TIPE - Compression d'images

Robin Champenois

Juin 2012

Table des matières

1 Introduction

Avec l'évolution technologique, nous sommes amenés à manipuler des données de plus en plus importantes, à les échanger et les stocker. Bien que les capacités de stockage et le débit internet augmentent eux aussi, trouver comment diminuer efficacement la taille prise par ces données reste d'un intérêt majeur.

C'est notamment le cas des images et des photos : aujourd'hui, la moindre photo contient plus de 5 millions de pixels, ce qui, sans compression, ferait des fichiers de plus de 15 Mo.

Nous allons donc nous intéresser à plusieurs méthodes de compression d'images assez utilisées.

1.1 Généralités

En informatique, toutes les données se présentent à la base comme une suite d'entiers écrits en base 2. La convention de l'octet implique que ces entiers sont codés sur 8 bits, et ont donc une valeur entre 0 et 255.

Une image usuelle (non transparente) se présente sous la forme d'une matrice de pixels, encodés sur trois octets : un pour chaque canal (Rouge, Vert, Bleu).

Notre but est donc de trouver des techniques pour encoder différemment une image...

2 Codage entropique

Avant de considérer des images, nous allons voir deux algorithmes usuels de compression de données.

Pour simplifier la lecture, les symboles considérés seront des lettres.

2.1 Run-length-encoding

L'algorithme RLE repose sur une idée assez simple : au lieu de répéter plusieurs fois un même symbole, on indique le nombre de fois qu'on le répète, puis ce symbole (une seule fois). Pour des données très redondantes, on peut ainsi gagner facilement de la place.

S'il y a peu de doublons à la suite, on a un problème : la taille peut être jusqu'à deux fois plus grande ! Il faut donc améliorer l'idée : un entier inférieur à 128 indique le nombre de répétition de la lettre suivante ; un entier supérieur à 128 indique le nombre de lettres (+ 128) qu'il y a à lire après, sans répétition (voir annexe A).

On est alors sûr de ne pas alourdir le fichier, et on peut encore décoder sans problème.

La compression RLE est utilisée pour les images .bmp. Mais elle présente, dans l'immédiat, peu d'intérêt : il n'y a jamais deux pixels consécutifs de même couleur dans une photo.

2.2 Algorithme de Huffman

L'algorithme de Huffman est plus complexe : il s'agit de travailler au niveau binaire, et d'utiliser moins de bits pour coder les symboles qui reviennent le plus souvent. On va construire un arbre binaire où les feuilles sont les symboles de la table, et où les branches les plus profondes portent les symboles les plus rares.

On construit d'abord une file de priorité où les priorités sont les fréquences d'apparition des symboles, en tant que feuilles d'un futur arbre. (Les premiers sortis de la file seront les symboles les moins fréquents).

Tant qu'il y a plus de deux éléments dans la file : on sort les deux premiers éléments de la file g et d (c'est-à-dire les moins fréquents), on construit un Nœud (g,d), qu'on rajoute dans la file avec comme priorité la somme des priorités de g et d.

On a alors un arbre, dont on indice les branches de la sorte : 0 pour une branche gauche, 1 pour une branche droite.

Enfin, on remplace les symboles du message à coder par la suite de bits nécessaires pour les atteindre dans l'arbre (voir annexe A).

Pour écrire ce code dans un fichier, il suffit d'écrire la lecture préfixe de l'arbre (avec un « identifiant » pour distinguer Feuille et Nœud), puis la suite de bits (complétés du nombre de zéros nécessaires pour avoir un nombre entiers d'octets, en indiquant ce nombre avant).

La décompression s'effectue en reconstituant l'arbre, puis la suite de bits associée à la suite d'entiers, puis en lisant l'arbre (comme un automate).

Cet algorithme est plus fort que celui de RLE : il n'a pas besoin d'une suite consécutive de symboles identiques pour compresser. Il est très utilisé pour la compression de fichier en général (zip...).

Il est cependant peu efficace en l'état sur une image quelconque : les fréquences d'apparition des valeurs sont assez proches.

2.3 Conclusion

Ces algorithmes sont assez puissants, mais leur efficacité est limitée sur des images : le problème est que la représentation sous forme de pixels introduit une trop grande disparité

entre les valeurs voisines.

On remarque aussi que ces algorithmes sont sans perte : l'information après décompression est exactement la même que celle avant compression.

L'information contenue dans un pixel de l'image est assez faible : si la couleur du pixel varie un peu, l'image globale en sera peu affectée, on verra peu de différence.

L'idée de la compression d'image est alors de trouver une autre manière de caractériser l'image, et de sacrifier des détails peu visibles pour l'œil humain afin de diminuer la masse d'informations.

3 Compression JPEG

3.1 Transformée cosinus discrète (DCT)

La compression d'image JPEG repose sur la transformée cosinus discrète (DCT).

