2. Le JPEG pour les petits : mode séquentiel

Le JPEG (*Joint Photographic Experts Group*) est un comité de standardisation pour la compression d'image dont le nom a été détourné pour désigner une norme en particulier, la norme JPEG, que l'on devrait en fait appeler `ISO/IEC IS 10918-1 | ITU-T Recommendation T.81`. [1]

Cette norme spécifie plusieurs alternatives pour la compression des images en imposant des contraintes uniquement sur les algorithmes et les formats du décodage. Notez que c'est très souvent le cas pour le codage source (ou compression en langage courant), car les choix pris lors de l'encodage garantissent la qualité de la compression. La norme laisse donc la réalisation de l'encodage libre d'évoluer. Pour une image, la qualité de compression est évaluée par la réduction obtenue sur la taille de l'image, mais également par son impact sur la perception qu'en a l'œil humain. Par exemple, l'œil est plus sensible aux changements de luminosité qu'aux changements de couleur. On préférera donc compresser les changements de couleur que les changements de luminosité, même si cette dernière pourrait permettre de gagner encore plus en taille. C'est l'une des propriétés exploitées par la norme JPEG.

Parmi les choix proposés par la norme, on trouve des algorithmes de compression avec ou sans perte (une compression avec pertes signifie que l'image décompressée n'est pas strictement identique à l'image d'origine) et différentes options d'affichage (séquentiel, l'image s'affiche en une passe pixel par pixel, ou progressif, l'image s'affiche en plusieurs passes en incrustant progressivement les détails, ce qui permet d'avoir rapidement un aperçu, quitte à attendre pour avoir l'image entière).

Dans son ensemble, il s'agit d'une norme plutôt complexe qui doit sa démocratisation à un format d'échange, le JFIF (JPEG File Interchange Format). En ne proposant au départ que le minimum essentiel pour le support de la norme, ce format s'est rapidement imposé, notamment sur Internet, amenant à la norme le succès qu'on lui connaît aujourd'hui. D'ailleurs, le format d'échange JFIF est également confondu avec la norme JPEG. Ainsi, un fichier possédant une extension `.jpg` ou `.jpeg` est en fait un fichier au format JFIF respectant la norme JPEG. Évidement, il existe d'autres formats d'échange supportant la norme JPEG comme les formats TIFF ou EXIF. La norme de compression JPEG peut aussi être utilisée pour encoder de la vidéo, dans un format appelé Motion-JPEG. Dans ce format, les images sont toutes enregistrées à la suite dans un flux. Cette stratégie permet d'éviter certains artefacts liés à la compression inter-images dans des formats types MPEG.

Cette section décrit le mode JPEG le plus simple et le plus répandu, nommé *baseline sequential* (compression séquentielle, avec pertes, codage par DCT et Huffman, précision 8 bits), dans un fichier au format JFIF. Le chapitre 3 abordera le mode *progressive* dont le

fonctionnement est très similaire mais en plus dur, juste, pour vous donner mal à la tête et faire plaisir à vos profs.

2.1 Principe général du codec JPEG *baseline sequential*

Cette section détaille les étapes successives mises en œuvre lors de l'encodage, c'est-à-dire la conversion d'une image au format PPM vers une image au format JPEG. En effet, bien que le projet s'attaque au décodage, le principe s'explique plus facilement du point de vue du codage.

Tout d'abord, l'image est partitionnée en macroblocs ou MCU pour *Minimum Coded Unit*. La plupart du temps, les MCUs sont de taille 8x8, 16x8, 8x16 ou 16x16 pixels selon le facteur d'échantillonnage (voir section sous-échantillonnage). Chaque MCU est ensuite réorganisée en un ou plusieurs blocs de taille 8x8 pixels.

La suite porte sur la compression d'un bloc 8x8. Chaque bloc est traduit dans le domaine fréquentiel par transformation en cosinus discrète (DCT). Le résultat de ce traitement, appelé bloc fréquentiel, est encore un bloc 8x8 mais dont les coordonnées ne sont plus des pixels, c'est-à-dire des positions du domaine spatial, mais des amplitudes à des fréquences données. On y distingue un coefficient continu *DC* aux coordonnées (0,0) et 63 coefficients fréquentiels *AC*. [2] Les plus hautes fréquences se situent autour de la case (7,7).

L'œil étant moins sensible aux hautes fréquences, il est plus facile de les filtrer avec cette représentation fréquentielle. Cette étape de filtrage, dite de quantification, détruit de l'information pour permettre d'améliorer la compression, au détriment de la qualité de l'image (d'où l'importance du choix du filtrage). Elle est réalisée bloc par bloc à l'aide d'un filtre de quantification, qui peut être générique ou spécifique à chaque image. Le bloc fréquentiel filtré est ensuite parcouru en zig-zag (ZZ) afin de transformer le bloc en un vecteur de 64x1 fréquences avec les hautes fréquences en fin. De la sorte, on obtient statistiquement plus de 0 en fin de vecteur.

Ce bloc vectorisé est alors compressé en utilisant successivement plusieurs codages sans perte : d'abord un codage RLE pour exploiter les répétitions de 0, un codage des différences plutôt que des valeurs, puis un codage entropique [3] dit de *Huffman* qui utilise un dictionnaire spécifique à l'image en cours de traitement.

Les étapes ci-dessus sont appliquées à tous les blocs composant les MCUs de l'image. La concaténation de ces vecteurs compressés forme un flux de bits (*bitstream*) qui est stocké dans le fichier JPEG.

Un décodeur effectue lui les mêmes opérations, mais dans l'ordre inverse. C'est logique.

