

%% Remplissage res

for i=2:J
    % Noising of the signal
    fn=f+res_k1(1,i)*randn(size(f));
    % SNR of the noisy signal
    res_k1(2,i)=get_SNR(f,fn);
    for j=3:I
        % Just for knowing the progress
            [i,j]
            % 6 methods :
            [f_rec, AttenFactorMap, flag_depth] = BlockThresholding(fn, res_k1(j,1), f_sampling, res_k1(1,i));
            res_k1(j,i)=get_SNR(f,f_rec);
            [f_rec, AttenFactorMap, flag_depth] = BlockThresholding_perso(fn, res_k2(j,1), f_sampling,
res_k2(1,i));
            res_k2(j,i)=get_SNR(f,f_rec);
            [f_rec, AttenFactorMap, flag_depth] = BlockThresholding_perso2(fn, res_k3(j,1), f_sampling,
res_k3(1,i));
            res_k3(j,i)=get_SNR(f,f_rec);
            [f_rec, AttenFactorMap, flag_depth] = BlockThresholding_inv8(fn, res_inv(j,1), f_sampling,
res_inv(1,i));
            res_inv(j,i)=get_SNR(f,f_rec);
            [f_rec, AttenFactorMap, flag_depth] = BlockThresholding_inv8_k2(fn, res_k2_inv(j,1), f_sampling,
res_k2_inv(1,i));
            res_k2_inv(j,i)=get_SNR(f,f_rec);
            [f_rec, AttenFactorMap, flag_depth] = BlockThresholding_inv8_k3(fn, res_k3_inv(j,1), f_sampling,
res_k3_inv(1,i));
            res_k3_inv(j,i)=get_SNR(f,f_rec);
        end
    end


    % Data extraction
    dlmwrite('bench/moz1_11kHz/res_k1.txt',res_k1);
    dlmwrite('bench/moz1_11kHz/res_k2.txt',res_k2);
    dlmwrite('bench/moz1_11kHz/res_k3.txt',res_k3);
    dlmwrite('bench/moz1_11kHz/res_inv.txt',res_inv);
    dlmwrite('bench/moz1_11kHz/res_k2_inv.txt',res_k2_inv);
    dlmwrite('bench/moz1_11kHz/res_k3_inv.txt',res_k3_inv);
```

Unmodified Block thresholding

```
function [f_rec, AttenFactorMap, flag_depth] = BlockThresholding(fn, time_win, f_sampling, sigma_noise)

% Block attenuation with bi-dimensional (time and frequency, LxW) blocks.
% Block size selected by SURE from LxW to 2^(-Kl+1)L x 2^(-Kw+1)W
%
% The blocks in each Macroblock are of the same size. The Macroblock is of
% size Lmacro x Wmacro, with Lmacro = Cl x Lmax and Wmacro = Cw x Wmax. We
% take Cl = 1, Cw > 1, i.e., the Macroblock is in vertical sense.
%

% Maximum block length and width
Lmax = 8;
Wmax = 16;
% Block length = Lmax*2^(-kl), kl=0,...,Kl
Kl = 3;
% Block width = Wmax*2^(-kw), kw=0,...,Kw
Kw = 5;
% Macroblock is of size Lmax x Wmax
Cw = 1;
Wmacro = Wmax * Cw;

% STFT
factor_redund = 1;
STFTcoef = STFT(fn, time_win, factor_redund, f_sampling);


AttenFactorMap = zeros(size(STFTcoef));
flag_depth = zeros(size(STFTcoef));

STFTcoef_th = zeros(size(STFTcoef));

sigma_noise_hanning = sigma_noise * sqrt(0.375);

nb_Macroblk = floor(size(STFTcoef, 2) / Lmax);

half_nb_Macroblk_freq = floor(((size(STFTcoef, 1)-1)/2)/ Wmacro);

% A matrix that stores the lambda For different LxW configurations.
% (8x16), (8x8), (8x4), (8x2), (8x1)
% (4x16), (4x8), (4x4), (4x2), (4x1)
% (2x16) (2x8), (2x4), (2x2), (2x1)
M_lambda = zeros(Kl, Kw);
M_lambda = [1.5, 1.8, 2, 2.5, 2.5;
1.8, 2, 2.5, 3.5, 3.5;
2, 2.5, 3.5, 4.7, 4.7];
```

```
for i = 1 : nb_Macroblk

    % Note that the size of the Hanning window is ODD. We have both -pi, pi
    % and 0 frequency components.
    % Use L = Lmax = 8, lambda = 2.5 For components at zero frequency.
    L_pi = 8;
    lambda_pi = 2.5;
    a = 1 - lambda_pi*L_pi*(sigma_noise_hanning)^2*size(STFTcoef, 1) ./ sum(abs(STFTcoef(1, (i-
1)*Lmax+1:(i-1)*Lmax + L_pi).^2));
    a = a * (a>0);
    STFTcoef_th(1, (i-1)*Lmax+1:(i-1)*Lmax + L_pi) = a .* STFTcoef(1, (i-1)*Lmax+1:(i-1)*Lmax
+ L_pi);
    AttenFactorMap(1, (i-1)*Lmax+1:(i-1)*Lmax + L_pi) = a;
    flag_depth(1, (i-1)*Lmax+1:(i-1)*Lmax + L_pi) = 13;

    % For negative frequencies
    for j = 1 : half_nb_Macroblk_freq
        SURE_M = zeros(Kl, Kw);

        % loop over block length in time
        for klkl = 1 : Kl
            ll = Lmax * 2^(-klkl+1);
            % loop over block width in frequency
            for kwkw = 1 : Kw
                ww = Wmax * 2^(-kwkw+1);
                lambda_JS = M_lambda(klkl, kwkw);

                % loop over blocks in time
                for ii = 1 : 2^(klkl-1);
                    % loop over blocks in frequency.
                    % Note that For Macroblock purpose, Cw is taken into
                    % account.
                    for jj = 1 : 2^(kwkw-1) * Cw
                        B = STFTcoef(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii
1)*ll+1:(i-1)*Lmax+ii*ll);

                        % Normalize the coeffcients For the real/imaginary part has unity noise
                        % variance.
                        % 1/sqrt(2) For normalizing real and imaginary
                        % parts separately.
                        B_norm = B / (sqrt(size(STFTcoef, 1))*sigma_noise_hanning/sqrt(2));
                        size_blk = ll * ww;

                        S_real = sum(real(B_norm(:)).^2);
                        SURE_real = size_blk + (lambda_JS^2*size_blk^2-2*lambda_JS*size_blk*(size_blk-
2))/S_real*(S_real>lambda_JS*size_blk) + (S_real-2*size_blk)*(S_real<=lambda_JS*size_blk);

                        SURE_M(klkl, kwkw) = SURE_M(klkl, kwkw) + SURE_real;
                    end
                end
            end

            % Get the configuration that has the minimum error
            [min_error, idx_min] = min(SURE_M(:));
            [klkl, kwkw] = ind2sub([Kl, Kw], idx_min);
        end


        % Do the block segmentation and attenuation with the configuration
        % that has the minimum error
        ll = Lmax * 2^(-klkl+1);
        ww = Wmax * 2^(-kwkw+1);
        lambda_JS = M_lambda(klkl, kwkw);
        for ii = 1 : 2^(klkl-1);
            % loop over block in frequency
            % Note that For Macroblock purpose, Cw is taken into
            % account.
            for jj = 1 : 2^(kwkw-1) * Cw
                B = STFTcoef(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-
1)*ll+1:(i-1)*Lmax+ii*ll);
                a = (1 - lambda_JS*ll*ww*(sigma_noise_hanning)^2*size(STFTcoef, 1) ./ sum(abs(B(:)).^2));
                a = a * (a > 0);

                STFTcoef_th(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-
1)*ll+1:(i-1)*Lmax+ii*ll) = a .* B;
                x2(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-1)*ll+1:(i-
1)*Lmax+ii*ll) = a;
                flag_depth(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-1)*ll+1:(i-
1)*Lmax+ii*ll) = idx_min;
            end
        end
    end


    % For the last few frequencies that do not make a 2D Macroblock, do BlockJS
    % with 1D block. % Use L = 8, lambda = 2.5 For these frequencies.
    L_pi = 8;
    lambda_pi = 2.5;
    if 1+Wmacro*half_nb_Macroblk_freq+1 <= (size(STFTcoef, 1)+1)/2
        a_V = (1 - lambda_pi*L_pi*(sigma_noise_hanning)^2*size(STFTcoef, 1) ./ sum(abs(STFTcoef(1+Wma
(i-1)*Lmax+1:(i-1)*Lmax + L_pi)).^2,2));
        a_V = a_V .* (a_V > 0);

        STFTcoef_th(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-1)*Lmax+1:(i-1)*Lmax +
L_pi) = repmat(a_V, [1, L_pi]) .* STFTcoef(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-
1)*Lmax+1:(i-1)*Lmax + L_pi);
        AttenFactorMap(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-1)*Lmax+1:(i-1)*Lmax
+ L_pi) = repmat(a_V, [1, L_pi]);
        flag_depth(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-1)*Lmax+1:(i-1)*Lmax + L_pi)
= 13;
```

