[Date]

Guénon Marie et Favreau Jean-Dominique

VIM / master SSTIM

Compression des images numériques

Compte rendu des travaux dirigés

Table des matières

[Préambule : 2](#_Toc374540557)

[Librairies : 3](#_Toc374540558)

[TD1 4](#_Toc374540559)

[1. Filtre de Haar, Analyse 4](#_Toc374540560)

[Méthode directe 4](#_Toc374540561)

[Filtrage 6](#_Toc374540562)

[2. Reconstitution, Synthèse 7](#_Toc374540563)

[Inversion directe 7](#_Toc374540564)

[Synthèse 8](#_Toc374540565)

[3. Récursivité 9](#_Toc374540566)

[Analyse 9](#_Toc374540567)

[Synthèse 10](#_Toc374540568)

[TD2 11](#_Toc374540569)

[1. Quantificateur scalaire 11](#_Toc374540570)

[2. Caractéristique Entrée / Sortie 12](#_Toc374540571)

[3. Distorsion 13](#_Toc374540572)

[TD3 15](#_Toc374540573)

[1. Quantification 15](#_Toc374540574)

[2. Entropie 15](#_Toc374540575)

[3. Reconstruction 16](#_Toc374540576)

**But :** réaliser une transformée en ondelettes (analyse / synthèse) au moyen du filtre de Haar.

# Préambule :

Après une première approche avec Scilab, nous avons préféré utiliser le langage de programmation C++ pour implémenter ce projet. En effet, ce langage nous permet d’avoir une approche simplifiée du traitement d’image ainsi que des calculs optimisés : les calculs matriciels et les itérateurs sont déjà implémentés et optimisés pour le genre de calculs que nous allons devoir effectuer. Nous obtiendrons donc ainsi de meilleurs résultats, autant au niveau qualitatif, qu’au niveau de la rapidité d’obtention.

# Librairies :

De manière native, C++ ne sait pas charger d’image. Nous avons donc du installer la librairie :

**OpenCv2**



Cette librairie permet de charger, afficher et traiter les images.



# TD1

## Filtre de Haar, Analyse

### Méthode directe

Une fois l’image à traiter chargée, nous allons la traiter. Pour ce faire, nous allons la parcourir par bloc de quatre pixels, que l’on nommera. On applique alors les quatre filtres de Haar à chaque point considéré de notre image de départ :

* Moyenne des pixels avoisinants, on applique le filtre : et on obtient l’image résultante :





* Gradient X, on applique le filtre : et on obtient l’image résultante :





* Gradient Y, on applique le filtre : et on obtient l’image résultante :





* « Gradient diagonal », on applique le filtre : et on obtient l’image résultante :





Ce qui donne, une fois touts les filtres réunis ensembles, on obtient[[1]](#footnote-1) :



### Filtrage

Avant de filtrer l’image, nous devons lui appliquer un traitement préalable. En effet, pour éviter les effets de bords et les problèmes aux limites, nous devons insérer une ligne et une colonne supplémentaire à la fin de l’image. Pour cela, nous insérons par miroir (copie du pixel précédent) ces lignes à droite et en bas de l’image.

Après quoi nous appliquons les quatre filtres vus précédemment mais sur l’intégralité de l’image :

* Moyenne des pixels avoisinants:



* Gradient X   :



* Gradient Y :



* « Gradient diagonal »  :



Pour finir, nous effectuons le sous échantillonnage sur les quatre filtres obtenus : nous prenons un pixel sur deux en fonction du filtrage effectué : Nous considérons ici que le numéro du premier pixel est (0,0)

* Moyenne des pixels avoisinants:  
  On prend les colonnes et les lignes paires.
* Gradient X  :  
  On prend les colonnes impaires et les lignes paires.
* Gradient Y :   
  On prend les colonnes paires et les lignes impaires.
* « Gradient diagonal »  :  
  On prend les colonnes et les lignes impaires.