Les opérations de codage/décodage sont résumées sur la figure ci-dessous:

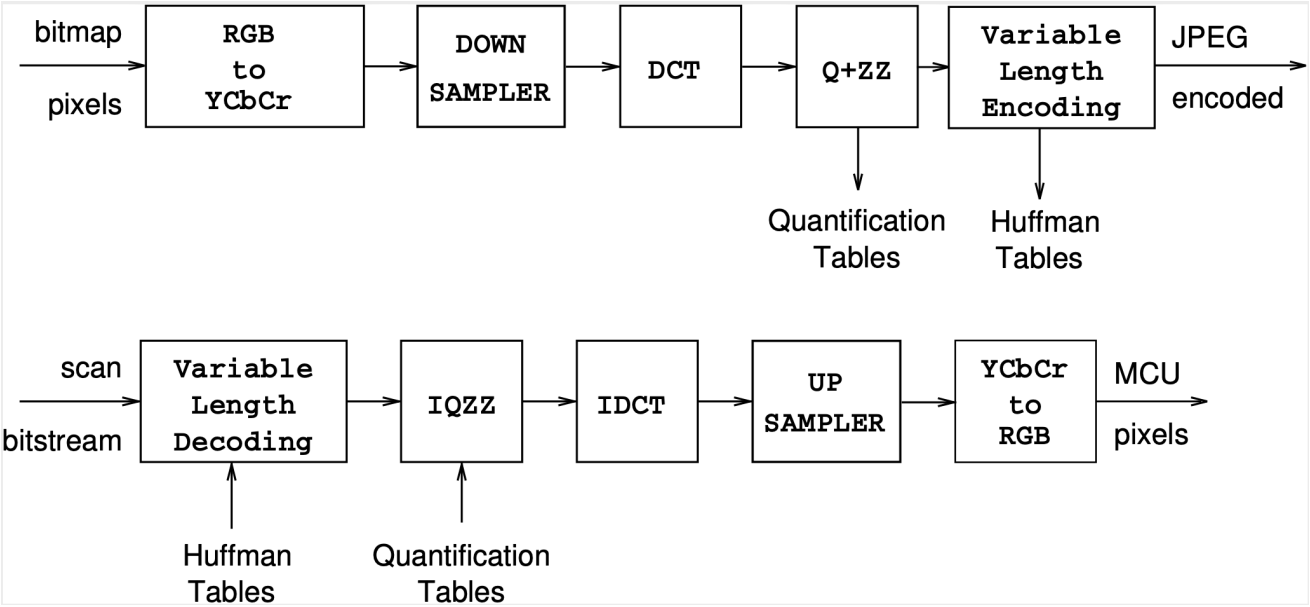


Figure 2.1 : Principe du codec JPEG: opérations de codage (en haut) et de décodage (en bas).

Pour pouvoir décoder une image, plusieurs informations sont nécessaires: les tables de Huffman et de quantification qui apparaissent sur la figure, mais aussi d'autres informations générales comme la taille de l'image, le nombre de composantes (monochrome ou couleur), etc. Ces informations sont regroupées dans un entête qui, comme son nom l'indique précède les données encodant l'image.

TODO

Votre décodeur devra donc:

1. lire **l'entête** se trouvant au début du fichier, pour retrouver les **paramètres** et **tables** utiles au décodage
2. puis lire le **flux de données donneés d'encodage** en le décodant au furet à mesure (!) en utilisant les informations lues dans l'entête

Rien que pour vous:

- La suite de cette page détaille les opérations de codage/décodage (dans le sens du décodage).
- l'annexe A décrit le format d'un entête (marqueurs et sections)
- l'annexe B fournit un exemple d'entête de fichier JPEG
- l'annexe C fournit un exemple d'encodage d'une MCU

2.2 Représentation des données

Il existe plusieurs manières de représenter une image. Une image numérique est en fait un tableau de pixels, chaque pixel ayant une couleur distincte. Dans le domaine spatial, le codec utilise deux types de représentation de l'image.

Le format RGB, le plus courant, est le format utilisé en entrée. Il représente chaque couleur de pixel en donnant la proportion de trois couleurs primitives: le rouge (R), le vert (G), et le bleu (B). Une information de transparence *alpha* (A) peut également être fournie (on parle alors de ARGB), mais elle ne sera pas utilisée dans ce projet. Le format RGB est le format utilisé en amont et en aval du décodeur.

Un deuxième format, appelé YCbCr, utilise une autre stratégie de représentation, en trois composantes: une luminance dite Y, une différence de chrominance bleue dite Cb, et une différence de chrominance rouge dite Cr. Le format YCbCr est le format utilisé en interne par la norme JPEG. Une confusion est souvent réalisée entre le format YCbCr et le format YUV.[4]

La stratégie de représentation YCbCr est plus efficace que le RGB (*Red*, *Green*, *Blue*) classique, car d'une part les différences sont codées sur moins de bits que les valeurs et d'autre part elle permet des approximations (ou de la perte) sur la chrominance à laquelle l'œil humain est moins sensible.

2.3 Décodage : quantification inverse et zig-zag inverse

2.3.1 Quantification inverse

Au codage, la quantification consiste à diviser (terme à terme) chaque bloc 8x8 par une matrice de quantification, elle aussi de taille 8x8. Les résultats sont arrondis, de sorte que plusieurs coefficients initialement différents ont la même valeur après quantification. De plus de nombreux coefficients sont ramenés à 0, essentiellement dans les hautes fréquences auxquelles l'oeil humain est peu sensible.

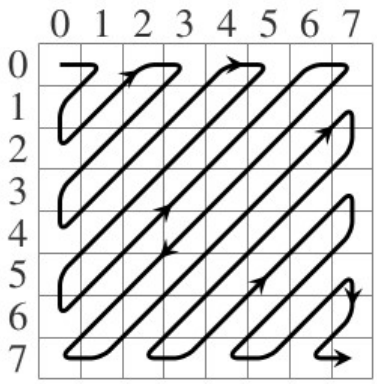
Deux tables de quantification sont généralement utilisées, une pour la luminance et une pour les deux chrominances. Le choix de ces tables, complexe mais fondamental quant à la qualité de la compression et au taux de perte d'information, n'est pas discuté ici. Les tables utilisées à l'encodage sont inclues dans le fichier JFIF. Avec une précision de 8 bits (le cas dans ce projet, même s'il est possible dans la norme d'avoir une précision sur 12 bits), les coefficients sont des entiers non signés entre 0 et 255.

La quantification est l'étape du codage qui introduit le plus de perte, mais aussi une de celles qui permet de gagner le plus de place (en réduisant l'amplitude des valeurs à encoder et en annulant de nombreux coefficients dans les blocs).