```
    end

end

% For positive frequencies, conjugate from the negative frequencies
STFTcoef_th(size(STFTcoef,1):-1:(end+1)/2+1, :) = conj(STFTcoef_th(2:(end+1)/2, :));
AttenFactorMap(size(STFTcoef,1):-1:(end+1)/2+1, :) = AttenFactorMap(2:(end+1)/2, :);
flag_depth(size(STFTcoef,1):-1:(end+1)/2+1, :) = flag_depth(2:(end+1)/2, :);

% For the last few coefficients that do not make up a block, do hard
% thresholding
STFTcoef_th(:, nb_Macroblk*Lmax+1 : end) = STFTcoef(:, nb_Macroblk*Lmax+1 : end) .* (abs(STFTcoef(
nb_Macroblk*Lmax+1 : end)) / sqrt(size(STFTcoef, 1)) > 3 * sigma_noise_hanning);

% figure;
% plot_spectrogram(STFTcoef_th,fn)
% title('Spectro avant Wiener');

% Inverse Windowed Fourier Transform
f_rec = inverseSTFT(STFTcoef_th, time_win, factor_redund, f_sampling, length(fn));

% Empirical Wiener
STFTcoef_ideal = STFT(f_rec, time_win, factor_redund, f_sampling);
STFTcoef_wiener = zeros(size(STFTcoef));
% % Wiener (Note that the noise' standard deviation is 1/(sqrt(2)) times
% % after multiplying with a Hanning window.)
STFTcoef_wiener = STFTcoef .* (abs(STFTcoef_ideal).^2 ./ (abs(STFTcoef_ideal).^2 + size(STFTcoef,
1)*(sigma_noise_hanning)^2));
% Inverse Windowed Fourier Transform
f_rec = inverseSTFT(STFTcoef_wiener, time_win, factor_redund, f_sampling, length(fn));

figure;
plot_spectrogram(STFTcoef_wiener,fn);
title('Spectro apres Wiener')




function STFTcoef = STFT(f, time_win, factor_redund, f_sampling)
%
% 1D Windowed Fourier Transform.
%
% Input:
% - f: Input 1D signal.
% - time_win: window size in time (in millisecond).
% - factor_redund: logarithmic redundancy factor. The actual redundancy
% factor is 2^factor_redund. When factor_redund=1, it is the minimum
% twice redundancy.
% - f_sampling: the signal sampling frequency in Hz.
%
```

```
% Output:
% - STFTcoef: Spectrogram. Column: frequency axis from -pi to pi. Row: time
% axis.
%
% Remarks:
% 1. The last few samples at the End of the signals that do not compose a complete
% window are ignored in the transform in this Version 1.
% 2. Note that the reconstruction will not be exact at the beginning and
% the End of, each of half window size. However, the reconstructed
% signal will be of the same length as the original signal.
%
% See also:
% inverseSTFT
%
% Guoshen Yu
% Version 1, Sept 15, 2006


% Check that f is 1D
if length(size(f))  = 2 — (size(f,1) =1 && size(f,2) =1)
    error('The input signal must 1D.');
end

if size(f,2) == 1
    f = f';
end

% Window size
size_win = round(time_win/1000 * f_sampling);

% Odd size For MakeHanning
if mod(size_win, 2) == 0
    size_win = size_win + 1;
end
halfsize_win = (size_win - 1) / 2;

w_hanning = MakeHanning(size_win);

Nb_win = floor(length(f) / size_win * 2);

% STFTcoef = zeros(2^(factor_redund-1), size_win, Nb_win-1);
STFTcoef = zeros(size_win, (2^(factor_redund-1) * Nb_win-1));

shift_k = round(halfsize_win / 2^(factor_redund-1));
% Loop over
for k = 1 : 2^(factor_redund-1)
    % Loop over windows
    for j = 1 : Nb_win - 2 % Ingore the last few coefficients that do not make a window
        f_win = f(shift_k*(k-1)+(j-1)*halfsize_win+1 : shift_k*(k-1)+(j-1)*halfsize_win+size_win);
        STFTcoef(:, (k-1)+2^(factor_redund-1)*j) = fft(f_win .* w_hanning');
    end
end



function f_rec = inverseSTFT(STFTcoef, time_win, factor_redund, f_sampling, length_f)
%
% Inverse windowed Fourier transform.
%
% Input:
% - STFTcoef: Spectrogram. Column: frequency axis from -pi to pi. Row: time
% axis. (Output of STFT).
% - time_win: window size in time (in millisecond).
% - factor_redund: logarithmic redundancy factor. The actual redundancy
% factor is 2^factor_redund. When factor_redund=1, it is the minimum
% twice redundancy.
% - f_sampling: the signal sampling frequency in Hz.
% - length_f: length of the signal.
%
% Output:
% - f_rec: reconstructed signal.
%
% Remarks:
% 1. The last few samples at the End of the signals that do not compose a complete
% window are ignored in the forward transform STFT of Version 1.
% 2. Note that the reconstruction will not be exact at the beginning and
% the End of, each of half window size.
%
% See also:
% STFT
%
% Guoshen Yu
% Version 1, Sept 15, 2006

% Window size
size_win = round(time_win/1000 * f_sampling);

% Odd size for MakeHanning
    if mod(size_win, 2) == 0
        size_win = size_win + 1;
    end
    halfsize_win = (size_win - 1) / 2;

    Nb_win = floor(length_f / size_win * 2);

    % Reconstruction
    f_rec = zeros(1, length_f);

    shift_k = round(halfsize_win / 2^(factor_redund-1));
    % Loop over windows
    for k = 1 : 2^(factor_redund-1)
        for j = 1 : Nb_win - 1
            f_win_rec = ifft(STFTcoef(:, (k-1)+2^(factor_redund-1)*j));
            f_rec(shift_k*(k-1)+(j-1)*halfsize_win+1 : shift_k*(k-1)+(j-1)*halfsize_win+size_win) = f_rec(shift_k
1)+(j-1)*halfsize_win+1 : shift_k*(k-1)+(j-1)*halfsize_win+size_win) + (f_win_rec');
        end
    end

    f_rec = f_rec / 2^(factor_redund-1);
```

Time invariant block thresholding

```
function [f_rec, AttenFactorMap, flag_depth] = BlockThresholding_inv8(fn, time_win, f_sampling,
sigma_noise)

% Block attenuation with bi-dimensional (time and frequency, LxW) blocks.
% Block size selected by SURE from LxW to 2^(-Kl+1)L x 2^(-Kw+1)W
%
% The blocks in each Macroblock are of the same size. The Macroblock is of
% size Lmacro x Wmacro, with Lmacro = Cl x Lmax and Wmacro = Cw x Wmax. We
% take Cl = 1, Cw > 1, i.e., the Macroblock is in vertical sense.
%
% This version, updated by Eric Martin, Marie de Masson d'Autume and
% Christophe Varray, use an time invariant method with a factor of
% redundancy of 1.
%
%

% Maximum block length and width
Lmax = 8;
Wmax = 16;
% Block length = Lmax*2^(-kl), kl=0,...,Kl
Kl = 3;
% Block width = Wmax*2^(-kw), kw=0,...,Kw
Kw = 5;
% Macroblock is of size Lmax x Wmax
Cw = 1;
Wmacro = Wmax * Cw;


STFTcoef_th2 = 0;

% cration de la boucle pour invariance par 1

for z=0:7


    % STFT
    factor_redund = 1;
    STFTcoef = STFT(fn, time_win, factor_redund, f_sampling);
    STFTcoef1 = zeros(size(STFTcoef,1),size(STFTcoef,2)+z);
    STFTcoef1(:,z+1:end) = STFTcoef;

    AttenFactorMap = zeros(size(STFTcoef1));
    flag_depth = zeros(size(STFTcoef1));

    STFTcoef_th = zeros(size(STFTcoef1));

    sigma_noise_hanning = sigma_noise * sqrt(0.375);

    nb_Macroblk = floor(size(STFTcoef1, 2) / Lmax);

    half_nb_Macroblk_freq = floor(((size(STFTcoef1, 1)-1)/2)/ Wmacro);

    % A matrix that stores the lambda For different LxW configurations.
    % (8x16), (8x8), (8x4), (8x2), (8x1)
    % (4x16), (4x8), (4x4), (4x2), (4x1)
    % (2x16) (2x8), (2x4), (2x2), (2x1)
    M_lambda = zeros(Kl, Kw);
    M_lambda = [1.5, 1.8, 2, 2.5, 2.5;
    1.8, 2, 2.5, 3.5, 3.5;
    2, 2.5, 3.5, 4.7, 4.7];

    for i = 1 : nb_Macroblk

        % Note that the size of the Hanning window is ODD. We have both -pi, pi
        % and 0 frequency components.
        % Use L = Lmax = 8, lambda = 2.5 For components at zero frequency.
        L_pi = 8;
        lambda_pi = 2.5;
        a = 1 - lambda_pi*L_pi*(sigma_noise_hanning)^2*size(STFTcoef1, 1) ./ sum(abs(STFTcoef1(1,
(i-1)*Lmax+1:(i-1)*Lmax + L_pi).^2));
        a = a * (a>0);
        STFTcoef_th(1, (i-1)*Lmax+1:(i-1)*Lmax + L_pi) = a .* STFTcoef1(1, (i-1)*Lmax+1:(i-
1)*Lmax + L_pi);
        AttenFactorMap(1, (i-1)*Lmax+1:(i-1)*Lmax + L_pi) = a;
        flag_depth(1, (i-1)*Lmax+1:(i-1)*Lmax + L_pi) = 13;

