

Ensimag - Printemps 2023 - Projet Logiciel en C

Sujet : Décodeur JPEG



Auteurs : Des enseignants actuels et antérieurs du projet C

Introduction

Le format JPEG est l'un des formats les plus répandus en matière d'image numérique. Il est en particulier utilisé comme format de compression par la plupart des appareils photo numériques, étant donné que le coût de calcul et la qualité sont acceptables, pour une taille d'image résultante petite.

Le *codec*¹ JPEG est tout d'abord présenté en détail au chapitre 2 et illustré sur un exemple en annexe 5. Le chapitre 3 présente les spécifications, les modules et outils fournis, et quelques conseils importants d'organisation pour ce projet. Finalement, le chapitre 4 formalise le travail à rendre et les détails de l'évaluation.

L'objectif de ce projet est de **réaliser en langage C un décodeur qui convertit les images format compressé (JPEG) en un format brut (PPM)**. Il est nécessaire d'avoir une bonne compréhension du codec, qui reprend notamment des notions vues dans les enseignements Théorie de l'information et Bases de la Programmation Impérative. Mais l'essentiel de votre travail sera bien évidemment de **concevoir** et d'**implémenter** votre décodeur en langage C.

Bon courage à toutes et tous, et bienvenue dans le monde merveilleux du JPEG ! *Enjoy* !

1. *code-decode*, format ou dispositif de compression/décompression d'informations ↩

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. Évidemment, 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 entropique³ 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:

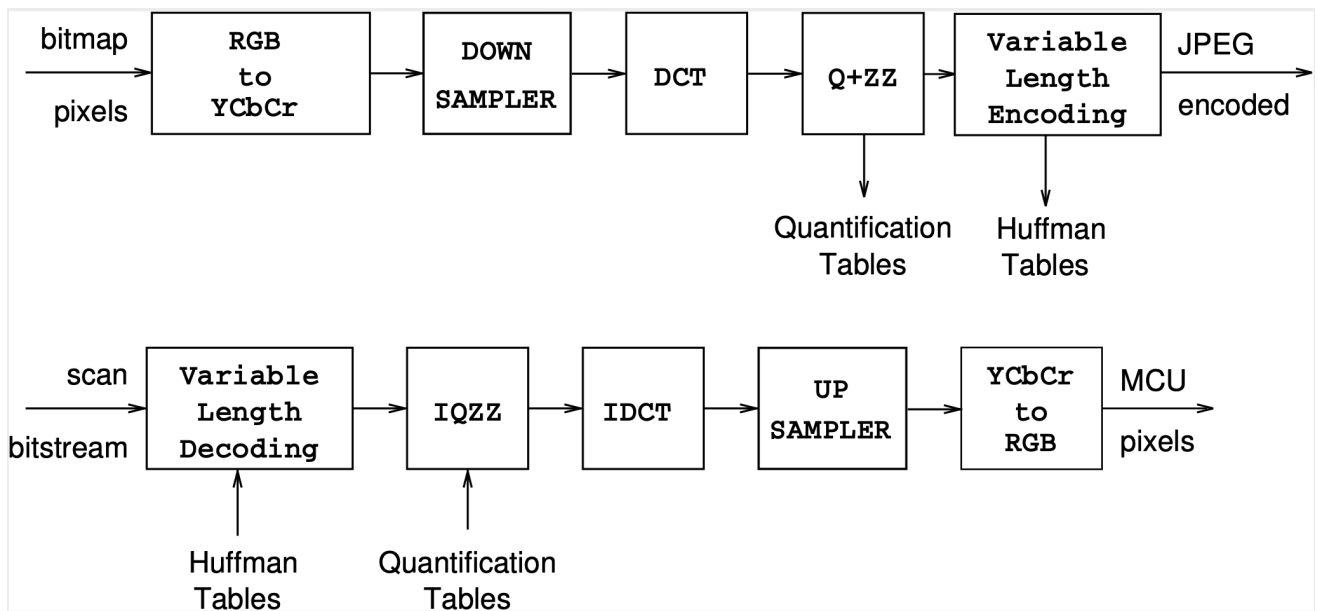


Figure 2.1 : Principe du codec JPEG: opérations de codage (en haut) et de décodage (en bas).

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** **données 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 incluses 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.