Au décodage, la quantification inverse consiste à multiplier élément par élément le bloc fréquentiel par la table de quantification. Moyennant la perte d'information liée aux arrondis, les blocs fréquentiels initiaux seront ainsi reconstruits.

2.3.2 Zig-zag inverse

L'opération de réorganisation en "zig-zag" permet de représenter un bloc 8x8 sous forme de vecteur 1D de 64 coefficients. Surtout, l'ordre de parcours présenté sur la figure ci-dessous place les coefficients des hautes fréquences en fin de vecteur. Comme ce sont ceux qui ont la plus forte probabilité d'être nuls suite à la quantification (cf sous-section précédente), ceci permet d'optimiser la compression RLE décrite dans la section 2.8.3.



0	1	2	. . .	62	63

Figure 2.2 : Réordonnancement Zig-zag pour représenter un bloc 8 × 8 par un vecteur 1 × 64. Le coefficient continu DC en (0, 0) va à l'indice 0. Les coefficients fréquentiels AC en (0, 1), (1, 0), ... , (7, 7) se retrouvent aux indices 1, 2, .., 63.

Au décodage, le vecteur 1x64 obtenu après lecture dans le flux JPEG et décompression doit être réorganisé par l'opération zig-zag inverse qui recopie les 64 coefficients en entrée aux coordonnées fournies par le zig-zag de la figure 2.2. Ceci permet d'obtenir à nouveau un bloc 8x8.

2.3.3 Ordre des opérations

Au codage, la quantification a lieu avant la réorganisation zig-zag. Par contre la matrice de quantification stockée dans le fichier JPEG est elle aussi réorganisée au format zig-zag ! Au décodage, l'ordre des opérations est donc inversé : il faut d'abord multiplier les coefficients du bloc par ceux de la matrice de quantification lue, puis effectuer la réorganisation zig-zag inverse.

2.4 Décodage : transformée en cosinus discrète inverse (iDCT)

A l'encodage, on applique au vecteur 1x64 une transformée en cosinus discrète (DCT) qui convertit les informations spatiales en informations fréquentielles. Au décodage, l'étape inverse consiste à retransformer les informations fréquentielles en informations spatiales. C'est une formule mathématique « classique » de transformée. Dans sa généralité, la formule de la transformée en cosinus discrète inverse (iDCT, pour *Inverse Discrete Cosinus Transform*) pour les blocs de taille nxn pixels est :

$$S(x,y) = \frac{1}{\sqrt{2n}} \sum_{\lambda=0}^{n-1} \sum_{\mu=0}^{n-1} C(\lambda)C(\mu) \cos\left(\frac{(2x+1)\lambda\pi}{2n}\right) \cos\left(\frac{(2y+1)\mu\pi}{2n}\right) \Phi(\lambda,\mu).$$

Dans cette formule, S est le bloc spatial et Φ le bloc fréquentiel. Les variables x et y sont les coordonnées des pixels dans le domaine spatial et les variables λ et μ sont les coordonnées des fréquences dans le domaine fréquentiel. Finalement, le coefficient C est tel que :

$$C(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } \xi = 0, 1 \\ \text{sinon.} \end{cases}$$

Dans le cas qui nous concerne, $n = 8$ bien évidemment.

A l'issue de cette opération :

- un offset de 128 est ajouté à chaque $S(x,y)$; [5]
- chaque valeur est « saturée », c'est-à-dire fixée à 0 si elle est négative, et à 255 si elle est supérieure à 255 ;
- finalement les valeurs sont converties en entier non-signé sur 8 bits. Attention, le calcul de l'iDCT se fait bien en flottants. Ce n'est qu'à la fin qu'on effectue une conversion des valeurs en entier 8 bits non-signés.

Vous remarquez rapidement que la verson naïve de l'iDCT selon la formule ci-dessus n'est, comment dire, pas très efficace. Le moment venu, allez voir en section 4.4.1.

2.5 Décodage : reconstitution des MCUs

Dans le processus de compression, le JPEG peut exploiter la faible sensibilité de l'œil humain aux composantes de chrominance pour réaliser un sous-échantillonnage (*downsampling*) de l'image.

Le sous-échantillonnage est une technique de compression qui consiste en une diminution du nombre de valeurs, appelées échantillons, pour certaines composantes de l'image. Pour prendre un exemple, imaginons qu'on travaille sur une image couleur YCbCr partitionnée en MCUs de 2x2 blocs de 8x8 pixels chacun, soit des MCUs 16x16 de 256 pixels au total.

Ces 256 pixels ayant chacun un échantillon pour chaque composante, leur stockage nécessiterait donc 256x3 = 768 échantillons. **On ne sous-échantillonne jamais la composante de luminance de l'image**. En effet, l'œil humain est extrêmement sensible à cette information, et une modification impacterait trop la qualité perçue de l'image. Cependant, comme on l'a dit, la chrominance contient moins d'information. On pourrait donc décider que pour 2 pixels de l'image consécutifs horizontalement, un seul échantillon par composante de chrominance suffit. Il faudrait seulement alors 256 + 128 + 128 = 512 échantillons pour représenter toutes les composantes, ce qui réduit notablement la place occupée! Si on

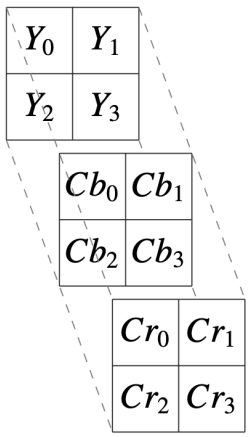
applique le même raisonnement sur les pixels de l'image consécutifs verticalement, on se retrouve à associer à 4 pixels un seul échantillon par chrominance, et on tombe à une occupation mémoire de 256 + 64 + 64 = 384 échantillons.