3.1.1 En dimension 1

La DCT de $(s_1...s_n) \in \mathbb{R}^n$ est donnée par :

$$\forall j \in [\![1,n]\!], S_j = C_j\sqrt{\frac{2}{n}} \times \sum_{k=0}^{n-1} s_k \cos\frac{(2k+1)j\pi}{2n}$$

avec $C_0 = \frac{1}{\sqrt{2}}$ et $C_k = 1$ pour $k > 0$

$(S_1...S_n)$ est l'écriture de $(s_1...s_n)$ sur la base orthonormée $(e_1...e_k)$, où :
$\forall j \in [\![1,n]\!], e_j = \left(C_j\sqrt{\frac{2}{n}}\cos\frac{(2k+1)j\pi}{2n}\right)_{k\in[\![1,n]\!]}$

Notons $M = [(e_1...e_n)]_{(\varepsilon)}$ (où (ε) est la base canonique de $\mathbb{R}^n$) : $M \in \mathcal{O}_n(\mathbb{R})$

$$\begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix} = {}^tM \times \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix}$$

La transformation inverse est donc $\begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} = M \times \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix}$:

$$s_k = \sqrt{\frac{2}{n}} \times \sum_{j=0}^{n-1} C_j S_j \cos\frac{(2k+1)j\pi}{2n}$$

3.1.2 En dimension 2

On applique à la fois la DCT aux lignes et aux colonnes d'une matrice (n,m) $A = [a_{i,j}]$:

$$S_{i,j} = \frac{2}{\sqrt{nm}} C_i C_j \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_{k,l} \cos\frac{(2k+1)i}{2n}\pi \cos\frac{(2l+1)j}{2m}\pi$$

Les vecteurs de la nouvelle base sont des images élémentaires, plus significatives qu'un seul pixel (voir annexe B).

3.2 Application en compression d'image

On a trouvé une base plus pertinente que la base canonique des matrices. Les vecteurs de la base de la DCT correspondent à des fréquences de plus en plus élevées.

Pour une image, on remarque que ces « hautes fréquences » correspondent à des détails auquel l'œil humain est peu sensible. On peut donc se permettre des approximations concernant leur valeur, afin de compresser un fichier.

Le JPEG utilise un autre format colorimétrique que le RVB : le format YUV. Y est la luminance (approximativement, la clarté du pixel), U la chrominance bleue (approximativement la différence entre canal bleu et la luminance), et V la chrominance rouge. L'œil est moins sensible aux variations de chrominance : on peut se permettre de plus grandes approximations.

Pour rendre les calculs plus rapides, la compression JPEG découpe ensuite l'image en carreaux de 8x8 pixels, auxquels elle applique la DCT.

On applique ensuite une quantification des valeurs, plus ou moins forte selon le niveau de compression et la fréquence associée. Pour cela, on divise chaque coefficient par une valeur donnée, et on arrondit le résultat. C'est à ce niveau-là que se situent les pertes.

Enfin, on linéarise l'image en lisant chaque carreau dans l'ordre croissant des fréquences (voir annexe B), on réalise une compression RLE, et une compression Huffman avant d'écrire dans le fichier.

En général, les hautes fréquences sont peu représentées : à part les premières valeurs, après quantification, il y a beaucoup de coefficients nuls. La compression RLE est alors utile pour éliminer ces nombreux 0 consécutifs. En fonction de la quantification, les valeurs trouvées peuvent être assez peu variées : la compression Huffman peut alors efficacement réduire le poids du fichier.

4 Compression par ondelettes

On utilisera par la suite le produit scalaire usuel de $L^2(\mathbb{R})$: $\langle f|g\rangle = \int_{\mathbb{R}} fg$

4.1 Analyse multi-résolution

Une analyse multi-résolution [1] de $L^2(\mathbb{R})$ est une suite $(V_j)_{j\in\mathbb{Z}}, V_j \subset L^2(\mathbb{R})$ telle que :

- $\forall j \in \mathbb{Z}, V_j \subset V_{j+1}$
- $\bigcap_{j\in\mathbb{Z}} V_j = \{0\},\ \overline{\bigcup_{j\in\mathbb{Z}} V_j} = L^2(\mathbb{R})$
- $f(\bullet) \in V_j \Leftrightarrow f(2\bullet) \in V_{j+1}$
- $f(\bullet) \in V_j \Leftrightarrow \forall m \in \mathbb{Z}, f(\bullet + m2^{-j}) \in V_j$
- Il existe $\phi \in V_0$ à support compact et $(\alpha_k)_{k\in\mathbb{Z}} \in \ell^2(\mathbb{N})$, telle que l'on ait l'équation de raffinement :

$$\forall x \in \mathbb{R}, \phi(x) = \sum_{k\in\mathbb{Z}} \alpha_k \phi(2x-k)$$

et tel que $(\phi(\bullet - k))_{k\in\mathbb{Z}}$ est une base de Riesz [2] de V_0

ϕ est fonction échelle de l'analyse multi - résolution.