## Reconstitution, Synthèse

### Inversion directe

Comme vu précédemment, nous avons une image de départ et nous l’avons parcours par blocs de quatre pixels que nous avons nommé. Cette image a été analysée en quatre images grâce aux filtres décrits ci-dessus :

Pour reconstituer l’image de départ, Il suffit d’inverser le système présenté ici et nous obtenons les formules suivantes :



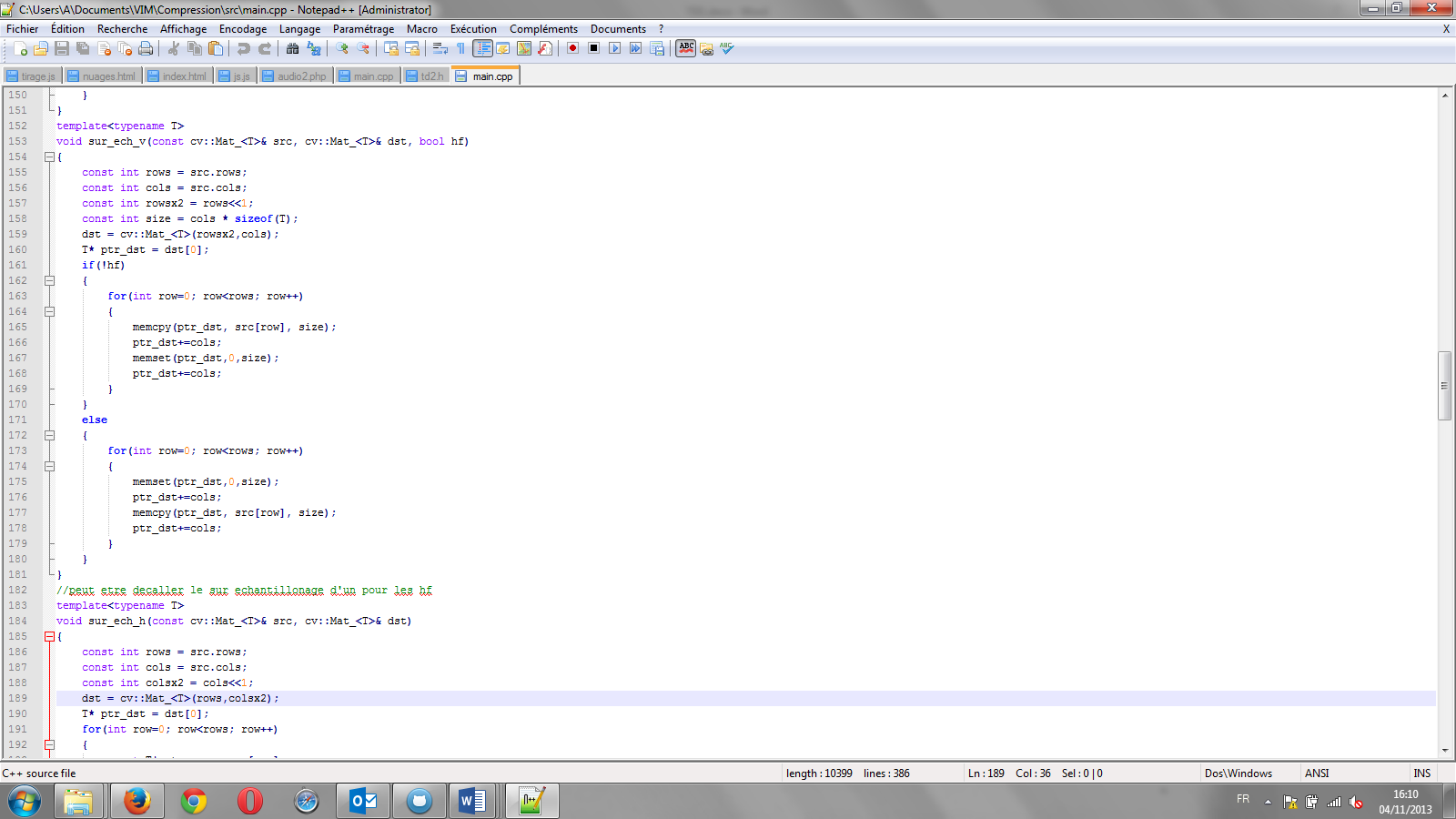
En appliquant ces formules sur les filtres obtenus précédemment, on obtient à gauche l’image d’origine et à droite l’image reconstruite :

### Synthèse

De la même manière que dans l’analyse de l’image, avant de filtrer l’image, nous devons lui appliquer un traitement préalable. En effet, pour éviter les effets de bords et les problèmes aux limites, nous devons insérer une ligne et une colonne supplémentaire à la fin de l’image. Pour cela, nous insérons par miroir (copie du pixel précédent) ces lignes à droite et en bas de l’image.