2.5.1 Au codage: sous-échantillonnage de l'image

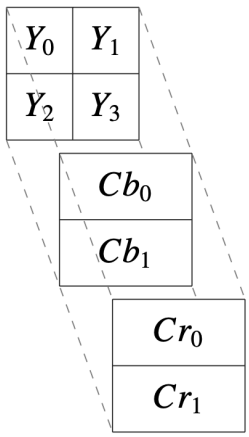
Dans ce document, nous utiliserons une notation directement en lien avec les valeurs présentes dans les sections JPEG de l'en-tête, décrites en annexe B, qui déterminent le facteurs d'échantillonnages (*sampling factors*). Ces valeurs sont identiques partout dans une image. En pratique, on utilisera la notation \(h\times v)\ de la forme :

$$h_1 \times v_1, h_2 \times v_2, h_3 \times v_3$$

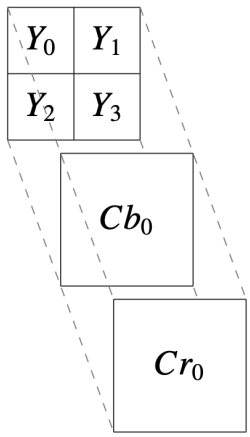
où h_i et v_i représentent le nombre de blocs horizontaux et verticaux pour la composante i. Comme Y n'est jamais compressé, **le facteur d'échantillonnage de Y donne les dimensions de la MCU en nombre de blocs**. Les sous-échantillonnages les plus courants sont décrits ci-dessous. Votre encodeur devra supporter au moins ces trois combinaisons, mais nous vous encourageons à gérer tous les cas !



(a) Sans sous-échantillonnage



(b) Sous-échantillonnage horizontal Cb et Cr



(c) Sous-échantillonnage horizontal et vertical Cb et Cr

La figure ci-dessus illustre les composantes Y, Cb et Cr avec et sans sous-échantillonnage, pour une MCU de 2 × 2 blocs.

- **Pas de sous-échantillonnage :** Le nombre de blocs est identique pour toutes les composantes. On se retrouve avec des facteurs de la forme $h_1 \times v_1, h_1 \times v_1, h_1 \times v_1$ (le même nombre de blocs répété pour toutes les composantes). La plupart du temps, on travaille dans ce cas sur des MCU de taille 1x1 ($h_1 = v_1 = 1$), mais on pourrait très bien trouver des images sans sous-échantillonnage découpées en MCUs de tailles différentes, comme par exemple 2x2 ou 1x2.[6] Sans sous-échantillonnage, la qualité de l'image est optimale mais le taux de compression est le plus faible.
- **Sous-échantillonnage horizontal :** La composante sous-échantillonnée comprend deux fois moins de blocs en horizontal que la composante Y. Le nombre de blocs en vertical reste le même pour les trois composantes. Par exemple, les facteurs d'échantillonnage 2x2, 1x2, 1x2 et 2x1, 1x1, 1x1 représentent tous deux une compression horizontale de

Cb et Cr, mais avec des tailles de MCU différentes : 2x2 blocs dans le premier cas, 2x1 blocs dans le second. Comme la moitié de la résolution horizontale de la chrominance est éliminée pour Cb et Cr (figure *(b)* ci-dessus), un seul échantillon par chrominance Cb et Cr est utilisé pour deux pixels voisins d'une même ligne. Cet échantillon est calculé sur la valeur RGB moyenne des deux pixels. La résolution complète est conservée verticalement. C'est un format très classique sur le web et les caméras numériques, qui peut aussi se décliner en vertical en suivant la même méthode.

- **Sous-échantillonnage horizontal et vertical :** La composante sous-échantillonnée comprend deux fois moins de blocs en horizontal et en vertical que la composante Y. La figure *(c)* illustre cette compression pour les composantes Cb et Cr, avec les facteurs d'échantillonnage 2x2, 1x1, 1x1. Comme la moitié de la résolution horizontale et verticale de la chrominance est éliminée pour Cb et Cr, un seul échantillon de chrominance Cb et Cr est utilisé pour quatre pixels. La qualité est visiblement moins bonne, mais sur un minitel ou un timbre poste, c'est bien suffisant!

Dans la littérature, on caractérise souvent le sous-échantillonnage par une notation de type **L:H:V**. Ces trois valeurs ont une signification qui permet de connaître le facteur d'échantillonnage. [7] Les notations suivantes font référence aux sous-échantillonnages les plus fréquemment utilisés :

- **4:4:4** : Pas de sous-échantillonnage ;
- **4:2:2** : Sous-échantillonnage horizontal (ou vertical) des composantes Cb et Cr ;
- **4:2:0** : Sous-échantillonnage horizontal et vertical des composantes Cb et Cr.

Restrictions sur les valeurs de h et de v

La norme fixe un certain nombre de restrictions sur les valeurs que peuvent prendre les facteurs d'échantillonnage. En particulier :

- La valeur de chaque facteur *h* ou *v* doit être comprise entre 1 et 4 ;
- La somme des produits $h_i \times v_i$ doit être inférieure ou égale à 10 ;
- Les facteurs d'échantillonnage des chrominances doivent diviser parfaitement ceux de la luminance.

Par exemple, les facteurs d'échantillonnage 2x1, 1x2, 1x1 ne sont pas corrects, puisque le facteur d'échantillonnage vertical de la chrominance bleue ne divise pas parfaitement celui de la luminance (2 ne divise pas 1).

2.5.2 Au décodage : sur-échantillonnage

Au niveau du décodeur, il faut à l'inverse sur-échantillonner (on parle d'*upsampling*) les blocs Cb et Cr pour qu'ils recouvrent la MCU en totalité. Ceci signifie par exemple que dans le cas d'un sous-échantillonnage horizontal, la moitié gauche de chaque bloc de chrominance couvre le premier bloc Y_0, alors que la moitié droite couvre le second bloc Y_1.

2.5.3 Ordre de lecture des blocs dans le flux JPEG

A l'encodage, l'image est découpée en MCUs, par balaiement de gauche à droite puis de haut en bas. La taille des MCUs est donnée par les facteurs d'échantillonnage de la composante Y. L'ordonnancement des blocs dans le flux **suit toujours la même séquence**. La plupart du temps, l'ordre d'apparition des blocs est le suivant : ceux de la composante Y arrivent en premier, suivis de ceux de la composante Cb et enfin de la composante Cr. Mais on peut trouver des images dont l'ordre d'apparition des composantes est différente. Dans tous les cas, cet ordre est indiqué dans les en-têtes de sections `SOF` et `SOS` décrites en annexe B. L'ordre d'apparition des blocs d'une MCU pour une composante donnée dépend du nombre de composantes traitées.