Une fonction ondelette mère ψ est telle que : $(\psi(\bullet - k))_{k\in\mathbb{Z}}$ est libre, et

$\text{Vect}(\psi(\bullet - k))_{k\in\mathbb{Z}} \oplus V_0 = V_1$

4.2 Théorie discrète

On va décomposer notre signal avec l'analyse multirésolution.

4.2.1 Ondelettes - cas général

Formellement, considérons une analyse multirésolution sur $(V_j)_{j\in\mathbb{Z}}$, une fonction d'échelle ϕ et une ondelette mère ψ

Notons, pour tout $p \in \mathbb{Z}, j \in \mathbb{Z}$, $\phi_{p,j} : x \mapsto 2^{p/2}\phi(2^p x - j)$ et $\psi_{p,j} : x \mapsto 2^{p/2}\psi(2^p x - j)$

On suppose qu'il existe $N = (2P+1) \in \mathbb{N}$ tel que $\begin{cases} \exists (\alpha_n)_n \in \mathbb{R}^{N+1} \text{ tel que } \phi = \frac{1}{2}\sum_{k=0}^{N} \alpha_k \phi_{1,k} \\ \exists (\beta_n)_n \in \mathbb{R}^{N+1} \text{ tel que } \psi = \frac{1}{2}\sum_{k=0}^{N} \beta_k \phi_{1,k} \end{cases}$

$V_p \subset V_{p+1}$, on peut alors considérer le supplémentaire W_p de V_p dans V_{p+1} engendré par les $(\psi_{p,k})_{k\in\mathbb{Z}}$

Vient alors la décomposition : $V_p = W_{p-1} \oplus V_{p-1} = ... = W_{p-1} \oplus ... \oplus W_0 \oplus V_0$

1. La définition générale est un peu différente, mais ce cas particulier est suffisant pour notre étude
2. il existe $A, B \in \mathbb{R}_+^*$ tel que pour tout $(a_i) \in \ell^2(\mathbb{N})$ $A\sum_{i\in\mathbb{Z}} a_i^2 \leqslant \left\|\sum_{i\in\mathbb{Z}} a_i\phi_i\right\| \leqslant B\sum_{i\in\mathbb{Z}} a_i^2$

Soit $f \in V_p : f = \sum_{k\in\mathbb{Z}} x_k\phi_{p,k}$. Mais $f \in V_{p-1} \oplus W_{p-1} : f = \sum_{k\in\mathbb{Z}} a_k\phi_{p-1,k} + d_k\psi_{p-1,k}$

D'après les équations de raffinement : $\phi_{p-1,k} = \frac{1}{2}\sum_{j=0}^{N} \alpha_j\phi_{p,2k+j}$ et $\psi_{p-1,k} = \frac{1}{2}\sum_{j=0}^{N} \beta_j\phi_{p,2k+j}$

On peut donc facilement réaliser la décomposition des $(a_k), (d_k)$ en (x_k) :

$$x_{2k} = \frac{1}{2}\sum_{j=0}^{P} \alpha_{2j}a_{k-j} + \beta_{2j}d_{k-j} \text{ et } x_{2k+1} = \frac{1}{2}\sum_{j=0}^{P} \alpha_{2j+1}a_{k-j} + \beta_{2j+1}d_{k-j}$$

La transformation inverse n'est pas évidente a priori. On peut juste dire qu'il existe $(\tilde{\alpha}_{j,k}, \tilde{\beta}_{j,k})_{j\in[\![0,P]\!],k\in\mathbb{Z}}$ tel que :

$$a_k = \frac{1}{2}\sum_{j=0}^{N} \tilde{\alpha}_{j,k}x_{2k+j} \text{ et } d_k = \frac{1}{2}\sum_{j=0}^{N} \tilde{\beta}_{j,k}x_{2k+j}$$

Elle est analogue à une inversion de matrice (en dimension infinie)

4.2.2 Ondelettes orthogonales

Il existe plusieurs familles d'ondelettes. Les plus simples à manipuler sont les ondelettes orthogonales :

On a alors, pour tout $k, j \in \mathbb{Z}, \langle\phi_{p,k}|\phi_{p,j}\rangle = \delta_{k,j} = \langle\psi_{p,k}|\psi_{p,j}\rangle$ et $\langle\phi_{p,k}|\psi_{p,j}\rangle = 0$

C'est à dire que $W_p \perp V_p$ et que les $(\phi_{p,k})$ et les $(\psi_{p,k})$ sont des systèmes orthonormés.

Dans ce cas, la transformation est orthogonale, en raisonnant sur les produits scalaires (ou sur les matrices dans le cas fini), il vient alors : (voir annexe D)

$$\text{Pour tout } k : \tilde{\alpha}_{j,k} = \alpha_j, \text{ et } \tilde{\beta}_{j,k} = \beta_j$$

4.2.3 Ondelettes périodiques

Jusqu'à présent, on travaillait sur $L^2(\mathbb{R})$. Mais, comme nous cherchons à compresser des signaux bornés, il serait plus commode de se restreindre à $[0, 1]$.