Après quoi, on sur-échantillonne les quatre images. Nous ajoutons des pixels noirs là où on les avait enlevés précédemment :



Sur-échantillonnage vertical

Ensuite, nous appliquons les filtres :

* Filtre basse fréquence



* Filtre haute fréquence



En sommant les résultats obtenus, on obtient l’image d’origine à et à droite l’image reconstruite :

## Récursivité

### Analyse

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre d’itération | Résultat obtenu | Nombre d’itération | Résultat obtenu |
| 0 |  | 3 |  |
| 1 |  | 4 |  |
| 2 |  | 5 |  |

## 

### Synthèse

Nous avons appliqué cinq fois notre synthétiseur (méthode directe) sur l’image obtenu après analyse et nous avons eu le résultat suivant :

Pour vérifier la cohérence de nos résultats, nous avons de plus effectué un calcul de distance entre les deux images (norme de la différence) et nous avons constaté que celle-ci était nulle. Et donc nous pouvons dire que nos résultats sont bons.

A droit image originale, à gauche image obtenue après analyse et synthèse :

# TD2

## Quantificateur scalaire

Nous cherchons ici à construire un quantificateur scalaire uniforme à L niveaux de quantification comme suit :

0

*val*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x |  |  |  | x |  |
|  |  |  |  |  |  |

min max

L

Sur cet exemple, min et max sont les bornes des valeurs atteignables. Nous avons de plus ici six intervalles, L est la longueur de chacun de ces intervalles. *val* est une valeur quelconque de l’intervalle [min, max]. Ici, *val* est sur le quatrième intervalle (nous comptons à partir de zéro).  
De manière générale, nous cherchons à calculer le représentant de chaque classe *i* (numéro d’intervalle) telle que 0 soit centré au milieu d’un intervalle. Pour cela, nous calculons le pas de quantification centré en zéro:

A partir de là, nous pouvons calculer l’indice de la classe contant la valeur *val*:

Où E(.) est la partie entière. D’où nous obtenons la valeur du représentant de *val*:

Soit définit de manière itérative tel que :

(1)

## Caractéristique Entrée / Sortie

Nous cherchons à avoir un quantificateur centré en zéro, pour cela, on doit fixer les bornes dans lesquelles on va l’appliquer. C'est-à-dire, nous calculons les valeurs suivantes :

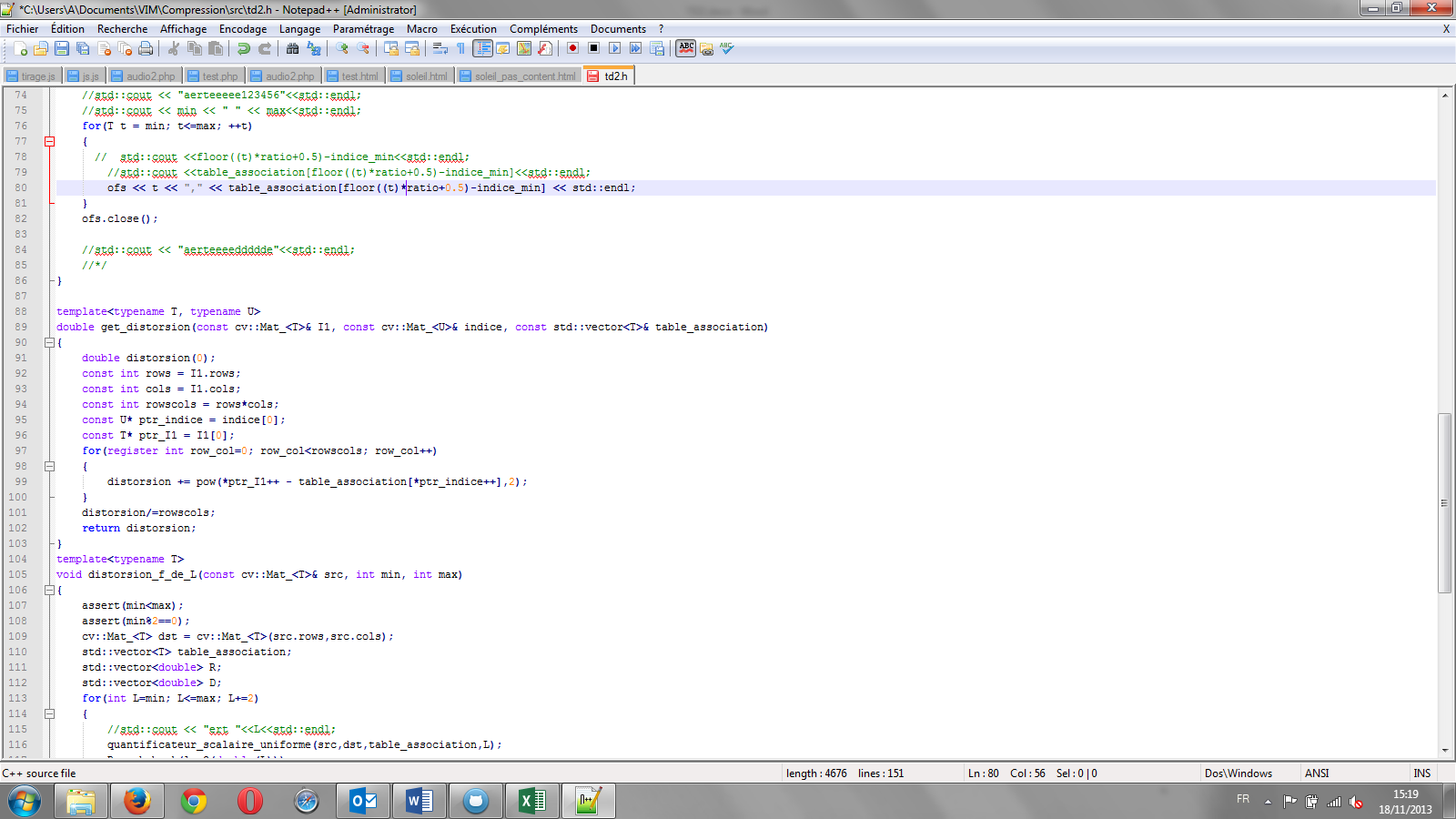
Ensuite, nous pouvons appliquer notre algorithme définit en (1) ce qui nous donne les résultats suivants pour trois nombres de niveaux différents :