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 $n \times n$ 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 remarquerez rapidement que la version 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 $256 \times 3 = 768$ 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

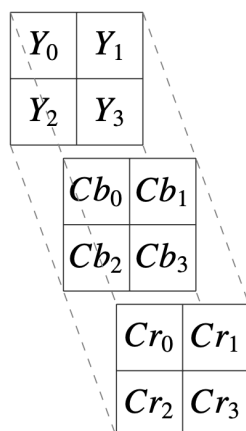
applique le même raisonnement sur les pixels de l'image consécutifs verticalement, on se retrouve à associer à 4 pixels un seul échantillon par chrominance, et on tombe à une occupation mémoire de $256 + 64 + 64 = 384$ échantillons.

2.5.1 Au codage: sous-échantillonnage de l'image

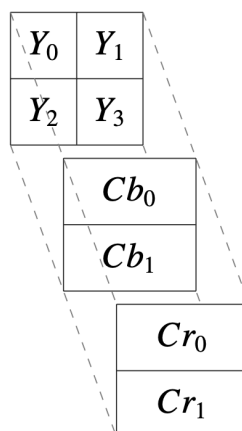
Dans ce document, nous utiliserons une notation directement en lien avec les valeurs présentes dans les sections JPEG de l'en-tête, décrites en [annexe B](#), qui déterminent le facteurs d'échantillonnages (*sampling factors*). Ces valeurs sont identiques partout dans une image. En pratique, on utilisera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

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

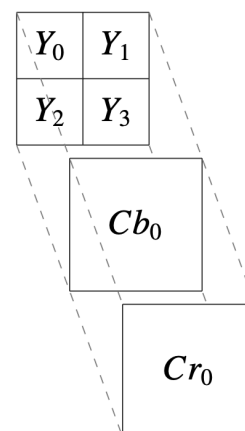
où h_i et v_i représentent le nombre de blocs horizontaux et verticaux pour la composante i . Comme Y n'est jamais compressé, **le facteur d'échantillonnage de Y donne les dimensions de la MCU en nombre de blocs**. Les sous-échantillonnages les plus courants sont décrits ci-dessous. Votre encodeur devra 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 1×1 ($h_1 = v_1 = 1$), mais on pourrait très bien trouver des images sans sous-échantillonnage découpées en MCUs de tailles différentes, comme par exemple 2×2 ou 1×2 .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 $2 \times 2, 1 \times 2, 1 \times 2$ et $2 \times 1, 1 \times 1, 1 \times 1$ 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'ordonnement 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 différent. 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 apparaître 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.

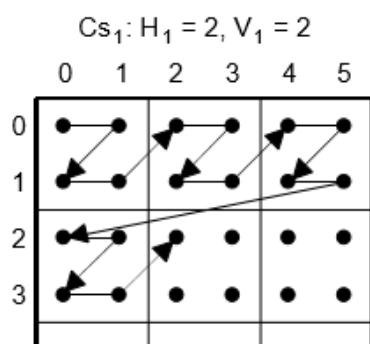


Figure 2.4 : Ordonnement de blocs à 3 composantes de couleur

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 :

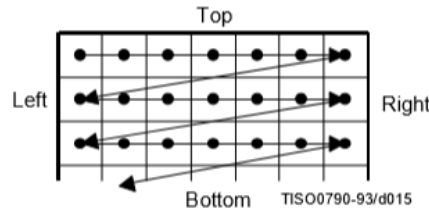


Figure 2.5 : Ordonnancement de blocs en niveaux de gris

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

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

$$G = Y - 0,34414 \times (C_b - 128) - 0,71414 \times (C_r - 128)$$

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

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.

0	1	2	3	4	5	
6	...					
			...	22	23	
24	25	26	27	28	29	

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 8×8 (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.

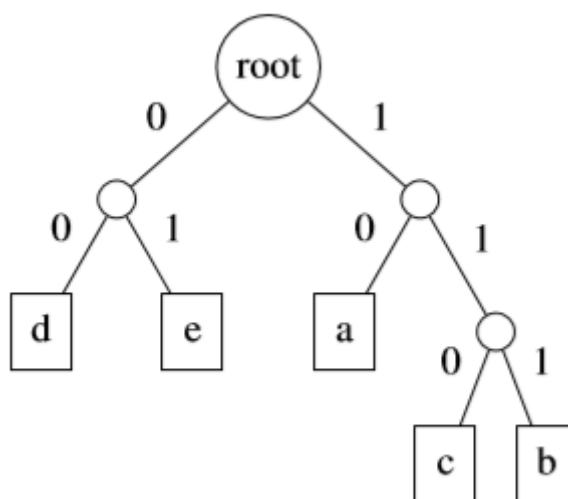
Ainsi, il n'est pas nécessaire d'avoir des « séparateurs » entre les symboles même s'ils n'ont pas tous la même taille, ce qui est ingénieux.⁹ Par contre, le droit à l'erreur n'existe pas : si l'on perd un bit en route, tout le flux de données est perdu et l'on décodera n'importe quoi.