        % For negative frequencies
        for j = 1 : half_nb_Macroblk_freq
            SURE_M = zeros(Kl, Kw);

            % loop over block length in time
            for klkl = 1 : Kl
                ll = Lmax * 2^(-klkl+1);
                % loop over block width in frequency
                for kwkw = 1 : Kw
                    ww = Wmax * 2^(-kwkw+1);
                    lambda_JS = M_lambda(klkl, kwkw);

                    % loop over blocks in time
                    for ii = 1 : 2^(klkl-1);
                        % loop over blocks in frequency.
                        % Note that For Macroblock purpose, Cw is taken into
                        % account.
                        for jj = 1 : 2^(kwkw-1) * Cw
                            B = STFTcoef1(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-
1)*Lmax+(ii-1)*ll+1:(i-1)*Lmax+ii*ll);

                            % Normalize the coeffcients For the real/imaginary part has unity noise
                            % variance.
                            % 1/sqrt(2) For normalizing real and imaginary
                            % parts separately.
                            B_norm = B / (sqrt(size(STFTcoef1, 1))*sigma_noise_hanning/sqrt(2));
                            size_blk = ll * ww;

                            S_real = sum(real(B_norm(:)).^2);
                            SURE_real = size_blk + (lambda_JS^2*size_blk^2-2*lambda_JS*size_blk*(size_blk-
2))/S_real*(S_real>lambda_JS*size_blk) + (S_real-2*size_blk)*(S_real<=lambda_JS*size_blk);

                            SURE_M(klkl, kwkw) = SURE_M(klkl, kwkw) + SURE_real;
                        end
                    end
                end

                % Get the configuration that has the minimum error
                [min_error, idx_min] = min(SURE_M(:));
                [klkl, kwkw] = ind2sub([Kl, Kw], idx_min);
            end


            % Do the block segmentation and attenuation with the configuration
            % that has the minimum error
            ll = Lmax * 2^(-klkl+1);
            ww = Wmax * 2^(-kwkw+1);
            lambda_JS = M_lambda(klkl, kwkw);
            for ii = 1 : 2^(klkl-1);
                % loop over block in frequency
                % Note that For Macroblock purpose, Cw is taken into
                % account.
                for jj = 1 : 2^(kwkw-1) * Cw
                    B = STFTcoef1(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-
1)*ll+1:(i-1)*Lmax+ii*ll);
                    a = (1 - lambda_JS*ll*ww*(sigma_noise_hanning)^2*size(STFTcoef1, 1) ./ sum(abs(B(:)).^2
                    a = a * (a > 0);

                    STFTcoef_th(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-
1)*ll+1:(i-1)*Lmax+ii*ll) = a .* B;
```

```
                    x2(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-1)*ll+1:(i-
1)*Lmax+ii*ll) = a;
                    flag_depth(1+(j-1)*Wmacro+(jj-1)*ww+1:1+(j-1)*Wmacro+jj*ww, (i-1)*Lmax+(ii-
1)*ll+1:(i-1)*Lmax+ii*ll) = idx_min;
                end
            end
        end


        % For the last few frequencies that do not make a 2D Macroblock, do BlockJS
        % with 1D block. % Use L = 8, lambda = 2.5 For these frequencies.
        L_pi = 8;
        lambda_pi = 2.5;
        if 1+Wmacro*half_nb_Macroblk_freq+1 <= (size(STFTcoef1, 1)+1)/2
            a_V = (1 - lambda_pi*L_pi*(sigma_noise_hanning)^2*size(STFTcoef1, 1) ./ sum(abs(STFTcoef1(1+
(i-1)*Lmax+1:(i-1)*Lmax + L_pi)).^2,2));
            a_V = a_V .* (a_V > 0);

            STFTcoef_th(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-1)*Lmax+1:(i-1)*Lmax
+ L_pi) = repmat(a_V, [1, L_pi]) .* STFTcoef1(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-
1)*Lmax+1:(i-1)*Lmax + L_pi);
            AttenFactorMap(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-1)*Lmax+1:(i-1)*Lmax
+ L_pi) = repmat(a_V, [1, L_pi]);
            flag_depth(1+Wmacro*half_nb_Macroblk_freq+1:(end+1)/2, (i-1)*Lmax+1:(i-1)*Lmax +
L_pi) = 13;
        end

    end

    % For positive frequencies, conjugate from the negative frequencies
    STFTcoef_th(size(STFTcoef1,1):-1:(end+1)/2+1, :) = conj(STFTcoef_th(2:(end+1)/2, :));
    AttenFactorMap(size(STFTcoef1,1):-1:(end+1)/2+1, :) = AttenFactorMap(2:(end+1)/2, :);
    flag_depth(size(STFTcoef1,1):-1:(end+1)/2+1, :) = flag_depth(2:(end+1)/2, :);

    % For the last few coefficients that do not make up a block, do hard
    % thresholding
    STFTcoef_th(:, nb_Macroblk*Lmax+1 : end) = STFTcoef1(:, nb_Macroblk*Lmax+1 : end) .*
(abs(STFTcoef1(:, nb_Macroblk*Lmax+1 : end)) / sqrt(size(STFTcoef1, 1)) > 3 * sigma_noise_hanning);


    % sortie de la boucle

    STFTcoef_th2 = STFTcoef_th2 + STFTcoef_th(:,z+1:end);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

STFTcoef_th2=STFTcoef_th2./8;
```

```
% Inverse Windowed Fourier Transform
f_rec = inverseSTFT(STFTcoef_th2, time_win, factor_redund, f_sampling, length(fn));

% Empirical Wiener
STFTcoef_ideal = STFT(f_rec, time_win, factor_redund, f_sampling);
STFTcoef_wiener = zeros(size(STFTcoef));
% % Wiener (Note that the noise' standard deviation is 1/(sqrt(2)) times
% % after multiplying with a Hanning window.)
STFTcoef_wiener = STFTcoef .* (abs(STFTcoef_ideal).^2 ./ (abs(STFTcoef_ideal).^2 + size(STFTcoef,
1)*(sigma_noise_hanning)^2));
% Inverse Windowed Fourier Transform
f_rec = inverseSTFT(STFTcoef_wiener, time_win, factor_redund, f_sampling, length(fn));

figure;
plot_spectrogram(STFTcoef_wiener,f_rec)




function STFTcoef = STFT(f, time_win, factor_redund, f_sampling)
%
% 1D Windowed Fourier Transform.
%
% Input:
% - f: Input 1D signal.
% - time_win: window size in time (in millisecond).
% - factor_redund: logarithmic redundancy factor. The actual redundancy
% factor is 2^factor_redund. When factor_redund=1, it is the minimum
% twice redundancy.
% - f_sampling: the signal sampling frequency in Hz.
%
% Output:
% - STFTcoef: Spectrogram. Column: frequency axis from -pi to pi. Row: time
% axis.
%
% Remarks:
% 1. The last few samples at the End of the signals that do not compose a complete
% window are ignored in the transform in this Version 1.
% 2. Note that the reconstruction will not be exact at the beginning and
% the End of, each of half window size. However, the reconstructed
% signal will be of the same length as the original signal.
%
% See also:
% inverseSTFT
%
% Guoshen Yu
% Version 1, Sept 15, 2006


% Check that f is 1D
```

```
if length(size(f))  = 2 — (size(f,1) =1 && size(f,2) =1)
    error('The input signal must 1D.');
end

if size(f,2) == 1
    f = f';
end

% Window size
size_win = round(time_win/1000 * f_sampling);

% Odd size For MakeHanning
if mod(size_win, 2) == 0
    size_win = size_win + 1;
end
halfsize_win = (size_win - 1) / 2;

w_hanning = MakeHanning(size_win);

Nb_win = floor(length(f) / size_win * 2);

% STFTcoef = zeros(2^(factor_redund-1), size_win, Nb_win-1);
STFTcoef = zeros(size_win, (2^(factor_redund-1) * Nb_win-1));

shift_k = round(halfsize_win / 2^(factor_redund-1));
% Loop over
for k = 1 : 2^(factor_redund-1)
    % Loop over windows
    for j = 1 : Nb_win - 2 % Ingore the last few coefficients that do not make a window
        f_win = f(shift_k*(k-1)+(j-1)*halfsize_win+1 : shift_k*(k-1)+(j-1)*halfsize_win+size_win);
        STFTcoef(:, (k-1)+2^(factor_redund-1)*j) = fft(f_win .* w_hanning');
    end
end


function f_rec = inverseSTFT(STFTcoef, time_win, factor_redund, f_sampling, length_f)
%
% Inverse windowed Fourier transform.
%
% Input:
% - STFTcoef: Spectrogram. Column: frequency axis from -pi to pi. Row: time
% axis. (Output of STFT).
% - time_win: window size in time (in millisecond).
% - factor_redund: logarithmic redundancy factor. The actual redundancy
% factor is 2^factor_redund. When factor_redund=1, it is the minimum
% twice redundancy.
% - f_sampling: the signal sampling frequency in Hz.
% - length_f: length of the signal.
%
% Output:
```