Le plus simple est d'utiliser des ondelettes orthogonales périodiques.

On définit, pour $p, j \in \mathbb{Z} \times \mathbb{Z}, \chi_{p,j} : x \mapsto \sum_{n=-\infty}^{+\infty} \phi_{p,j}(x+n) = \sum_{n=-\infty}^{+\infty} \phi_{p,j+n2^p}(x)$ (bien défini car $\phi \in L^2(\mathbb{R})$)

On reste bien en présence d'un système orthogonal.

On se restreint alors à V_p de dimension 2^p . On peut donc réaliser toutes les opérations précédemment citées, mais avec un nombre fini d'opérations.

4.3 Sur une image

On réalise la décomposition en ondelettes alternativement sur les lignes et les colonnes de l'image, avec un nombre d'itérations fixé.

On récupère les valeurs, on les quantifie, et on réalise un codage RLE puis Huffman.

La transformation en ondelettes revient à un filtrage hautes et basses fréquences (les hautes fréquences étant sur les ondelettes, les basses sur les fonctions d'échelle).

Il y a beaucoup de 0 pour les hautes fréquences, on gagne en espace de façon similaire à la compression JPEG.

5 Conclusion

Des programmes ont été réalisés pour chaque type de compression d'image, et le résultat est montré en annexe D.

On constate que chaque type de compression amène des distorsions spécifiques. A un faible niveau de compression, pour chacun, ces altérations de l'image sont à peine visible. Elles apparaissent à des taux de compression plus élevés.

Pour un fichier très compressé, la compression par les ondelettes de Daubechies semble "rendre flous" les bords, quand les autres méthodes font apparaître des carreaux. Globalement, la compression JPEG semble meilleure.

En pratique, le format JPEG2000 utilise la compression par un type d'ondelettes non orthogonales, et atteint des taux de compression très élevés avec peu de pertes. Ce format-là est plus efficace que le format JPEG.

6 Bibliographie

Adrew B. Watson. « Image compression using the discrete Cosine Transform », *Mathematica Journal 4*, 1994, p81-88.

Irène Gannaz et Voichita Maxim. « Introduction aux ondelettes », présentation du 19 janvier 2010.

Anestis Antoniadis. « Compression et Débruitage avec les ondelettes », présentation du 21 mars 2003

Luc Claustres. « Introduction aux ondelettes », présentation, 2002-2003

Kévin Drapel. « Le format JPEG 2000 » (mise à jour le 10/09/2006), sur Wikibooks. http ://fr.wikibooks.org/wiki/Le_format_JPEG_2000

7 Annexes

7.1 Annexe A - Codage entropique

7.1.1 Exemple de codage RLE

Exemple de message compressé par l'algorithme RLE :

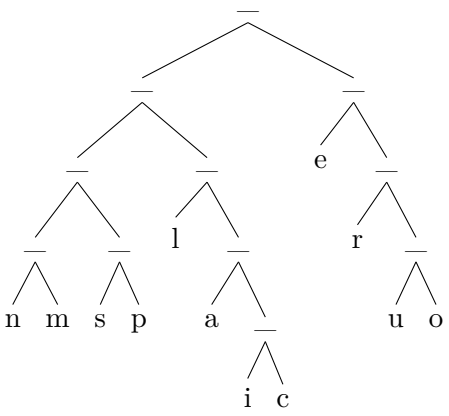
```
let msg = rle_compress [|5; 5; 1; 1; 2; 3; 4; 5; 5; 5; 5; 6; 7; 9; 9; 7; 8; 8; 8; 8; 8; 8; 9|];;

- : int list = [2; 5; 2; 1; 131; 2; 3; 4; 4; 5; 133; 6; 7; 9; 9; 7; 6; 8; 129; 9]

rle_decompress msg;;

- : int list = [5; 5; 1; 1; 2; 3; 4; 5; 5; 5; 5; 6; 7; 9; 9; 7; 8; 8; 8; 8; 8; 8; 9]
```