Dans le cas d'une image couleur (3 composantes, Y, Cb, Cr), les blocs sont ordonnés de gauche à droite et de haut en bas. Prenons l'exemple d'une image de taille 16x16 pixels, composée de deux MCUs de 2x1 blocs et dont les composantes Cb et Cr sont sous-échantillonnées horizontalement. Dans le flux, on verra d'abord apparaitre les blocs des trois composantes de la première MCU ordonnés comme ci-dessus, à savoir $Y_0^0 Y_1^0 Cb^0 Cr^0$, puis ceux de la deuxième MCU ordonnés de la même façon, soit $Y_0^1 Y_1^1 Cb^1 Cr^1$. La séquence complète lue dans le flux sera donc $Y_0^0 Y_1^0 Cb^0 Cr^0 Y_0^1 Y_1^1 Cb^1 Cr^1$. La figure 2.4 tirée de la norme JPEG illustre cet ordonnancement.

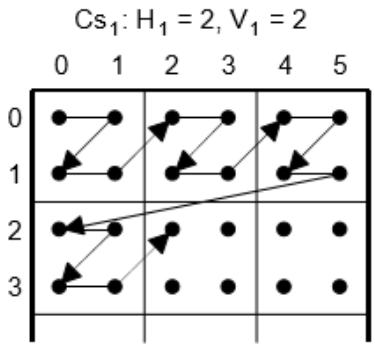


Figure 2.4 : Ordonnancement de blocs à 3 composantes de couleur

Cette figure illustre l'ordre d'apparition des blocs dans le bitstream, dans le cas d'une image couleur composée de MCUs de 2x2 blocs. Chaque point représente un bloc 8x8. Les barres verticales et horizontales délimitent les MCUs. La flèche indique l'ordre dans lequel les blocs sont écrits dans le flux.

Dans le cas d'une image niveaux de gris (une seule composante, Y) les blocs sont ordonnés dans le flux de la même façon que les pixels qu'ils représentent, c'est-à-dire séquentiellement de gauche à droite et de haut en bas pour toute l'image, et ce quelles que soient les dimensions de la MCU considérée, comme représenté sur la figure ci-dessous :

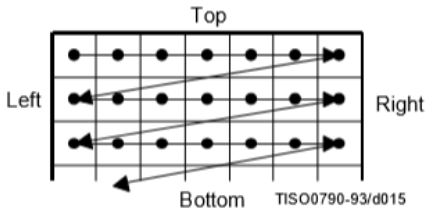


Figure 2.5 : Ordonnancement de blocs en niveaux de gris

Comme pour la figure 2.4, chaque point représente un bloc 8x8. La flèche indique encore une fois l'ordre dans lequel les blocs sont écrits dans le bitstream : on ne tient pas compte ici du découpage de l'image en MCUs pour ordonnancer les blocs.

2.6 Décodage : conversion vers des pixels RGB

La conversion YCbCr vers RGB s'effectue pixel par pixel à partir des trois composantes de la MCU reconstituée (éventuellement issues d'un sur-échantillonnage pour les composantes de chrominance). Ainsi, pour chaque pixel dans l'espace YCbCr, on effectue le calcul suivant (donné dans la norme de JPEG) pour obtenir le pixel RGB :

$$R = Y - 0,0009267 \times (C_b - 128) + 1,4016868 \times (C_r - 128)$$

$$G = Y - 0,3436954 \times (C_b - 128) - 0,7141690 \times (C_r - 128)$$

$$B = Y + 1,7721604 \times (C_b - 128) + 0,0009902 \times (C_r - 128)$$

Les formules simplifiées suivantes sont encore acceptables pour respecter les contraintes de rapport signal sur bruit du standard : [8]

$$R = Y + 1,402 \times (Cr - 128)$$

$$G = Y - 0,34414 \times (Cb - 128) - 0,71414 \times (Cr - 128)$$

$$B = Y + 1,772 \times (Cb - 128)$$

Ces calculs incluent des opérations arithmétiques sur des valeurs flottantes et signées, et dont les résultats sont potentiellement hors de l'intervalle [0 : : : 255]. En sortie, les valeurs de R, G et B devront être entières et saturées entre 0 et 255 (comme suite à la DCT inverse).

2.7 Décodage : des MCUs à l'image

La dernière étape est de reconstituer l'image à partir de la suite des MCUs décodées. La taille de l'image n'étant par forcément un multiple de la taille des MCUs, le découpage en MCUs peut « déborder » à droite et en bas (figure 2.6). A l'encodage, la norme recommande de compléter les MCUs en dupliquant la dernière colonne (respectivement ligne) contenue dans

l'image dans les colonnes (respectivement lignes) en trop. Au décodage, les MCUs sont reconstruites normalement, il suffit de tronquer celles à droite et en bas de l'image selon le nombre exact de pixels.

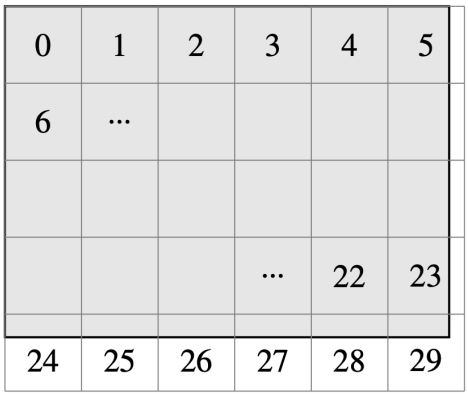


Figure 2.6 : Découpage en MCUs

La figure 2.6 donne un exemple d'une image 46 × 35 (fond gris) à compresser sans sous-échantillonnage, soit avec des MCUs composées d'un seul bloc 8 × 8. Pour couvrir l'image en entier, 6 × 5 MCUs sont nécessaires. Les MCUs les plus à droite et en bas (les 6e, 12e, 18e, 24e, puis 25e à 30e) devront être tronquées lors de la décompression.