Nous pouvons constater que les caractéristiques obtenues coupent bien les axes en (0,0) ce qui est cohérent avec ce qui était demandé. De plus, nous pouvons voir que la tailles des pas sont homogènes, hormis au tout début et à la toute fin de la courbe. Ce qui est logique puisque notre système est sur-contraint et c’est là que se répercute la variable liée (en trop). Ceci est aussi lié au fait que nous avons forcé la taille de l’intervalle du quantificateur comme entier (pour éviter les problèmes d’arrondis et que chaque pas soit de même longueur).

## Distorsion

Nous avons la formule de distorsion suivante :

Où est l’image de référence et est l’image comparée. Ce qui nous donne l’algorithme suivant :



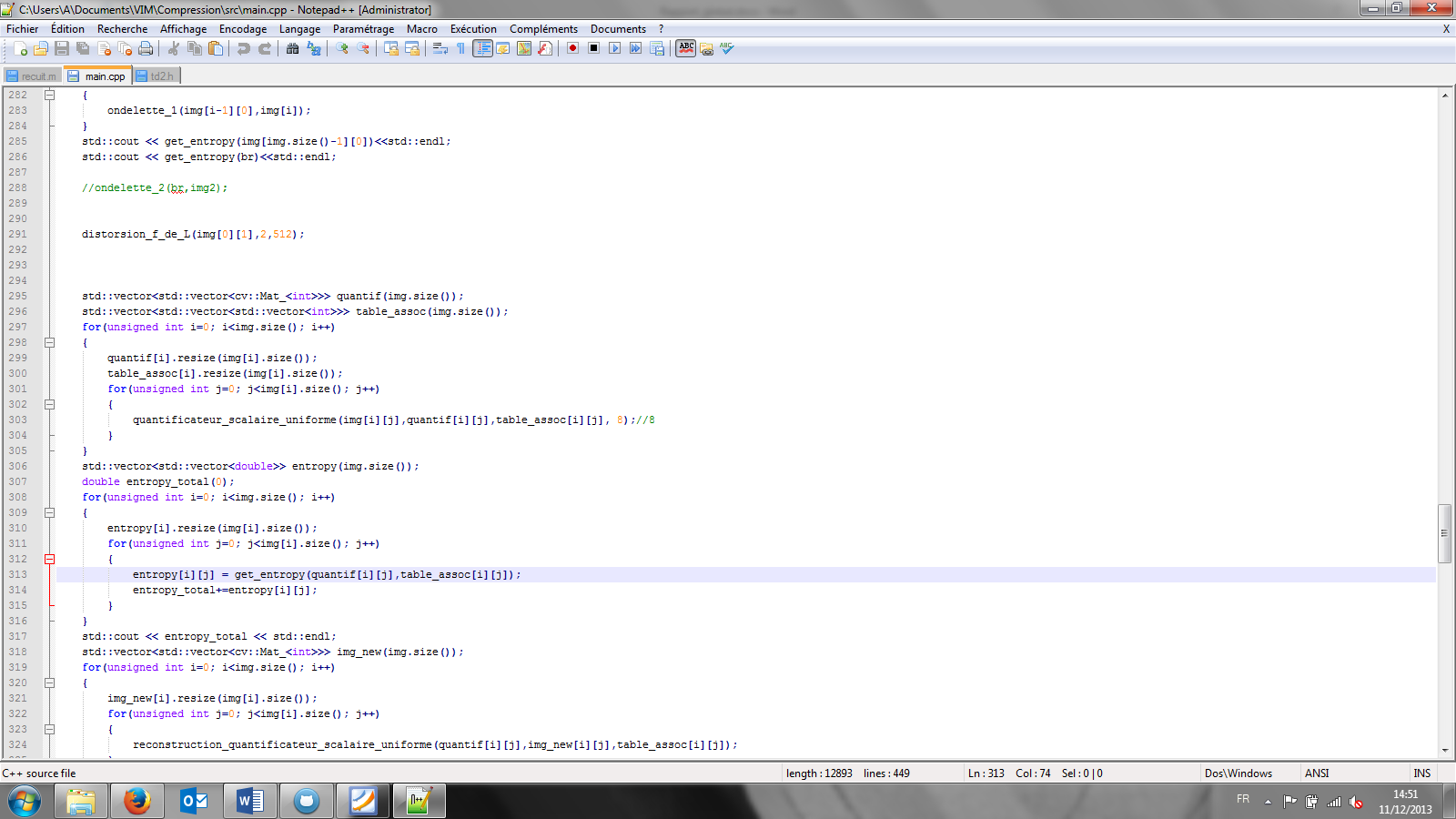
En appliquant notre méthode sur une sous-bande de coefficients d’ondelettes obtenus sur une image de Lena, nous obtenons la courbe suivante pour les L pairs :

Pour le traitement suivant de la distorsion, nous auront besoin de bits entiers et donc de garder la partie la partie entière supérieur des résultats obtenus :

# TD3

## Quantification

La quantification mise en œuvre ici est la même que celle utilisée au [TD2.1](#_Quantificateur_scalaire). Et nous l’appliquons maintenant à chaque sous bande de l’image :



## Entropie

La formule de l’entropie est la suivante :

Où est le nombre d’éléments codé par le représentant de la classe *i*, divisé par le nombre de représentants total (probabilité d’apparition du représentant), c'est-à-dire   