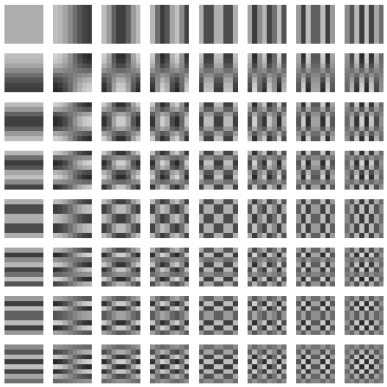
7.1.2 Algorithme de Huffman



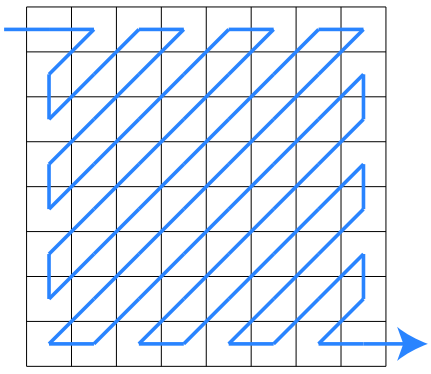
On a codé "ecolenormalesuperieure" selon l'algorithme de Huffman.
L'algorithme construit l'arbre ci-contre, il associe ensuite "1" pour une lecture d'arbre droit, "0" pour une lecture d'arbre gauche. "r" s'écrit alors "110", et "s", "0010".
Quand au mot entier, il s'écrit "10 01111 1111 010 10 0000 1111 110 0001 0110 010 10 0010 1110 0011 10 110 01110 10 1110 110 10", soit un gain en longueur de 17% par rapport au mot codé avec des lettres sur 4 bits.

7.2 Annexe B - Compression JPEG

Vecteurs de base de la DCT sur des images 8x8 pixels



Sens de lecture pour la linéarisation d'un carreau 8x8



7.3 Annexe C - Compression par ondelettes

7.3.1 Matrice de changement de base pour les ondelettes périodiques orthogonales

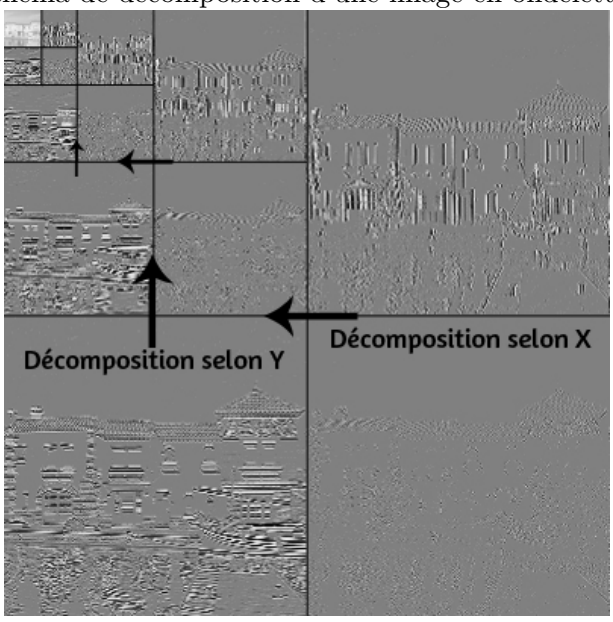
Avec les notations de la partie 4., dans le cas où $N = 3$, pour $p = 3$, on a la matrice de changement de base :

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \alpha_0 & \beta_0 & & & & & \alpha_2 & \beta_2 \\ \alpha_1 & \beta_1 & & & & & \alpha_3 & \beta_3 \\ \alpha_2 & \beta_2 & \alpha_0 & \beta_0 & & & & \\ \alpha_3 & \beta_3 & \alpha_1 & \beta_1 & & & & \\ & & \alpha_2 & \beta_2 & \alpha_0 & \beta_0 & & \\ & & \alpha_3 & \beta_3 & \alpha_1 & \beta_1 & & \\ & & & & \alpha_2 & \beta_2 & \alpha_0 & \beta_0 \\ & & & & \alpha_3 & \beta_3 & \alpha_1 & \beta_1 \end{pmatrix} \begin{pmatrix} a_0 \\ d_0 \\ a_1 \\ d_1 \\ a_2 \\ d_2 \\ a_3 \\ d_3 \end{pmatrix}$$

Comme la matrice est supposée orthogonale, on l'inverse facilement : on trouve alors les coefficients donnés en partie 4.

$$\begin{pmatrix} a_0 \\ d_0 \\ a_1 \\ d_1 \\ a_2 \\ d_2 \\ a_3 \\ d_3 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 & & & & \\ \beta_0 & \beta_1 & \beta_2 & \beta_3 & & & & \\ & & \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 & & \\ & & \beta_0 & \beta_1 & \beta_2 & \beta_3 & & \\ & & & & \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 \\ & & & & \beta_0 & \beta_1 & \beta_2 & \beta_3 \\ \alpha_2 & \alpha_3 & & & & & \alpha_0 & \alpha_1 \\ \beta_2 & \beta_3 & & & & & \beta_0 & \beta_1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

Schéma de décomposition d'une image en ondelettes



7.3.2 Ondelettes de Haar

Les ondelettes de Haar sont les plus simples :

– La fonction d'échelle est $\phi : x \mapsto \begin{cases} 1 \text{ si } x \in [0,1] \\ 0 \text{ sinon} \end{cases}$

– L'ondelette mère est $\psi : x \mapsto \begin{cases} 1 \text{ si } x \in [0,1/2[\\ -1 \text{ si } x \in [1/2,1] \\ 0 \text{ sinon} \end{cases}$