2.8 (Dé)compression des données des blocs fréquentiels

Bien, voyons maintenant comment sont codés/décodés les blocs de base 8x8 (représentés en vecteurs de 64 coefficients).

Les blocs fréquentiels sont compressés sans perte dans le *bitstream* JPEG par l'utilisation de plusieurs techniques successives. Tout d'abord, les répétitions de 0 sont exploitées par un codage de type *RLE* (voir 2.8.3); puis les valeurs non nulles sont codées comme *différence* par rapport aux valeurs précédentes (voir 2.8.2); enfin, les symboles obtenus par l'application des deux codages précédents sont codés par un codage entropique de Huffman (2.8.1).

Nous présentons dans cette section les trois codages, en commençant par détailler le codage de Huffman qui sera utilisé pour encoder les informations générées par les deux autres. L'annexe A vous donnera un exemple complet de compression de blocs.

2.8.1 Le codage de Huffman

Les codes de Huffman sont appelés codes *préfixés*. C'est une technique de codage statistique à longueur variable.

Les codes de Huffman associent aux symboles les plus utilisés les codes les plus petits et aux symboles les moins utilisés les codes les plus longs. Si on prend comme exemple la langue française, avec comme symboles les lettres de l'alphabet, on coderait la lettre la plus utilisée

(le '*e*') avec le code le plus court, alors que la lettre la moins utilisée (le '*w*' si on ne considère pas les accents) serait codée avec un code plus long. Notons qu'on travaille dans ce cas sur toute la langue française. Si on voulait être plus performant, on travaillerait avec un dictionnaire de Huffman propre à un texte. Le JPEG exploite cette remarque, les codes de Huffman utilisés sont propres à chaque frame JPEG.

Ces codes sont dits *préfixés* car par construction aucun code de symbole, considéré ici comme une suite de bits, n'est le préfixe d'un autre symbole. Autrement dit, si on trouve une certaine séquence de bits dans un message et que cette séquence correspond à un symbole qui lui est associé, cette séquence correspond forcément à ce symbole et ne peut pas être le début d'un autre code.

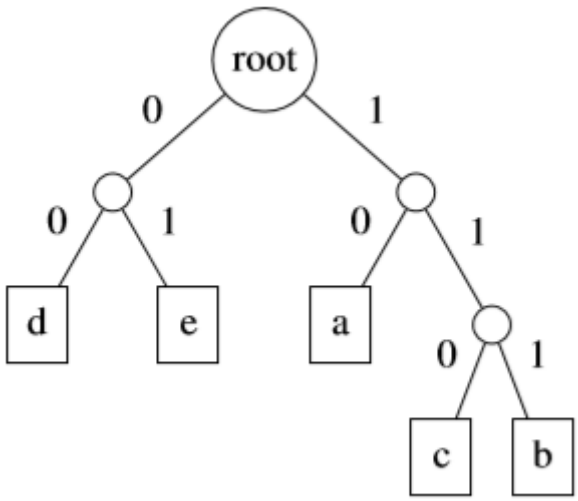
Ainsi, il n'est pas nécessaire d'avoir des « séparateurs » entre les symboles même s'ils n'ont pas tous la même taille, ce qui est ingénieux.[9] Par contre, le droit à l'erreur n'existe pas : si l'on perd un bit en route, tout le flux de données est perdu et l'on décodera n'importe quoi.

La construction des codes de Huffman n'entre pas dans le cadre initial de ce projet. Par contre, il est nécessaire d'en comprendre la représentation pour pouvoir les utiliser correctement.

Un code de Huffman peut être représenté par un arbre binaire. Les feuilles de l'arbre représentent les symboles et à chaque nœud correspond un bit du code : à gauche, le `'0'`, à droite, le `'1'`.

Le petit exemple suivant illustre ce principe :

Symbole	Code
a	10
b	111
c	110
d	00
e	01



Le décodage du bitstream `0001110100001` produit la suite de symboles `decade`. Il aurait fallu 3 bits par symbole pour distinguer 5 symboles avec un code de taille fixe (où tous les codes ont même longueur), et donc la suite `decade` de 6 symboles aurait requis 18 bits, alors que 13 seulement sont nécessaires ici.

Cette représentation en arbre présente plusieurs avantages non négligeables, en particulier pour la recherche d'un symbole associé à un code. On remarquera que les feuilles de l'arbre représentent un code de longueur « la profondeur de la feuille ». Cette caractéristique est utilisée pour le stockage de l'arbre dans le fichier (voir ci-dessous). Un autre avantage réside dans la recherche facilitée du symbole associé à un code : on parcourt l'arbre en prenant le sous arbre de gauche ou de droite en fonction du bit lu, et dès qu'on arrive à une feuille terminale, le symbole en découle immédiatement. Ce décodage n'est possible que parce que les codes de Huffman sont préfixés.

> **"I know JPEG ! - Show me." (citation pourrie)**
>
> Autre particularité, **la norme interdit les codes exclusivement composés de 1**. L'arbre de Huffman présenté ci-dessus ne serait donc pas valide du point de vue du JPEG, et il faudrait stocker le symbole `b` sur une feuille de profondeur 4 pour construire un arbre conforme.

Dans le cas du JPEG, les tables de codage[10] sont fournies avec l'image. On notera que la norme requiert l'utilisation de plusieurs arbres pour compresser plus efficacement les différentes composantes de l'image.

Ainsi, en mode baseline, l'encodeur supporte quatre tables:

- deux tables pour la luminance Y, une pour les coefficients DC et une pour les coefficients AC;
- deux tables communes aux deux chrominances Cb et Cr, une DC et une AC.

Les différentes tables sont caractérisées par un indice et par leur type (AC ou DC). Lors de la définition des tables (marqueur DHT) dans le fichier JPEG, l'indice et le type sont donnés. Lorsque l'on décode l'image encodée, la correspondance indice/composante (Y, Cb, Cr) est donnée au début et permet ainsi le décodage. Attention donc à toujours utiliser le bon arbre pour la composante et le coefficient en cours de traitement.