```
% - f_rec: reconstructed signal.
%
% Remarks:
% 1. The last few samples at the End of the signals that do not compose a complete
% window are ignored in the forward transform STFT of Version 1.
% 2. Note that the reconstruction will not be exact at the beginning and
% the End of, each of half window size.
%
% See also:
% STFT
%
% Guoshen Yu
% Version 1, Sept 15, 2006

% Window size
size_win = round(time_win/1000 * f_sampling);

% Odd size For MakeHanning
if mod(size_win, 2) == 0
    size_win = size_win + 1;
end
halfsize_win = (size_win - 1) / 2;

Nb_win = floor(length_f / size_win * 2);

% Reconstruction
f_rec = zeros(1, length_f);

shift_k = round(halfsize_win / 2^(factor_redund-1));

% Loop over windows
for k = 1 : 2^(factor_redund-1)
    for j = 1 : Nb_win - 1
        f_win_rec = ifft(STFTcoef(:, (k-1)+2^(factor_redund-1)*j));
        f_rec(shift_k*(k-1)+(j-1)*halfsize_win+1 : shift_k*(k-1)+(j-1)*halfsize_win+size_win) = f_rec(shift_k*(k
1)+(j-1)*halfsize_win+1 : shift_k*(k-1)+(j-1)*halfsize_win+size_win) + (f_win_rec');
    end
end

f_rec = f_rec / 2^(factor_redund-1);
```

7.2 ANSI C code

Unmodified block Thresholding

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sndfile.h>
```

```
#include <fftw3.h>

#include "sound.h"
#include "stftGuoshen.h"
#include "cmplx.h"
#include "matrix.h"
#include "block.h"




void blockThresholding(const double * signal, long length , double time_win, lo
{

    // Block attenuation with bi-dimensional (time and frequency, LxW) blocks.
    // Block size selected by SURE from LxW to 2^(-kL+1)L x 2^(-kW+1)W

    // The blocks in each Macroblock are of the same size. The Macroblock is of
    // size lMax x wMax.


    output->f_rec = (double *) malloc(sizeof(double) * length); // f_rec is the

    // definitions
    int lMax = 8; // maximum block length
    int wMax = 16; // maximum block width
    int kL = 3; // block length = lMax*2^(-kl), kl=0,...,kL
    int kW = 5; // block width = wMax*2^(-kw), kw=0,...,kW

    int factorRedund  = 1;

    cmplxMatrix transpStftCoef;

    // computation of the signal STFT
    transpStftCoef = stft_Guoshen(signal, length, time_win, f_sampling);


    double sigmaNoiseHanning;
    sigmaNoiseHanning = sigmaNoise * sqrt(0.375); // the noise variance changes


    // A matrix that stores the lambda for different LxW  configurations. Atten
    // (8x16), (8x8), (8x4), (8x2), (8x1)
    // (4x16), (4x8), (4x4), (4x2), (4x1)
    // (2x16)  (2x8), (2x4), (2x2), (2x1)
    const double matrixLambda[3][5]={
    {    1.5,    1.8,    2.0,    2.5,    2.5},
    {    1.8,    2.0,    2.5,    3.5,    3.5},
    {    2.0,    2.5,    3.5,    4.7,    4.7}
    };
```

```
double lambda_JS;

// STFT size
long N, M;
M = transpStftCoef.n;
N = transpStftCoef.m;


// definition of the different matrices used in the algorithm
cmplxMatrix stftCoef;
newMatrix(stftCoef, N, M, cmplx);
transposeCmplxMat(transpStftCoef, stftCoef);

cmplxMatrix stftCoefTh;
cmplxMatrix transpStftCoefTh;

doubleMatrix absStftCoef;
doubleMatrix squareStftCoef;

doubleMatrix SURE_M;

cmplxMatrix B;
cmplxMatrix BB;
cmplxMatrix B_norm;
doubleMatrix realB_norm;
doubleMatrix squareRealB_norm;
doubleMatrix absBB;
doubleMatrix squareBB;

cmplxMatrix stftCoefIdeal;
cmplxMatrix stftCoefWiener;
doubleMatrix absStftCoefIdeal;
doubleMatrix squareStftCoefIdeal;
doubleMatrix absStftCoefWiener;
doubleMatrix absStftCoefTh;

doubleMatrix A;
doubleMatrix AA;

// creation and initialization of these matrices
newMatrix(SURE_M, kL, kW, double);
initializeMatrix(SURE_M);

newMatrix(stftCoefTh, N, M, cmplx);
initializeCmplxMatrix(stftCoefTh);
newMatrix(transpStftCoefTh, M, N, cmplx);
initializeCmplxMatrix(transpStftCoefTh);
newMatrix(output->attenFactorMap, N, M, double);
initializeMatrix(output->attenFactorMap);
```

```
newMatrix(output->flagDepth, N, M, long);
initializeMatrix(output->flagDepth);

newMatrix(absStftCoef, N, M, double);
initializeMatrix(absStftCoef);
newMatrix(squareStftCoef, N, M, double);
initializeMatrix(squareStftCoef);

absMat (stftCoef, absStftCoef);
squareMat ( absStftCoef , squareStftCoef);
newMatrix(absStftCoefWiener, M, N, double);
initializeMatrix(absStftCoefWiener);
newMatrix(absStftCoefTh, N, M, double);
initializeMatrix(absStftCoefTh);



// computation of the number of macroblocks
long nbMacroblk, halfNbMacroblkFreq, size_blk; //
nbMacroblk = M/lMax;
halfNbMacroblkFreq = (N-1)/(2*wMax);


long i,j,t,s,tt,klkl,ll,kwkw,ww,ii,jj,b,c,klklkl,kwkwkw; // indices
double sum, SURE_real, a, lambdaPi;
Coordonnees min;
long lPi;

double aa,bb;




// loop over the number of macroblock in time, study of a N*lMax Matrix
for(i=1; i<= nbMacroblk; i++)
{
    /*Note that the size of the Hanning window is ODD. We have both -pi, pi
     and 0 frequency components.
     Use L = Lmax = 8, lambda = 2.5 for components at zero frequency. */



    lPi = 8; //block size
    lambdaPi = 2.5; // lambda for block of size 1x8


    sum = sumMatrix( 1, 1, (i-1)*lMax+1, (i-1)*lMax+lPi, squareStftCoef);

    a = 1 - (lambdaPi*lPi*sigmaNoiseHanning*sigmaNoiseHanning)*N/sum; // at
    if (a < 0)
```

```
{
    a = 0;
}

for (j=1 ; j<=lPi ; j++)
{
    t = (i-1)*lMax+j;
    stftCoefTh.c[0][t-1].re = a*stftCoef.c[0][t-1].re;  // thresholding
    stftCoefTh.c[0][t-1].im = a*stftCoef.c[0][t-1].im;
    output->attenFactorMap.c[0][t-1] = a;               // the coeffici
    output->flagDepth.c[0][t-1] = 13;                   // the block si
}


// for negative frequencies
// loop over each macroblock
for (j = 1; j<=halfNbMacroblkFreq; j++)
{
    // initialisation of SURE_M
    initializeMatrix(SURE_M);

    // loop over block length in time
    for (klkl = 1 ; klkl<=kL ; klkl++)
    {
        ll = lMax * pow(2,1-klkl);


        //loop over block width in frequency
        for (kwkw = 1; kwkw<=kW; kwkw++)
        {
            ww = wMax * pow(2, 1-kwkw);
            lambda_JS = matrixLambda[klkl-1][kwkw-1];

            //loop over miniblock in time
            for (ii = 1; ii<=pow(2,klkl-1); ii++)
            {
                //loop over miniblock in frequency.
                for (jj = 1; jj<=pow(2,kwkw-1); jj++)
                {
                    newMatrix(B, ww, ll, cmplx);
                    initializeCmplxMatrix(B);
                    newMatrix(B_norm, ww, ll, cmplx);
                    initializeCmplxMatrix(B_norm);
                    newMatrix(realB_norm, ww, ll, double);
                    initializeMatrix(realB_norm);
                    newMatrix(squareRealB_norm, ww, ll, double);
                    initializeMatrix(squareRealB_norm);
```

```
                              t = 1+(j-1)*wMax+jj*ww -ww;
                              s = (i-1)*lMax+ii*ll-ll;



                              copyCmplxMat (t+1, t+ww, s+1, s+ll, stftCoef, B);
// copyCmplxMat copies in B (wwxll) the coefficients of stftCoef( t+1 : t+ww ,



                              /* Normalize the coeffcients for the real/imaginary
                                 1/sqrt(2) for normalizing real and imaginary par

                               for (b=0; b<ww; b++)
                               {
                                   for (c=0; c<ll; c++)
                                   {
                                       B_norm.c[b][c].re = B.c[b][c].re * sqrt(2)
                                       B_norm.c[b][c].im = B.c[b][c].im * sqrt(2)
                                   }
                               }



                               realMat(B_norm, realB_norm);
                               squareMat(realB_norm, squareRealB_norm);

                               size_blk = ll*ww;
                               sum = sumMatrix(1, ww, 1, ll, squareRealB_norm);



                               if (sum > lambda_JS*size_blk)
                               {
                                   SURE_real = size_blk + (lambda_JS*lambda_JS *
(size_blk - 2)) / sum;


                               }

                               else
                               {
                                   SURE_real = sum-size_blk;
                               }
                               SURE_M.c[klkl-1][kwkw-1] += SURE_real;

                          }
                      }

                 }
```

```
                    // Get the configuration that has the minimum error
                    min = minDoubleMat(SURE_M);
                    klklkl = min.x+1;
                    kwkwkw = min.y+1;

                }


                min = minDoubleMat(SURE_M);


                // Do the block segmentation and attenuation with the configuration
that has the minimum error

                ll = lMax * pow(2,1-klklkl);
                ww = wMax * pow(2,1-kwkwkw);
                lambda_JS = matrixLambda[min.x][min.y];


                for (ii = 1; ii<=pow(2,klklkl-1); ii++)
                {
                    // loop over the block in frequency
                    for (jj = 1; jj<=pow(2,kwkwkw-1); jj++)
                    {
                        newMatrix(BB, ww, ll, cmplx);
                        initializeCmplxMatrix(BB);
                        t = 1+(j-1)*wMax+jj*ww -ww;
                        s = (i-1)*lMax+ii*ll-ll;
                        copyCmplxMat (t+1, t+ww, s+1, s+ll, stftCoef, BB);

                        newMatrix(absBB, ww, ll, double);
                        initializeMatrix(absBB);
                        newMatrix(squareBB, ww, ll, double);
                        initializeMatrix(squareBB);

                        absMat(BB, absBB);
                        squareMat(absBB, squareBB);

                        sum = sumMatrix(1, ww, 1, ll, squareBB);
                        a = (1 - lambda_JS * ll * ww * sigmaNoiseHanning * sigmaNoi
// attenuation coefficient for this block
                        if (a<0)
                        {
                            a=0;
                        }

                        for (b=1 ; b<=ww ; b++)
                        {
                            for (c=1; c<=ll; c++)
                            {
                              t = 1 + (j-1) * wMax + jj * ww - ww + b;
                              s = (i-1)*lMax+ii*ll-ll+c;
                              aa = stftCoef.c[t-1][s-1].re;
                              bb = stftCoefTh.c[t-1][s-1].re;
                              stftCoefTh.c[t-1][s-1].re = a*stftCoef.c[t-1][s-1].
// thresholding
                              stftCoefTh.c[t-1][s-1].im = a*stftCoef.c[t-1][s-1].
                              output->attenFactorMap.c[t-1][s-1] = a;
// the attenuation coefficient is stored
                              output->flagDepth.c[t-1][s-1] = kL*(kwkwkw-1)+klklk
// the block size is stored
                          }
                      }

                  }
              }
          }
              //For the last few frequencies that do not make a 2D Macroblock, do
              // Use L = 8, lambda = 2.5 for these frequencies.
          lPi = 8;
          lambdaPi = 2.5;


          if (1+wMax*halfNbMacroblkFreq+1 <= (N+1)/2)
          {
              tt = 1+wMax*halfNbMacroblkFreq+1;

              double * a_V = (double *) malloc(sizeof(double) * (N+1)/2-tt+1);
              for (b = 0; b<(N+1)/2-tt+1; b++ )
              {

                  sum = sumMatrix(tt+b, tt+b, (i-1)*lMax+1, (i-1)*lMax+lPi, squa
                  a_V[b] = (1 - lambdaPi * lPi * sigmaNoiseHanning*sigmaNoiseHan
                  if (a_V[b] < 0)
                  {
                      a_V[b] = 0;
                  }
              }
              for (b=0; b<(N+1)/2-tt+1; b++)
              {
                  for (c=0; c<lPi; c++)
                  {
                      t = tt +b;
                      s = (i-1)*lMax+1+c;
                      stftCoefTh.c[t-1][s-1].re = a_V[b] * stftCoef.c[t-1][s-1].
                      stftCoefTh.c[t-1][s-1].im = a_V[b] * stftCoef.c[t-1][s-1].
                      output->attenFactorMap.c[t-1][s-1] = a_V[b];
                      output->flagDepth.c[t-1][s-1] = 13;
                  }
              }
```

```
        }

    }

    // For positive frequencies, conjugate from the negative frequencies

    for (b = 1; b < (N+1)/2; b++)
    {
        for (c = 0; c < M; c++)
        {
            stftCoefTh.c[N-b][c] = c_conj(stftCoefTh.c[b][c]);
            output->attenFactorMap.c[N-b][c]=output->attenFactorMap.c[b][c];
            output->flagDepth.c[N-b][c]=output->flagDepth.c[b][c];
        }
    }

    // For the last few coefficients that do not make up a block, do hard thres
    for (b=0;b<N;b++)
    {
        for(c=nbMacroblk*lMax; c<M; c++)
        {
            if  ( (absStftCoef.c[b][c]/sqrt(N) ) > 3*sigmaNoiseHanning)
            {
                stftCoefTh.c[b][c].re *= absStftCoef.c[b][c]/sqrt(N);
                stftCoefTh.c[b][c].im *= absStftCoef.c[b][c]/sqrt(N);
            }
            else
            {
                stftCoefTh.c[b][c].re = 0;
                stftCoefTh.c[b][c].im = 0;

            }



        }
    }
    absMat(stftCoefTh, absStftCoefTh);

    // Inverse Windowed Fourier Transform
    transposeCmplxMat(stftCoefTh, transpStftCoefTh);

    output->f_rec = stft_inverse_Guoshen(transpStftCoefTh, length, time_win, f_

    // Empirical Wiener

    stftCoefIdeal = stft_Guoshen(output->f_rec, length, time_win, f_sampling);
    newMatrix(absStftCoefIdeal, stftCoefIdeal.n, stftCoefIdeal.m, double);
    initializeMatrix(absStftCoefIdeal);
    newMatrix(squareStftCoefIdeal, stftCoefIdeal.n, stftCoefIdeal.m, double);
    initializeMatrix(squareStftCoefIdeal);


    // Wiener (Note that the noise' standard deviation is 1/(sqrt(2)) times aft

    absMat(stftCoefIdeal, absStftCoefIdeal);
    squareMat(absStftCoefIdeal, squareStftCoefIdeal);
    A = sumMatConst(squareStftCoefIdeal, N*sigmaNoiseHanning*sigmaNoiseHanning)
    AA = divideDoubleMat(squareStftCoefIdeal, A);
    stftCoefWiener = productCmplxDoubleMat(transpStftCoef, AA);



    absMat(stftCoefWiener, absStftCoefWiener);


    // Inverse Windowed Fourier Transform
    output->f_rec = stft_inverse_Guoshen(stftCoefWiener, length, time_win, f_sa




    return 0;


}
```