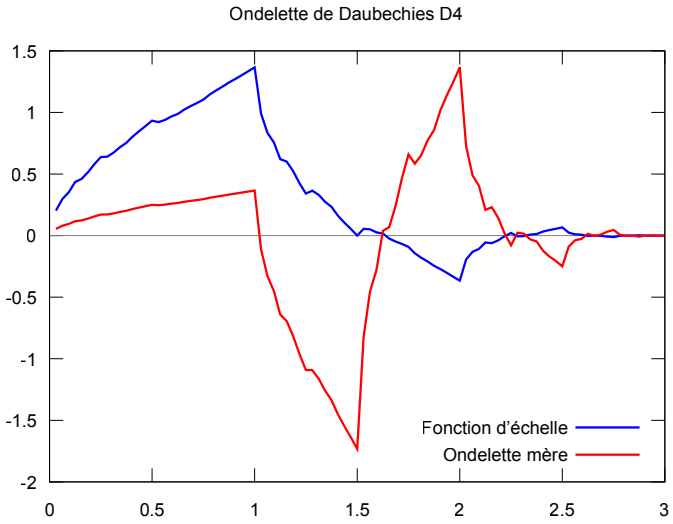
Les coefficients d'ondelettes sont alors simples : avec les notations déjà utilisées, on a :

$$\alpha_0 = \sqrt{2}, \alpha_1 = \sqrt{2}, \beta_0 = \sqrt{2}, \beta_1 = -\sqrt{2}$$

La décomposition sur les ondelettes de Haar revient à considérer la somme et la différence de 2 valeurs consécutives.

7.3.3 Ondelettes de Daubechies - D4

Les ondelettes de Daubechies sont une famille d'ondelettes orthogonales très utilisées. Il y en a 10, numérotées de 2 à 20 (pour des entiers pairs). L'ondelette D2 est celle de Haar.