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 codage¹⁰ 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 $p, p = 1, \dots, 16$. 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 :

$$\langle 0320000000000000deacb \rangle$$

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 prédicteur). Pour le premier bloc, on initialise le prédicteur 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 000b0eceeded 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 précèdent 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	<i>End Of Block</i>
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'apparaît 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'apparaît 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 !

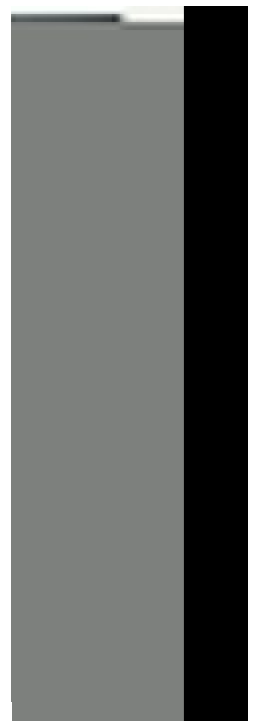
-
1. Donc votre projet C est en fait un "décodeur `ISO/IEC IS 10918-1 | ITU-T Recommendation T.81`". Qu'on se le dise! ↩
 2. Il s'agit là de fréquences spatiales 2D avec une dimension verticale et une dimension horizontale. ↩
 3. C'est-à-dire qui cherche la quantité minimale d'information nécessaire 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 préoccupe 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. ↩

3. Le JPEG pour les glands : mode progressif

Le chapitre précédent vous a présenté le mode baseline sequential (DCT, Huffman, précision 8 bits), qui est le mode JPEG le plus simple et le plus répandu. Pourtant la norme JPEG définit plusieurs autres modes (progressif, sans perte (*lossless*) ou hiérarchique) et alternatives de codage (codage arithmétique au lieu de Huffman, précision 12 bits).

Nous allons maintenant nous intéresser au second mode très utilisé avec le séquentiel, qui est le mode progressif. En mode séquentiel, l'ordre d'encodage (et donc de décodage) des MCUs d'une image est de gauche à droite et de haut en bas (voir figure 2.6). Si une image est très grande ou chargée via une connexion lente, elle est décodée et potentiellement affichée progressivement « par bandes » de haut en bas à mesure du chargement des données.

L'objectif du mode progressif est de pouvoir fournir très vite un aperçu global de l'image, de qualité dégradée mais à la taille finale, puis d'affiner les détails de l'image au fur et à mesure que le chargement/décodage se poursuit. La vidéo ci-dessous, tirée de https://cloudinary.com/blog/progressive_jpegs_and_green_martians, illustre le processus de décodage d'une image JPEG en mode *baseline sequential* (à gauche) et en mode *progressive* (à droite).



0:00 / 0:50

La qualité finale d'une image est la même dans les deux modes. Le mode *progressive* peut permettre une compression plus efficace, grâce à l'utilisation de tables de Huffman spécifiques pour les différents scans. Pour autant, la taille d'un fichier JPEG progressif est généralement plus importante qu'en séquentiel en raison d'un nombre plus important de scans. Le temps de total de codage/décodage est aussi plus important en progressif, mais l'avantage du progressif est d'avoir l'aperçu rapide d'une image complète, même dégradée, avant la fin du décodage.

Le principe du mode progressif est le suivant. Contrairement au mode séquentiel, les données ne sont pas transmises en un seul scan mais en plusieurs. Il y a donc plusieurs sections de type **SOS** (Start of Scan ; voir annexe B.2.7) dans le flux JPEG. Chaque scan contient bien tous les blocs nécessaires à la reconstruction de l'image, mais les données de chaque bloc ne sont que partiellement encodées. Il existe deux procédures d'encodage partiel des données :