Header

```
#ifndef __BLOCK_H__
#define __BLOCK_H__

#include "cmplx.h"
#include "matrix.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "coordonnees.h"
#include "block.h"




typedef struct OutputBlockThresholding {
  doubleMatrix attenFactorMap;
  longMatrix flagDepth;
  double * f_rec;
} OutputBlockThresholding;

void blockThresholding(const double * signal, long length , double timeWin, lon
#endif


#ifndef __CMPLX__H__
#define __CMPLX__H__


#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct cmplx {
  double im;
  double re;
} cmplx;

double c_abs(cmplx c);

cmplx c_conj(cmplx c);

#endif

#ifndef __COORDONNEES_H__
#define __COORDONNEES_H__

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "coordonnees.h"


typedef struct Coordonnees Coordonnees;
struct Coordonnees
{
    int x;
    int y;
};




#endif

#ifndef __MATRIX_H__
#define __MATRIX_H__

#include "cmplx.h"
#include "matrix.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include "coordonnees.h"

double c_abs(cmplx c);



typedef struct sIntM
{
  long n, m;
  int **c;

} intMatrix;

typedef struct sLongM
{
  long n, m;
  long **c;

} longMatrix;

typedef struct sDoubleM
{
  long n, m;
  double **c;

} doubleMatrix;

typedef struct sComplexM
{
  long n, m;
  cmplx **c;

} cmplxMatrix;

Coordonnees minDoubleMat (doubleMatrix M);

void transposeCmplxMat (cmplxMatrix Matrix, cmplxMatrix NewMatrix  );

void copyCmplxMatIn ( long na, long nb,long ma, long mb, cmplxMatrix Matrix, cm
);

doubleMatrix sumMatConst ( doubleMatrix Matrix, const double x  );

void printDoubleVector(const char *message, double *v, long n);

void printComplexMatrix(const char *outfile, cmplxMatrix M);

void squareMat ( doubleMatrix Matrix, doubleMatrix squareMatrix  );

void absMat (cmplxMatrix ComplexMatrix, doubleMatrix absMatrix);
void conjMat (cmplxMatrix ComplexMatrix, cmplxMatrix conjMatrix);

void realMat ( cmplxMatrix Matrix, doubleMatrix realMatrix  );

void copyCmplxMat ( long na, long nb,long ma, long mb, cmplxMatrix Matrix, cmpl

double sumMatrix( long na, long nb, long ma, long mb, doubleMatrix Matrix);

cmplxMatrix productCmplxDoubleMat(cmplxMatrix A, doubleMatrix B);

doubleMatrix divideDoubleMat(doubleMatrix A, doubleMatrix B);

 #define newMatrix(M, nM, mM, T)                           \
  { \
    long k;                                              \
   M.c = (T **) malloc(sizeof(T *) * (nM));          \
   M.c[0] = (T *) malloc(sizeof(T) * (nM) * (mM)); \
    for (k=1; k<nM; k++) M.c[k] = M.c[0] + k * (mM);   \
   M.n = nM; \
   M.m = mM; \
  }

#define initializeMatrix(M) \
{                          \
    long i, j;              \
    for (i=0; i<M.n; i++)    \
    {                      \
        for (j=0; j<M.m; j++)\
        {                  \
            M.c[i][j]=0;     \
        }                  \
    }                      \
}

#define initializeCmplxMatrix(M)          \
{                                         \
    long i, j;                             \
    for (i=0; i<M.n; i++)                 \
    {                                     \
        for (j=0; j<M.m; j++)             \
        {                                 \
            M.c[i][j].re=0;               \
            M.c[i][j].im=0;               \
        }                                 \
    }                                     \
}


#define deleteMatrix(M) \
  { \
    free(M.c[0]);                                   \
    free(M.c);                                      \
  }

#define printMatrix(message, M, format) \
  {                                     \
    long i,j;                           \
    fprintf(fDebug, "%s\n", message);   \
    for (i=0; i<M.m; i++) {             \
      for (j=0; j<M.n; j++)             \
        fprintf(fDebug, format, M.c[j][i]);       \
      fprintf(fDebug, "\n");                      \
    }                                             \
    fprintf(fDebug, "\n");                        \
  }

#define printCMatrix(message, M, format) \
  {                                     \
    long i,j;                           \
    fprintf(fDebug, "%s\n", message);   \
    for (i=0; i<M.m; i++) {             \
      for (j=0; j<M.n; j++)             \
        fprintf(fDebug, format, M.c[j][i].re, M.c[j][i].im); \
      fprintf(fDebug, "\n");                      \
    }                                             \
    fprintf(fDebug, "\n");                        \
  }


#endif

#ifndef _SOUND_H_
#define _SOUND_H_

#include <sndfile.h>
#include <stdlib.h>

#define alloc(T, n) (T *) malloc((n)*sizeof(T))

typedef struct _Sound {

  SF_INFO _sfinfo;

  double ** channel;
  long nChannels;
  long nSamples;
  long start;
  int sampleRate;
  int Nbit;
  char errMsg[1000];  /* memore statique (pas dynamique) */

} Sound;

void soundClean(Sound *S);

Sound * audioRead(const char * infile, long n1, long n2);
void audioWrite(Sound *S, const char * outfile);

void audioWriteText(const Sound *S, const char *fileName);
Sound * audioReadText(const char *fileName);


#endif

#ifndef __SFFT__H__
#define __SFFT__H__

#include <stdlib.h>
#include "cmplx.h"
#include "matrix.h"
#include "vector.h"

double * hanning   (long nW);

cmplxMatrix stft_forward(const double * data, long nData,
                        const double * window, long nWindow);

double * stft_backward(const cmplxMatrix y, long n,
                      const double *window, long nWindow);

cmplxMatrix stft_Guoshen(const double *signal,long length, double time_win, lon

cmplxMatrix stft_forward(const double *x, long n,
                        const double *wHanning, long L);

double get_SNR(double * fo, double * f, long length_f);

#endif

#ifndef __VECTOR_H__
#define __VECTOR_H__

#include "cmplx.h"
#include "vector.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "coordonnees.h"
```

```
double sumVect (double *f , long length_f);

void squareVect (double *f, double * f_square , long length_f);

void diffVect (double *fo, double * f, double * diff , long length_fo, long len


#endif
```