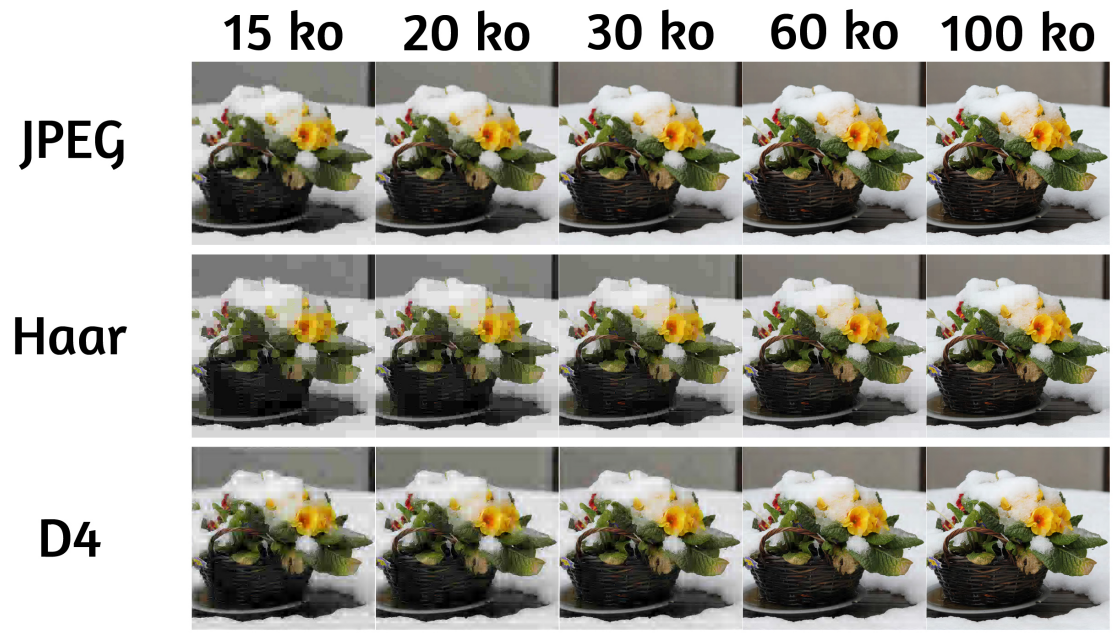


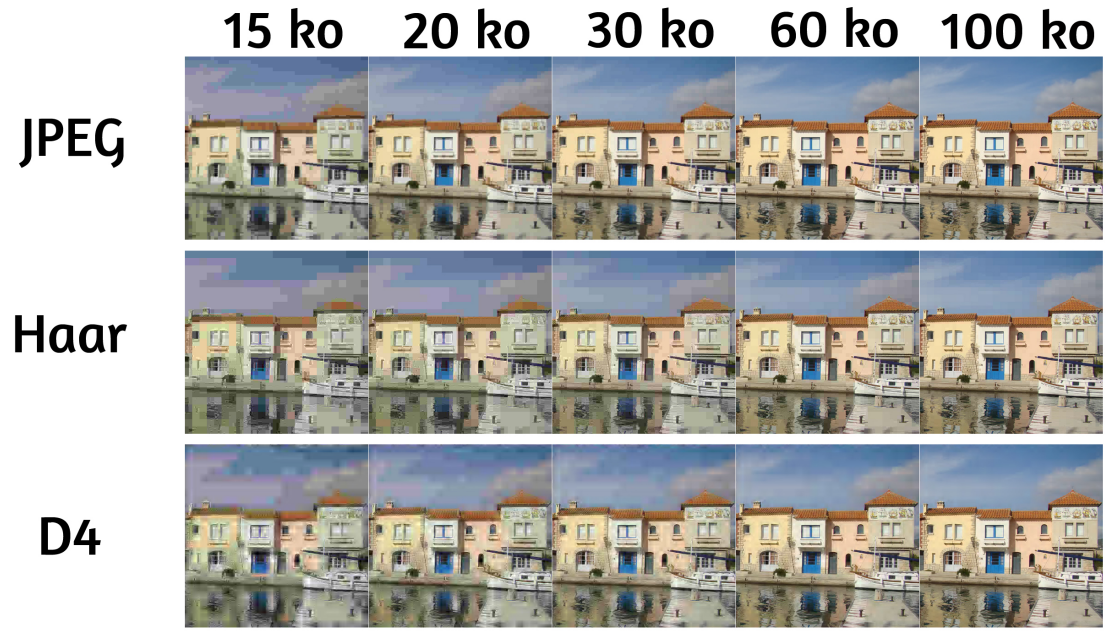
Pour la transformation discrète, on a les coefficients suivants :

α_0	0.6830127	β_0	-0.1830127
α_1	1.1830127	β_1	-0.3169873
α_2	0.3169873	β_2	1.1830127
α_3	-0.1830127	β_3	-0.6830127

On vérifie que la transformation est orthogonale.

7.4 Annexe D - Résultats et comparaison des méthodes





7.5 Annexe E - Extraits de programmes

Les programmes ont été réalisés en O-CAML.

7.5.1 Compression Huffman

```
type 'a pQueue = Head of int * 'a * 'a pQueue | Nil;;
let rec insert_pQueue priority valeur = function
    |Nil -> Head (priority, valeur, Nil)
    |Head (p,v,q) as a -> if p >= priority
        then Head (priority, valeur, a)
        else Head (p,v,insert_pQueue priority valeur q);;
let ( @:: ) (p,v) q = insert_pQueue p v q;;
type binTree=Node of binTree * binTree | Leaf of int;;
```

```
(* Prend une liste d'entiers, renvoie un couple (binTree * int list)
contenant l'arbre de Huffman et la liste des bits encodés *)
let huffman_compress intList =
    (* trouve les valeurs extrémales d'une liste *)
    let rec find_minmax mini maxi = function
        |[] -> mini,maxi
        |k::q -> if (k>maxi) then find_minmax mini k q
            else find_minmax (min mini k) maxi q
    in
    let h,t=List.hd intList, List.tl intList in
    let min,max=find_minmax h h t in
    let stats=Array.make (max-min+1) 0 in
    (* compte le nombre d'occurence de chaque valeur *)
    let rec remplirStats = function
        |[] -> ()
        |k::q -> stats.(k-min)<-stats.(k-min)+1; remplirStats q
    in
    remplirStats intList;
    (* initialise la file de priorité *)
    let rec initPrior liste tab = function
        |(-1) -> liste
        |k -> if(tab.(k)<>0)
                then initPrior ( (tab.(k), Leaf (k+min)) @:: liste) tab (k-1)
                else initPrior liste tab (k-1)
    in
    let maxis = initPrior Nil stats (Array.length stats-1) in
    (* construit l'arbre de Huffman *)
    let rec buildHuffmanTree = function
        |Nil -> failwith "Erreur"
        |Head (p,v,Nil) -> v
        |Head (t,v,Head(q,w,r)) -> buildHuffmanTree ((t+q,Node (v,w)) @:: r)
    in
    let asso = Array.make (max-min+1) [] in
    let rec buildAssoc l = function
        |Leaf k -> asso.(k-min)<-l
        |Node (g,d) -> buildAssoc (0::l) g;
              buildAssoc (1::l) d
    (* note : les codes binaires sont à l'envers *)
    in
    let tree=buildHuffmanTree maxis in
    buildAssoc [] tree;
    let rec encode accu = function
        |[] -> accu
        |k::q -> encode (asso.(k-min) @ accu) q
    in
    tree, (List.rev (encode [] intList));;

(* Décompression : prend (binTree * int list) et renvoie une liste d'entiers *)
let huffman_decompress (tree,code) =
    let rec makeval li tr = match(li,tr) with
        |_,Leaf k -> k,li
        |0::q,Node (g,d) -> makeval q g
        |1::q,Node (g,d) -> makeval q d
        |_,_ -> failwith "Erreur : code non terminé"
    in
    let rec buildList = function
        |[] -> []
        |l -> let v,reste = makeval l tree in
             v::(buildList reste)
    in buildList code ;;
```