- *spectral selection* : chaque scan ne fournit qu'une « bande » de coefficients parmi les 64 d'un bloc. Le premier scan ne fournit toujours que le coefficient 0 (DC), ce qui permet de reconstruire des blocs 8x8 de couleur uniforme (la valeur moyenne de chaque composante). Les scans suivants contiennent ensuite des groupes de coefficients AC de basses puis de hautes fréquences (1 à 63), jusqu'à ce que tous aient été transmis. L'œil est moins sensible aux hautes fréquences, qui ne sont transmises qu'en dernier.
- *successive approximation* : l'autre procédure consiste à ne pas encoder totalement les 8 bits d'un coefficient dans un seul scan, mais plutôt à encoder d'abord ses poids forts (plus informatifs) puis les poids plus faibles dans des scans suivants. Il est possible d'utiliser soit la procédure *spectral selection* seule, soit la procédure *full progression* qui combine *spectral selection* et *successive approximation*.

La structure d'un fichier JPEG en mode progressif est très similaire à celle d'un fichier séquentiel, suivant les spécifications décrites en annexe B. Les différences principales sont que le fichier contient plusieurs scans (sections **SOS**) et que les tables de Huffman (sections **DHT**) peuvent être redéfinies plusieurs fois afin d'être optimales pour le(s) scan(s) suivant(s). Voilà. Pour en savoir plus, tout se passe dans la norme JPEG (en particulier l'annexe G)! ¹

1. Vous êtes grands maintenant, on n'allait quand même pas vous en dire plus. ↩

4. Spécifications, modules fournis et organisation

Ce chapitre décrit ce qui est attendu, ce qui vous est fourni et comment l'utiliser, et quelques conseils sur la méthode à suivre pour mener à bien ce projet.

4.1 Spécifications

4.1.1 Décodeur JPEG baseline

Les spécifications sont très simples : vous devez implémenter un programme qui convertit une image JPEG baseline séquentiel en une image au format brut PPM :

- le nom de l'exécutable doit être `jpeg2ppm` ;
- il prend en unique paramètre le nom de l'image JPEG à convertir, d'extension `.jpeg` ou `.jpg`. Seules les images encodées en mode {JFIF, baseline sequential, DCT, Huffman, 8 bits} sont acceptées (application de type APP0, frame SOF0), avant d'éventuelles extensions ;
- en sortie une image au format PPM sera générée, de même nom que l'image d'entrée et d'extension `.ppm` si elle est en couleur ou `.pgm` si elle est en niveaux de gris.
- vérifiez bien que l'image de sortie est dans le même répertoire que l'image d'entrée. Par exemple une image dont le nom est `"../..images/zut.jpg"` sera décodée dans `"../..images/zut.ppm"`

Par exemple :

```
./jpeg2ppm shaun_the_sheep.jpeg
```

génèrera le fichier `shaun_the_sheep.ppm`. C'est tout. Aucune trace particulière n'est attendue, hormis peut-être en cas d'erreur. Vous pouvez si le souhaitez ajouter des options, par exemple `-v` (verbose) pour afficher des informations supplémentaires.¹

❗ Respect des spécifications

Il est très important de respecter ces consignes simple, pour que tous les projets puissent être facilement évalués par des tests automatiques et par les enseignants lors des soutenances.

Si votre programme s'appelle plutôt `mon_decodeur`, qu'il attend des paramètres bizarres, et qu'il génère une image décodée qui s'appelle toujours `toto.ppm`, ce sera vite ingérable. Ce qui sera mauvais pour notre humeur, et donc *in fine* pour votre note...

4.1.2 Décodeur JPEG progressif

Dans un second temps (si vous avancez bien, hein) votre décodeur sera étendu pour accepter également des images JPEG en mode *{JFIF, progressive, DCT, Huffman, 8 bits}*, application de type `APP0`, frame `SOF2`. Vous pourrez sauvegarder une image PPM après chaque scan, de qualité croissante. Si l'envie (et le temps) vous en prend, vous pourriez aussi afficher les données décodées directement dans une fenêtre graphique.

Attention cependant, la gestion du JPEG progressif est considérée comme une extension du projet, et ne sera évaluée à la soutenance que si votre décodeur JPEG baseline est terminé. Plus d'infos là-dessus dans la section sur les extensions possibles sur cette page.

4.1.3 Format de fichier de sortie PPM

Une fois décodées, les images seront enregistrées au format au format PPM (Portable PixMap) dans le cas d'une image en couleur, ou PGM (Portable GreyMap) pour le cas en niveaux de gris. Il s'agit d'un format « brut » très simple, sans compression, que vous avez peut-être déjà eu l'occasion d'utiliser dans la partie préparation au langage C et dans le projet de BPI.