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sndfile.h>
#include <fftw3.h>

#include "sound.h"
#include "stftGuoshen.h"
#include "cmplx.h"
#include "matrix.h"
#include "block.h"

int main()
{
    Sound *Snoisy, *S, *Srec;
    cmplxMatrix stftCoef;
    double snr;


    S = audioRead("deb40.wav", 1, 10000000000000);
    Snoisy = audioRead("deb40noisy.wav", 1, 100000000000);
    Srec = Snoisy;

    snr = get_SNR(S->channel[0], Snoisy->channel[0], S->nSamples);
    printf("snr = %f\n", snr);

    OutputBlockThresholding output;
    blockThresholding(Snoisy->channel[0], Snoisy->nSamples, 50.0, Snoisy->sampl

    snr = get_SNR(S->channel[0], output.f_rec, S->nSamples);
    printf("SNR = %f\n", snr);

    Srec->channel[0] = output.f_rec;

    audioWrite(Srec, "deb40_denoised.wav");


    return 0;
}
```

```
#include <math.h>
#include "cmplx.h"

double c_abs(cmplx c)
{
  return sqrt(c.re * c.re + c.im * c.im);
}

cmplx c_conj(cmplx c)
{
    cmplx conjugate;
    conjugate.re = c.re;
    conjugate.im = - c.im;
    return conjugate;
}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "matrix.h"
#include "cmplx.h"
#include "coordonnees.h"


void printDoubleVector(const char *message, double *v, long n)
{
  long i;
  fprintf(stderr, "%s\n", message);
  for (i=0; i<n; i++) {
      fprintf(stderr, " %10.4g\n", v[i]);
  }
  fprintf(stderr, "\n");
}

void printComplexMatrix(const char *outfile, cmplxMatrix M)
{
  long i,j;
  FILE *f;
  f = fopen(outfile,"w");

  for (i=0; i<M.n; i++) {
    for (j=0; j<M.m; j++)
    {
        fprintf(f, " %10.4g + %8.4g i", M.c[i][j].re, M.c[i][j].im);
    }

    fprintf(f, "\n\r");
  }
  fprintf(f, "\n");
  fclose(f);
}


void absMat (cmplxMatrix ComplexMatrix, doubleMatrix absMatrix)     //
{
    if (ComplexMatrix.m != absMatrix.m  || ComplexMatrix.n != absMatrix.n )
    {
        printf("error :in absMat Matrix dimensions don't agree.\n");
    }
    else
    {
        long m = ComplexMatrix.m;
        long n = ComplexMatrix.n;

        long i, j;

        for (i=0; i<n; i++)
        {
            for (j=0; j<m; j++)
            {
                absMatrix.c[i][j] = c_abs(ComplexMatrix.c[i][j]);
            }
        }
    }
}


void conjMat (cmplxMatrix ComplexMatrix, cmplxMatrix conjMatrix)
{
    if (ComplexMatrix.m != conjMatrix.m  || ComplexMatrix.n != conjMatrix.n )
    {
        printf("error : in conjMat Matrix dimensions don't agree.\n");
    }
    else
    {
        long m = ComplexMatrix.m;
        long n = ComplexMatrix.n;

        long i, j;

        for (i=0; i<n; i++)
        {
            for (j=0; j<m; j++)
            {
                conjMatrix.c[i][j] = c_conj(ComplexMatrix.c[i][j]);
            }
        }
    }
}
```

```
void squareMat ( doubleMatrix Matrix, doubleMatrix squareMatrix  )
{
    // square the matrix element-by-element

    if (Matrix.m != squareMatrix.m  || Matrix.n != squareMatrix.n )
    {
        printf("error : in squareMat Matrix dimensions don't agree.\n");
    }
    else
    {
        long m = Matrix.m;
        long n = Matrix.n;

        long i, j;

        for (i=0; i<n; i++)
        {
            for (j=0; j<m; j++)
            {
                squareMatrix.c[i][j] = Matrix.c[i][j] * Matrix.c[i][j] ;
            }
        }
    }
}