7.5.2 Transformée cosinus discrète

```
(* Prend la base de destination, l'image à compresser, et ses tailles, et applique
    la DCT sur l'image *)
let dct base signal width height =
    let temptab=Array.make (Array.length signal) 0. in
    let outtab=Array.make (Array.length signal) 0. in
    let len=width*height/8 in   (* a priori : images découpables en carreaux 8x8 *)
```

```
    let coldct () =     (* DCT des colonnes *)
        let rec crossProduct start tab = function
            |8 -> 0.
            |k -> tab.(k) *. intab.(start+k) +. (crossProduct start tab (k+1))
            (* produit scalaire des 8 colonnes en partant de start avec le vecteur
                de base tab *)
        in
        let rec dct8 start =
            for k=0 to 7 do
                temptab.(start+k) <- crossProduct start base.(k) 0
            done
        in
        for k=0 to (len-1) do
            dct8 (k*8)
            (* on calcule la DCT sur les colonnes (== cases consécutives) *)
        done;
    in
    let rowdct () =
        let rec crossProduct start tab = function
            |8 -> 0.
            |k -> tab.(k) *. temptab.(start+k*width)
                    +. (crossProduct start tab (k+1))
        in
        let dct8 start =
            for k=0 to 7 do
                outtab.(start+k*width) <- crossProduct start base.(k) 0
            done;
        in
        for j=0 to (height/8-1) do
            let yy=j*8*width in
            for k=0 to (width-1) do
                dct8 (yy + k)
            done;
        done
    in
    coldct ();
    rowdct ();
    outtab;;
```

7.5.3 Décomposition sur la base des ondelettes de Daubechies

```
(* Prend une précision, un nombre d'itérations, une image et sa taille et compresse
    l'image par les ondelettes de Daubechies *)
let h04d=0.6830127 and h14d=1.1830127 and h24d=0.3169873 and h34d=-.0.1830127;;
let g04d=h34d and g14d=-.h24d and g24d=h14d and g34d=-.h04d;;

let dwt_D4_2d precision iterations signal width height =
    let n=Array.length signal in
    let t1=Array.init n (fun x->signal.(x)) in  (* on copie le signal d'entrée *)
    let t2=Array.make n 0. in
    let rec compress sizeX sizeY input output = function
        |0 -> input
        |k -> let nX=sizeX/2 in
              for j=0 to (sizeY-1) do   (* colonnes *)
                let dy=j*width in
                output.(dy+nX-1)<-0.5*.(input.(dy+sizeX-2)*.h04d
                        +. input.(dy+sizeX-1)*.h14d +. input.(dy)*.h24d
                        +. input.(dy+1)*.h34d);
                output.(dy+sizeX-1)<-0.5*.(input.(dy+sizeX-2)*.g04d
                        +. input.(dy+sizeX-1)*.g14d +. input.(dy)*.g24d
                        +. input.(dy+1)*.g34d);
                for i=0 to (nX-2) do
                    let t1=input.(dy+2*i) and t2 = input.(dy+2*i+1)
                        and t3=input.(dy+(2*i+2)) and t4=input.(dy+(2*i+3)) in
                    output.(dy+i)<-0.5*.(t1*.h04d +. t2*.h14d
                            +. t3*.h24d +. t4*.h34d);
                    output.(dy+nX+i)<-0.5*.(t1*.g04d +. t2*.g14d
                            +. t3*.g24d  +. t4*.g34d);
                done;
              done;
```

```
            let nY=sizeY/2 in
            let dn=nY*width in
            for i=0 to (sizeX-1) do
              input.(i+(nY-1)*width)<-0.5*.(output.(i+(sizeY-2)*width)*.h04d
                      +. output.(i+(sizeY-1)*width)*.h14d +. output.(i)*.h24d
                      +. output.(i+width)*.h34d);
              input.(i+(sizeY-1)*width)<-0.5*.(output.(i+(sizeY-2)*width)*.g04d
                      +. output.(i+(sizeY-1)*width)*.g14d +. output.(i)*.g24d
                      +. output.(i+width)*.g34d);
              for j=0 to (nY-2) do
                  let dj=j*width in
                  let t1=output.(2*dj+i) and t2=output.(2*dj+width+i)
                      and t3=output.((2*j+2)*width+i) and t4=output.((2*j+3)*
                          width+i)in
                  input.(dj+i)<-0.5*.(t1*.h04d +. t2*.h14d
                          +. t3*.h24d +. t4*.h34d);
                  input.(dn+dj+i)<-0.5*.(t1*.g04d +. t2*.g14d
                          +. t3*.g24d  +. t4*.g34d);
              done;
            done;
            compress nX nY input output (k-1)
in
quantify precision (compress width height t1 t2 iterations);;
(* quantify divise toutes les valeurs par le paramètre précision, et renvoie
    des entiers *)
```