Le principe est de lire d'abord un en-tête *textuel* comprenant:

- un *magic number* précisant le format de l'image, `P6` pour des pixels à trois couleurs RGB ;
- la largeur et la hauteur de l'image, en nombre de pixels ;
- le nombre de valeurs d'une composante de couleur (255 dans notre cas: chaque couleur prend une valeur entre 0 et 255) ;

Ensuite, les données de couleur sont directement écrites en **binaire** : trois octets pour les valeurs R, G et B du premier pixel, puis trois autres pour le second pixel, et ainsi de suite. Le format PGM est une variante pour les images en niveaux de gris : l'identifiant est cette fois `P5`, et la couleur de chaque pixel est codée sur un seul octet dont la valeur varie entre 0 (noir) et 255 (blanc). Un exemple est donné ci-dessous :

```
$ hexdump -C cocorico.ppm

00000000  50 36 0a 33 20 32 0a 32  35 35 0a 00 00 ff ff ff  |P6.3 2.255.....|
00000010  ff ff 00 00 00 00 ff ff  ff ff ff 00 00  |.....|

$ hexdump -C cocorico_bw.ppm

00000000  50 35 0a 33 20 32 0a 32  35 35 0a 12 ff 36 12 ff  |P5.3 2.255...6..|
00000010  36                                     |6|
```

En haut, trace de la commande `hexdump -C` sur une image 3×2 représentant un drapeau français. Notez les caractères ASCII de l'en-tête textuel puis la suite des couleurs RGB, pixel par pixel. En bas, la version en niveaux de gris (français, italien, irlandais ? C'est moins clair...). Cette fois, il n'y a qu'un seul octet de couleur par pixel.

4.2 Organisation, démarche conseillée

Vous êtes bien entendu libres de votre organisation pour mener à bien votre projet, selon les spécifications présentées sur cette page. Nous vous proposons tout de même une démarche générale, incrémentale, adaptée aux images de test fournies. À vous de la suivre, de l'adapter ou de l'ignorer, selon votre convenance!

4.2.1 Résumé des étapes & difficultés

Pour résumer, les étapes du décodeur sont les suivantes :

- 1. Extraction de l'entête JPEG, récupération de la taille de l'image, des facteurs d'échantillonnage et des tables de quantification et de Huffman ;
- 2. Décodage de chaque MCU :
 - 2.1. Reconstruction de chaque bloc :
 - Extraction (lecture dans le flux de bits) et décompression ;
 - Quantification inverse (multiplication par les tables de quantification) ;
 - Réorganisation zig-zag ;
 - Calcul de la transformée en cosinus discrète inverse (iDCT) ;
 - 2.2. Mise à l'échelle des composantes Cb et Cr (*upsampling*), en cas de sous-échantillonnage ;
 - 2.3. Reconstruction des pixels : conversion YCbCr vers RGB ;
- 3. Ecriture du résultat dans le fichier PPM.

Vous trouverez ci-dessous une estimation de la difficulté de ces différentes étapes :

Etape	Difficulté pressentie
Lecture en-tête JPEG	★★★ à ★★★★★
Gestion des tables de Huffman	★★★★★
Extraction des blocs	★★★
Quantification inverse	★
Zig-zag	★★
iDCT	★★
<i>Upsampling</i>	★★★★
Conversion YCbCr vers RGB	★
Ecriture du fichier PPM	★★ à ★★★★★
Gestion complète du décodage	★★ à ★★★★★★

Pour faire tout ça vous allez aussi devoir lire dans le flux de données, parfois pour lire des octets mais souvent pour lire 3 bits, 11 bits, 1 bit, etc. Humm, ça vaut bien "quelques" ★!

Pour l'extension au mode progressif, à vous de nous mettre des ★ plein les yeux !

4.2.2 Progression incrémentale sur les images à décoder

Bien évidemment, votre décodeur devra *in fine* être capable de traiter toutes les images JPEG qui répondent aux spécifications du sujet. Mais vouloir résoudre d'emblée le problème complet est risqué : il est possible que vous ayez pu implémenter la quasi-totalité des étapes mais sans avoir pu les valider ou finir de les intégrer. Vous aurez alors fourni un travail conséquent et écrit de magnifiques "bouts de programme", mais n'aurez pas écrit un décodeur qui fonctionne !