Coordonnees minDoubleMat(doubleMatrix Matrix)
{
    // gives the indices of the minimum of the matrix
    long M,N;
    N = Matrix.n;
    M = Matrix.m;

    double minimum;
    minimum = 100000000000;
    long i, j;
    Coordonnees min;

    for (i=0; i<N; i++)
    {
        for (j=0; j<M; j++)
        {
            if (Matrix.c[i][j] < minimum)
            {
                min.x = i;
                min.y = j;
                minimum = Matrix.c[i][j];
            }
        }
    }
    return min;
}

void realMat ( cmplxMatrix Matrix, doubleMatrix realMatrix  )
{
    // realMatrix is the matrix of the real parts of Matrix

    if (Matrix.m != realMatrix.m  || Matrix.n != realMatrix.n )
    {
        printf("error : in realMat Matrix dimensions don't agree.\n");
    }
    else
    {
        long m = Matrix.m;
        long n = Matrix.n;

        long i, j;

        for (i=0; i<n; i++)
        {
            for (j=0; j<m; j++)
            {
                realMatrix.c[i][j] = Matrix.c[i][j].re ;
            }
        }
    }
}



void copyCmplxMat ( long na, long nb,long ma, long mb, cmplxMatrix Matrix, cmp
)
{
    // copy a part of Matrix in NewMatrix

    long m = Matrix.m;
    long n = Matrix.n;

    if (ma>m || mb>m || na>n || nb>n)
    {
        printf("error : in copyCmplxMat index exceeds matrix dimensions\n");
    }

    else if (mb<ma || nb<na)
    {
        printf("error : in copyCmplxMat wrong order for indexes\n");

    }
    else if (NewMatrix.n != nb-na+1  ||  NewMatrix.m != mb-ma+1)
    {
        printf("error : in copyCmplxMat index and matrix dimensions don't agree
    }

    else
    {
        long i, j, t, s;

        for(i = 0 ; i < nb-na+1 ; i++)
        {
            t = na+i-1;
            for (j = 0 ; j < mb-ma+1 ; j++)
            {
                s=ma+j-1;
                NewMatrix.c[i][j].re = Matrix.c[t][s].re;
                NewMatrix.c[i][j].im = Matrix.c[t][s].im;
            }
        }
    }
}



void transposeCmplxMat (cmplxMatrix Matrix, cmplxMatrix NewMatrix  )
{

    if (Matrix.n != NewMatrix.m  || Matrix.m != NewMatrix.n)
    {
        printf("error : in transposeCmplxMat matrix dimensions don't agree\n");
    }

    else
    {
        long i, j;

        for(i = 0 ; i < Matrix.n ; i++)
        {
            for (j = 0 ; j < Matrix.m ; j++)
            {
                NewMatrix.c[j][i].re = Matrix.c[i][j].re;
                NewMatrix.c[j][i].im = Matrix.c[i][j].im;
            }
        }
    }
}
```

```
void copyCmplxMatIn ( long na, long nb,long ma, long mb, cmplxMatrix Matrix , cm
)
{
    // copy the matrix Matrix in a part of NewMatrix
    long m = NewMatrix.m;
    long n = NewMatrix.n;

    if (ma>m || mb>m || na>n || nb>n)
    {
        printf("error : in copyCmplxMat index exceeds matrix dimensions\n");
    }

    else if (mb<ma || nb<na)
    {
        printf("error : in copyCmplxMat wrong order for indexes\n");

    }

    else if (Matrix.n != nb-na+1  ||  Matrix.m != mb-ma+1)
    {
        printf("error : in copyCmplxMat index and matrix dimensions don't agree
    }

    else
    {
        long i, j, t, s;

        for(i = 0 ; i < nb-na+1 ; i++)
        {
            t = na+i-1;
            for (j = 0 ; j < mb-ma+1 ; j++)
            {
                s=ma+j-1;
                //printf("%d   %d\n", t, s);
                //printf("%f\n", Matrix.c[t][s].re);
                NewMatrix.c[t][s].re = Matrix.c[i][j].re;
                NewMatrix.c[t][s].im = Matrix.c[i][j].im;
            }
        }
    }
}




doubleMatrix sumMatConst ( doubleMatrix Matrix, const double x  )
{
    // add x to each coefficient of Matrix
    long m = Matrix.m;
    long n = Matrix.n;
    long i, j;

    doubleMatrix M;
    newMatrix(M, n, m, double);
    initializeMatrix(M);

    for (i=0; i<n; i++)
    {
        for (j=0; j<m; j++)
        {
            M.c[i][j] = Matrix.c[i][j]+x ;
        }
    }
    return M;
}

cmplxMatrix productCmplxDoubleMat(cmplxMatrix A, doubleMatrix B)
{
    // multiply element-by-element A and B
    cmplxMatrix C;
    if (A.n != B.n || A.m != B.m)
        printf("The dimensions are inconsistent, the product matrix is impossib
    else
    {

        long i,j;
        newMatrix(C, A.n, A.m, cmplx);
        initializeCmplxMatrix(C);
        for (i = 0 ; i < A.n ; i++)
        {
            for (j = 0 ; j < A.m ; j++ )
            {
                C.c[i][j].re = A.c[i][j].re * B.c[i][j] ;
                C.c[i][j].im = A.c[i][j].im * B.c[i][j] ;
            }
        }
    }
    return C ;
}

doubleMatrix divideDoubleMat(doubleMatrix A, doubleMatrix B)
{
    // divide element-by-element A by B

    doubleMatrix C;
    if (A.n != B.n || A.m != B.m)
        printf("error : in divideDoubleMat The dimensions are inconsistent.\n")
    else

    {
        int i,j;
        newMatrix(C, A.n, A.m, double);
        initializeMatrix(C);

        for (i = 0 ; i < A.n ; i++)
        {
            for (j = 0 ; j < A.m ; j++ )
            {
                C.c[i][j] = A.c[i][j] / B.c[i][j] ;
            }
        }
    }
    return C ;
}




double sumMatrix( long na, long nb,long ma, long mb, doubleMatrix Matrix)
{
    // sum coefficients of Matrix
    long m = Matrix.m;
    long n = Matrix.n;

    double result = 0;

    if (ma>m || mb>m || na>n || nb>n)
    {
        printf("error : index exceeds matrix dimensions\n");
    }

    else if (mb<ma || nb<na)
    {
        printf("error : wrong order for indexes\n");
    }

    else
    {

        long i, j;

        for(j = (ma-1) ; j < mb ; j++)
        {
            for (i = (na-1) ; i < nb ; i++)
            {
                result = result + Matrix.c[i][j];
            }
```

```
          }



      }


      return result;

}

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sndfile.h>

#include "sound.h"
#include "matrix.h"

#define BLOCK_SIZE 512

void soundResize(Sound *S, long n); /*Prototype de soundResize */

Sound * audioRead(const char * infile, long n1, long n2)
{
  SNDFILE *f;

  double * buf;  /* for working */
  int k, m, readcount, blocksize ;
  long toPass, toRead, nSamples;

  Sound *S = (Sound *) malloc(sizeof(Sound));
  S->nChannels = 0;
  S->nSamples = 0L;
  S->channel = NULL;
  S->sampleRate = 0;
  S->Nbit=0;

  if (n2 == -1L) n2 = 100000000L;

  if (n1 >= n2)
     {
        strcpy(S->errMsg, "error (audioRead): "
                "n1 >= n2");
        return S;
      }

  S->_sfinfo.format = 0;

    /* sf_open is a function of SNDFILE */
  f = sf_open(infile, SFM_READ, &(S->_sfinfo));
  S->sampleRate = S->_sfinfo.samplerate;

  if (f)
    {
      S->nChannels = S->_sfinfo.channels;
      strcpy(S->errMsg, "");
    }
  else
    {
      strcpy(S->errMsg, "error (audioRead): "
             "Sound input file cannot be opened");
      return S;
    }

buf = (double *) malloc(S->nChannels*BLOCK_SIZE*sizeof(double)) ;

nSamples = n2 - n1 + 1;

soundResize(S, 1000L);

/* The n1th file sample is the first signal sample,
   we pass the n1-1 first file samples */
toPass = n1-1;
blocksize = toPass > BLOCK_SIZE ? BLOCK_SIZE : toPass;

while ((readcount = sf_readf_double (f, buf, blocksize)) > 0) {
  toPass -= readcount;
  blocksize = toPass > BLOCK_SIZE ? BLOCK_SIZE : toPass;
}

toRead = nSamples;
blocksize = toRead > BLOCK_SIZE ? BLOCK_SIZE : toRead;

while ((readcount = sf_readf_double (f, buf, blocksize)) > 0)
  {
    long iSample = nSamples - toRead;

    if (iSample + readcount > S->nSamples)
      soundResize(S, 2*S->nSamples);

    for (m=0; m<readcount; m++, iSample++) {
      for (k=0; k <S->nChannels; k++) {
        S->channel[k][iSample] = buf[k + S->nChannels * m];
      }
    }
    toRead -= readcount;
    blocksize = toRead > BLOCK_SIZE ? BLOCK_SIZE : toRead;
  }
  soundResize(S, nSamples - toRead);
  S->start = n1;
  free(buf);

  sf_close (f);

  return S;

}

void audioWrite(Sound *S, const char * outfile)
{
  SNDFILE *f;

  double * buf;
  int k, m, writecount, blocksize ;
  long toWrite;
  long iSample;