Le format JPEG stocke les tables de Huffman d'une manière un peu particulière, pour gagner de la place. Plutôt que de donner un tableau représentant les associations codes/valeurs de l'arbre pour l'image, les informations sont fournies en deux temps. D'abord, on donne le nombre de codes de chaque longueur comprise entre 1 et 16 bits. Ensuite, on donne les valeurs triées dans l'ordre des codes. Pour reconstruire la table ainsi stockée, on fonctionne donc profondeur par profondeur. Ainsi, on sait qu'il y a n_p codes de longueur $p, p = 1, \ldots, 16$. Notons que, sauf pour les plus longs codes de l'arbre, on a toujours $n_p \leq 2^p - 1$. On va donc remplir l'arbre, à la profondeur 1, de gauche à droite, avec les n_1 valeurs. On remplit ensuite la profondeur 2 de la même manière, toujours de gauche à droite, et ainsi de suite pour chaque profondeur.

Pour illustrer, reprenons l'exemple précédent. On aurait le tableau suivant pour commencer :

Longueur	Nombre de codes
1	0
2	3
3	2
4	0
...	...
16	0

Ensuite, la seule information que l'on aurait serait l'ordre des valeurs :

$$\langle d, e, a, c, b \rangle$$

soit au final la séquence suivante, qui représente complètement l'arbre :

$$\langle 03200000000000000 deacb \rangle$$

Dans le cas du JPEG, les tables de Huffman permettent de coder (et décoder) des symboles pour reconstruire le coefficient DC et les coefficients AC d'un bloc fréquentiel.

2.8.2 Coefficient continu: DPCM, magnitude et arbre DC

Le coefficient continu (DC) d'un bloc représente la moyenne du bloc. Donc, sauf en cas de changement brutal ou de retour à la ligne, il a de grandes chances d'être proche en valeur de celui des blocs voisins dans la même composante. C'est pourquoi on ne code pas directement la valeur DC d'un bloc mais plutôt la différence par rapport au coefficient DC du bloc précédent (dit prédicateur). Pour le premier bloc, on initialise le prédicateur à 0. Ce codage s'appelle DPCM (*Differential Pulse Code Modulation*).

2.8.2.1 Représentation par magnitude

La norme permet d'encoder une différence comprise entre -2047 et 2047. Si la distribution de ces valeurs était uniforme, on aurait recours à un codage sur 12 bits. Or les petites valeurs sont beaucoup plus probables que les grandes. C'est pourquoi la norme propose de classer les valeurs par ordre de magnitude, comme le montre le tableau ci-dessous.

Magnitude	valeurs possibles
0	`0`
1	`-1, 1`
2	`-3, -2, 2, 3`
3	`-7, ..., -4, 4, ..., 7`

Magnitude	valeurs possibles
...	...
11	`-2047, ..., -1024, 1024, ..., 2047`

Classes de magnitude des coefficients DC (pour AC, la classe max est 10 et la magnitude 0 n'est jamais utilisée).

Une valeur dans une classe de magnitude m donnée est retrouvée par son "indice", codé sur m bits. Ces indices sont définis par ordre croissant au sein d'une ligne du tableau. Par exemple, on codera `-3` avec la séquence de bits `00` (car c'est le premier élément de la ligne), `-2` avec `01` et `7` avec `111`.

De la sorte, on n'a besoin que de 4 + m bits pour coder une valeur de magnitude m : 4 bits pour la classe de magnitude et m pour l'indice dans cette classe. S'il y a en moyenne plus de magnitudes inférieures à 8 (4 + m = 12 bits), on gagne en place.

2.8.2.2 Encodage dans le flux de bits: Huffman

Les classes de magnitude ne sont pas encodées directement dans le flux binaire. Au contraire un arbre de Huffman DC est utilisé afin de minimiser la longueur (en nombre de bits) des valeurs les plus courantes. C'est donc le chemin (suite de bits) menant à la feuille portant la classe considérée qui est encodé.

Ainsi, le *bitstream* au niveau d'un début de bloc contient un symbole de Huffman à décoder donnant une classe de magnitude m, puis une séquence de m bits qui est l'indice dans cette classe.

2.8.3 Arbres AC et codage RLE

Les algorithmes de type *Run Length Encoding* ou RLE permettent de compresser sans perte en exploitant les répétitions successives de symboles. Par exemple, la séquence `000b0eeeeed` pourrait être codée `30b05ed`. Dans le cas du JPEG, le symbole qui revient le plus souvent dans les blocs après quantification est le 0. L'utilisation du zig-zag (section 2.7.1) permet de ranger les coefficients des fréquences en créant de longues séquences de 0 à la fin, qui se prêtent parfaitement à une compression de type RLE.

2.8.3.1 Codage des coefficients AC

Chacun des 63 coefficients AC non nul est codé par un symbole sur un octet suivi d'un nombre variable de bits.

Le symbole est composé de 4 bits de poids fort qui indiquent le nombre de coefficients zéro qui précédent le coefficient actuel et 4 bits de poids faibles qui codent la classe de magnitude du coefficient, de la même manière que pour la composante DC (voir 2.8.2). Il est à noter que les 4 bits de la partie basse peuvent prendre des valeurs entre 1 et 10 puisque le zéro n'a pas besoin d'être codé et que la norme prévoit des valeurs entre -1023 et 1023 uniquement.

Ce codage permet de sauter au maximum 15 coefficients AC nuls. Pour aller plus loin, des symboles particuliers sont en plus utilisés:

- code `ZRL` : `0xF0` désigne un saut de 16 composantes nulles (et ne code pas de composante non nulle) ;
- code `EOB` : `0x00` (*End Of Block*) signale que toutes les composantes AC restantes du bloc sont nulles.

Ainsi, un saut de 21 coefficients nuls serait codé par (`0xF0, 0x5?`) où le ' `?` ' est la classe de magnitude du prochain coefficient non nul. La table ci-après récapitule les symboles RLE possibles.