En appliquant cette formule, nous obtenons l’algorithme suivant :

L’entropie totale suit elle la formule suivante :

Le taux de compression est donné par 100 moins la division entre le nombre de bits présents dans l’image obtenue (donné par H*tot*) et le nombre de bits présents initialement dans l’image (donné par l’entropie de l’image initiale), c'est-à-dire :

## Reconstruction

La transformée en ondelette inverse utilisée ici est celle que nous avons développée au [TD1.3](#_Inversion_directe). Nous mesurons alors son PSNR à partir de la formule suivante :

Où :

# TD4

## Allocation de débit

On initialise la pente λ que nous cherchons, la précision ε avec laquelle on veut notre résultat et le débit maximal autorisé RSB.

On cherche les points *Ri* vérifiant l’équation :

(\*)

On calcule le débit total :

On vérifie la distance du débit total calculé par rapport au débit maximal autorisé :

Si cette inégalité est respectée, on considère qu’il y a convergence. Sinon, on choisit une autre valeur de λ et on relance l’algorithme à l’étape (\*)

Wi = 1 (sur haar)

Ai = 1/2^2i

D(r)=d^sigma2(r)

Si |lambda| augmente, plus on augmente la compression

1. Les images qui sont affichées ici ont été retouchées pour pouvoir être visualisable. En effet, une fois les filtres appliqués, les valeurs obtenues ne sont pas des images et doivent être translatées et étirées entre 0 et 255. [↑](#footnote-ref-1)