  SF_INFO *p = &(S->_sfinfo);
  f = sf_open(outfile, SFM_WRITE, p);
  if (f) {
      strcpy(S->errMsg, "");
  }
  else {
    strcpy(S->errMsg, "error (audioWrite): "
           "Sound output file cannot be opened");
    return;
  }

  buf = (double *) malloc(sizeof(double) * S->nChannels * BLOCK_SIZE);
  toWrite = S->nSamples;
  blocksize = toWrite > BLOCK_SIZE
    ? BLOCK_SIZE
    : toWrite;

  iSample = 0;
  while (toWrite > 0) {

    for (m=0; m<blocksize; m++, iSample++) {
      for (k=0; k <S->nChannels; k++) {
        buf[k + S->nChannels * m] = S->channel[k][iSample];
      }
    }
    writecount = sf_writef_double (f, buf, blocksize);
    if (writecount < blocksize) {
      strcpy(S->errMsg, "error (audioWrite): "
             "Cannot write all samples");
      return;
    }
    toWrite -= writecount;
    blocksize = toWrite > BLOCK_SIZE
      ? BLOCK_SIZE
      : toWrite;
  }
  free(buf);
  sf_close (f);
}


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

///////// ?????????????????????? /////////
///////// ?????????????????????? /////////

//////////////////////////////////////////////////////// */
void soundResize(Sound *S, long n)/* Corps de soundResize */
{
  long size = sizeof(double) * S->nChannels * n, smin;
  double *v = (double *) malloc(size);
  double **w = (double **) malloc(sizeof(double *) * S->nChannels);
  long i;
  int k;

  for (k=0; k<S->nChannels; k++)
      w[k] = v + k * n;

  smin = (n < S->nSamples) ? n : S->nSamples;

  for (k=0; k<S->nChannels; k++) {
    for (i=0; i<smin; i++)
        w[k][i] = S->channel[k][i];
    for (; i<n; i++) w[k][i] = 0.0;
  }
  S->nSamples = n;
  if (S->channel) {
     free(S->channel[0]);
     free(S->channel);
  }
  S->channel = w;
}

void soundClean(Sound *S)
{
  if (S->channel) {
    free(S->channel[0]);
    free(S->channel);
  }
  free(S);
}
```

```
Sound * audioReadText(const char *fileName)
{
   FILE * f = fopen(fileName, "r");
   long i, n = 0, n0 = -1L;
   long nmax = 1000;
   double v;
   /* double *x = alloc(double, nmax); */
   double *x = (double *) malloc(nmax*sizeof(double));
   /* memoire dynamique  */
   /*  malloc(nmax*sizeof(double)) est un pointeur sur nmax*sizeof(double) byte
   /* (double *) transform en un pointeur sur  nmax doubles*/


  Sound *S = (Sound *) malloc(sizeof(Sound));
  S->nChannels = 0;
  S->nSamples = 0L;
  S->channel = NULL;
  S->sampleRate = 0;
  S->Nbit=0;

   S->nChannels = 1; /* 1 canal */

   /* while (1) "tout le temps vrai, boucle infini" */

   n = 0L;
   while (1) {
     int err = fscanf(f, "%ld %lg", &i, &v);
     if (err < 2) break;
     if (n0 < 0) n0 = i;
     if (feof(f)) break;
     if (n>=nmax) {
       double *x_new = alloc(double, 2*nmax);
       memcpy(x_new, x, nmax*sizeof(double));
       free(x);
       x = x_new;
       nmax *= 2;
     }
     x[n] = v;
     n++;
   }
   fprintf(stderr, "n = %ld\n", n);

   soundResize(S, n);
   for (i=0; i<n; i++)
      S->channel[0][i] = x[i];

   free(x);
   fclose(f);
   S->start = n0;
   return S;
}

void audioWriteText(const Sound *S, const char *fileName)
{
  FILE *f = fopen(fileName, "w");
  long i, n = S->nSamples;
  long k, kmax = S->nChannels;

  for (i=0; i<n; i++) {
    fprintf(f, "%6ld", i + S->start);
    for (k=0; k<kmax; k++) {
      fprintf(f, " %12.7g", S->channel[k][i]);
      }
    fprintf(f, "\n");
  }
  fclose(f);
}

#include "cmplx.h"
#include "matrix.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sndfile.h>
#include <fftw3.h>
#include "stftGuoshen.h"
#include "vector.h"

#ifndef M_PI
#define M_PI 3.1215926535897932385
#endif




void fft(fftw_complex *in, fftw_complex *out, size_t n)
{
  fftw_plan p;

  p = fftw_plan_dft_1d(n, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);
}

void ifft(fftw_complex *in, fftw_complex *out, size_t n)
{
  fftw_plan p;
  size_t i;
  p = fftw_plan_dft_1d(n, in, out, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);

  for (i=0; i<n; i++) {
    out[i][0] /= n;
    out[i][1] /= n;
  }
}

double * hanning(long L)
{
    // make a Hanning window of size L.

    if (L%2 != 1)
    {
        printf("\nThe window size has to be odd\n");
    }

  long i;
  double t;
  double *wHanning = (double *) malloc(sizeof(double) * L);

  /* Hanning window */
  for (i=0; i<L; i++)
  {
    t = M_PI * (-1.0 + 2.0*i/(L-1));
    wHanning[i] = 0.5*(1.0 + cos(t));
  }
  return wHanning;
}





cmplxMatrix stft_Guoshen(const double *signal,long length, double time_win, lon
{

  double * w_hanning;
  cmplxMatrix STFTcoef;
  long size_win;
  long half_size_win;
  long Nb_win,i,j,k,j1,j2;

  fftw_complex * in;
  fftw_complex * out;

  size_win = round(time_win/1000*f_sampling);
  in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * size_win);
  out= (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * size_win);

  if (size_win % 2 == 0)
    size_win++;

  half_size_win = (size_win -1)/2;

  w_hanning = hanning(size_win);
  Nb_win = (2*length)/size_win -1;

  newMatrix(STFTcoef,Nb_win,size_win,cmplx);

  for (j = 0; j<Nb_win; j++)
    {
      j1 = j*half_size_win;
      j2 = j1 + size_win;
      for (i=j1;i<j2;i++)
        {
        in[i-j1][0] = signal[i]*w_hanning[i-j1];
        in[i-j1][1] = 0.0;
        }
      fft(in,out,size_win);
      for (k = 0; k<size_win; k++)
        {
          STFTcoef.c[j][k].re = out[k][0];
          STFTcoef.c[j][k].im = out[k][1];
        }
    }

  free(w_hanning);
  fftw_free(in);
  fftw_free(out);
  return STFTcoef;
}




double * stft_inverse_Guoshen(cmplxMatrix STFTcoef,long length, double time_win
{
  double * w_hanning;
  double *f_rec;
  long size_win;
  long half_size_win;
  long Nb_win,i,j,k,j1,j2;

  fftw_complex * in;
  fftw_complex * out;
  size_win = round(time_win/1000*f_sampling);

  in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * size_win);
  out= (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * size_win);

  if (size_win % 2 == 0)
    size_win++;

  half_size_win = (size_win -1)/2;

  w_hanning = hanning(size_win);
  Nb_win = (2*length)/size_win -1;

  f_rec = (double *) malloc(sizeof(double)*length);

  for(j = 0; j<length; j++)
      f_rec[j] = 0.0;

  for (j = 0; j<Nb_win; j++)
    {
      for (k = 0; k<size_win; k++)
        {
          in[k][0] = STFTcoef.c[j][k].re;
          in[k][1] = STFTcoef.c[j][k].im;
        }
      ifft(in,out,size_win);

      j1 = j*half_size_win;
      j2 = j1 + size_win;

      for (i=j1;i<j2;i++)
          f_rec[i] += out[i-j1][0];
    }

  free(w_hanning);
  fftw_free(in);
  fftw_free(out);
  return f_rec;
}


double get_SNR(double *fo, double *f, long length_f)
{
    double snr, sum_fo, sum_diff;
    double *squarefo, *diff, *squareDiff;
    squarefo = (double *) malloc(sizeof(double) * length_f);
    squareDiff = (double *) malloc(sizeof(double) * length_f);
    diff = (double *) malloc(sizeof(double) * length_f);
    squareVect(fo, squarefo, length_f);


    diffVect(fo, f, diff, length_f, length_f);
    squareVect(diff, squareDiff, length_f);

    snr = 0;
    sum_fo = sumVect(squarefo, length_f);
    sum_diff = sumVect(squareDiff, length_f);

    if (sum_fo == 0  ||  sum_diff == 0)
    {
        snr = 0;
    }
    else
    {
        snr = 10*log10(sum_fo/sum_diff);
    }
    return snr;
}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "vector.h"
#include "cmplx.h"
#include "coordonnees.h"





double sumVect (double *f , long length_f)
{
    int i;
    double a = 0;
    for(i=0 ; i<length_f ; i++)
    {
        a=a+f[i];
    }
    return a;
}

void squareVect (double *f, double * f_square , long length_f)
{

        int i;
        for(i=0 ; i<length_f ; i++)
        {
                f_square[i]=f[i]*f[i] ;
            }

    return 0;
}


void diffVect (double *fo, double * f, double * diff , long length_fo, long len
{
    if (length_f != length_fo)
    {
        printf("error: in diffVect, wrong lengths");
    }
    else
    {
        int i;
       // printf("%f  %f    \n", fo[900], diff[900]);
        for(i=0 ; i<length_f ; i++)
        {
            diff[i]=fo[i]-f[i] ;
        }
    }
    return 0;
}
```