Symbole RLE	Signification
`0x00`	*End Of Block*
`0xF0`	16 coefficients nuls
`0x?0`	symbole invalide (interdit!)
`0x` $\alpha\gamma$	α coefficients nuls, puis coefficient non nul de magnitude γ

Pour chaque coefficient non nul, le symbole RLE est ensuite suivi d'une séquence de bits correspondant à l'indice du coefficient dans sa classe de magnitude. Le nombre de bits est la magnitude du coefficient, comprise entre 1 et 10.

2.8.3.2 Encodage dans le flux

Les symboles RLE sur un octet (162 possibles) ne sont pas directement encodés dans le flux, mais là encore un codage de Huffman est utilisé pour minimiser la taille symboles les plus courants. On trouve donc finalement dans le flux (*bitstream*), après le codage de la composante DC, une alternance de symboles de Huffman (à décoder en symboles RLE) et d'indices de magnitude.

2.8.3.3 Décodage

Pour ce qui est du décodage, il faut évidemment faire l'inverse et donc étendre les données compressées par les algorithmes de Huffman et de RLE pour obtenir les données d'un bloc sous forme d'un vecteur de 64 entiers représentant des fréquences.

2.9 Lecture dans le flux JPEG

2.9.1 Structure d'un fichier JPEG

Une image JPEG est stockée dans un fichier binaire considéré comme un flux d'octets, appelé dans la suite le *bitstream*. La norme JFIF dicte la façon dont ce flux d'octets est organisé. En pratique, le *bitstream* représente l'intégralité de l'image encodée et il est constitué d'une succession de *marqueurs* et de données. Les marqueurs permettent d'identifier ce que représentent les données qui les suivent. Cette identification permet ainsi, en se référant à la norme, de connaître la sémantique des données, et leur signification (*i.e.*, les actions à effectuer pour les traiter lors du décodage). Un marqueur et ses données associées représentent une *section*.

On distingue deux grands types de sections :

- celles qui permettent de définir l'environnement : ces sections contiennent des données permettant d'initialiser le décodage du flux. La plupart des informations du JPEG étant dépendantes de l'image, c'est une étape nécessaire. Les informations à récupérer concernent, par exemple, la taille de l'image, les facteurs d'échantillonnage ou les tables de Huffman utilisées. Elles peuvent nécessiter un traitement particulier avant d'être utilisables. Il arrive souvent qu'on fasse référence à ces sections comme faisant partie de *l'en-tête* d'un fichier JPEG ;
- celles qui permettent de représenter l'image : ce sont les données brutes qui contiennent l'image encodée.

Une liste exhaustive des marqueurs est définie dans la norme JFIF. Les principaux vous sont donnés en annexe B de ce document, avec la représentation des données utilisées et la liste des actions qui leur sont associées lors du décodage de l'image. On notera ici 4 marqueurs importants :

- `SOI` : le marqueur *Start Of Image* n'apparait qu'une fois par fichier JFIF, il représente le début du fichier ;
- `SOF` : le marqueur *Start Of Frame* marque le début d'une *frame* JPEG, c'est-à-dire le début de l'image effectivement encodée avec son en-tête et ses données brutes. Le marqueur SOF est associé à un numéro, qui permet de repérer le type d'encodage utilisé. Dans notre cas, ce sera toujours *SOF0*. La section SOF contient notamment la taille de l'image et les facteurs d'échantillonnage utilisés. Un fichier JFIF peut éventuellement contenir plusieurs frames et donc plusieurs marqueurs SOF, par exemple s'il représente une vidéo au format MJPEG qui contient une succession d'images. Nous ne traiterons pas ce cas ;
- `SOS` : le marqueur *Start Of Scan* indique le début des données brutes de l'image encodée. En mode *baseline*, on en trouve autant que de marqueurs SOF dans un fichier JFIF (1 dans notre cas). En mode *progressive* les données des différentes résolutions sont dans des *Scans* différents ;

- `EOI` : le marqueur *End Of Image* marque la fin du fichier et n'apparait donc qu'une seule fois.

2.9.2 Byte stuffing

Dans le flux des données brutes des blocs compressés, un *scan*, il peut arriver qu'une valeur `0xff` alignée apparaisse. Cependant, cette valeur est particulière puisqu'elle pourrait aussi marquer le début d'une section JPEG (voir annexe B). Afin de permettre aux décodeurs de faire un premier parcours du fichier en cherchant toutes les sections, ou de se rattraper lorsque le flux est partiellement corrompu par une transmission peu fiable, la norme prévoit de distinguer ces valeurs à décoder des marqueurs de section. Une valeur `0xff` à décoder est donc toujours suivie de la valeur `0x00`, c'est ce qu'on appelle le *byte stuffing*. Pensez bien à ne pas interpréter ce `0x00` pour reconstruire les blocs, il faut jeter cet octet nul !

---

1. Donc votre projet C est en fait un "décodeur `ISO/IEC IS 10918-1 | ITU-T Recommendation T.81`". Qu'on se le dise! ↩
2. Il s'agit là de fréquences spatiales 2D avec une dimension verticale et une dimension horizontale. ↩
3. C'est-à-dire qui cherche la quantité minimale d'information nécessaire pour représenter un message, aussi appelé entropie. ↩
4. L'utilisation du YUV vient de la télévision. La luminance seule permet d'obtenir une image en niveau de gris, le codage YUV permet donc d'avoir une image en niveaux de gris ou en couleurs, en utilisant le même encodage. Le YCbCr est une version corrigée du YUV. ↩
5. A l'encodage, ce décalage des valeurs de [0; 255] vers [-128; 127] permet d'utiliser au mieux le codage par magnitude, avec des valeurs positives et négatives et de plus faible amplitude. ↩
6. La seule contrainte imposée par la norme JPEG étant que la somme sur i des $h_i \times v_i$ soit inférieure ou égale à 10. ↩
7. Pour être précis, ces valeurs représentent la fréquence d'échantillonnage, mais on ne s'en préoccupera pas ici. ↩
8. L'intérêt de ces formules n'est visible en terme de performance que si l'on effectue des opérations en point fixe ou lors d'implantation en matériel (elles aussi en point fixe). ↩
9. Pour une fréquence d'apparition des symboles connue et un codage de chaque symbole indépendamment des autres, ces codes sont optimaux. ↩
10. Chacune représentant un arbre de Huffman. ↩