Nous vous conseillons donc d'adopter une approche dite **incrémentale** :

- L'objectif est d'obtenir le plus rapidement possible un décodeur **fonctionnel**, même s'il ne permet de traiter que des images très simples ;
- Chacune des étapes sera ensuite complétée (voire totalement reprise) pour couvrir des spécifications de plus en plus complètes.

En plus d'être efficace et rationnelle, cette approche permet d'avoir toujours un programme fonctionnel, même incomplet, à présenter à votre client (nous), à tout instant. Imaginez que le rendu soit subitement avancé de deux jours², et bien vous rendrez votre projet dans l'état courant, sans (trop de) stress ; et hop. Et votre client sera satisfait, ce qui est bon pour lui et au final pour vous³!

Plusieurs images de tests sont fournies, de "complexité" croissante. Il est conseillé de travailler sur ces images dans l'ordre suggéré (dans l'ordre du tableau), permettant une progression graduelle d'un décodeur simple vers celui capable de traiter des images quelconques.













Image	Caractéristiques	Progression
 invader	<ul style="list-style-type: none"> • 8x8 (un seul bloc) 	<ul style="list-style-type: none"> • Niveaux de gris (une seule composante) • Encodage MCU très simple • PPM simplifié
 poupoupidou_bw	<ul style="list-style-type: none"> • 16x16 	<ul style="list-style-type: none"> • Niveaux de gris • Plusieurs blocs • Pas de troncature
 gris	<ul style="list-style-type: none"> • 320x320 	<ul style="list-style-type: none"> • Niveaux de gris • Plusieurs blocs • Pas de troncature • Je sais plus pourquoi, mais des fois cette im
 bisou	<ul style="list-style-type: none"> • 585x487 	<ul style="list-style-type: none"> • Niveaux de gris • Plusieurs blocs • Troncature à droite et en bas
 poupoupidou	<ul style="list-style-type: none"> • 16x16 	<ul style="list-style-type: none"> • Couleur • Pas de troncature • Glamour
 zig-zag	<ul style="list-style-type: none"> • 480x680 	<ul style="list-style-type: none"> • Couleur • Pas de troncature

Image	Caractéristiques	Progression
 thumbs	<ul style="list-style-type: none"> • 439x324 	<ul style="list-style-type: none"> • Couleur • Tronquée à droite et/ou en bas
 horizontal	<ul style="list-style-type: none"> • 367x367 	<ul style="list-style-type: none"> • Echantillonnage horizontal
 vertical	<ul style="list-style-type: none"> • 704x1246 	<ul style="list-style-type: none"> • Echantillonnage vertical
 shaun_the_sheep	<ul style="list-style-type: none"> • 300x225 	<ul style="list-style-type: none"> • Echantillonnage horizontal ET vertical
 complexite	<ul style="list-style-type: none"> • 2995x2319 	<ul style="list-style-type: none"> • Niveaux de gris • C'est looong...
 biiiiiig	<ul style="list-style-type: none"> • 7392x3240 	<ul style="list-style-type: none"> • Couleur • C'est looooooooooong !



Un développement et une validation incrémentale pour la niveaux de gris serait donc:

1. Décodeur d'images 8x8 en niveaux de gris (ex: `invader.pgm`) ;
2. Extension à des images grises comportant plusieurs blocs (ex: `gris.pgm`) ;
3. Extension à des images dont les dimensions ne sont pas multiples de la taille d'un bloc (ex: `bisou.pgm`) ;

Et bien sûr on procèdera de la même façon pour étendre le décodeur à la gestion des images couleur de complexité croissante.

Ne sous-estimez la longueur/difficulté de l'étape 1, à savoir implémenter un décodeur d' `invader.jpg` ! Certes, commencer par travailler sur cette image simplifie beaucoup le décodage : elle ne contient qu'un seul bloc 8x8, et est en noir et blanc. En revanche, le décodage de cette image nécessite d'avoir mis en place la majeure partie de l'architecture de votre décodeur. Vous devrez en particulier être capable de récupérer les informations nécessaires dans l'entête JPEG avant même de commencer à décoder les données de l'image.

Plus généralement, notez que ces images doivent vous aider à valider certaines parties du projet, **mais ne seront pas suffisantes tester tout votre décodeur**. Il sera en particulier utile/nécessaire d'ajouter des **tests unitaires** pour tester hors contexte des parties spécifiques du décodeur (par exemple est-il possible de valider l'iDCT en tant que telle, pas au milieu du décodage?). Il sera aussi nécessaire d'ajouter **vos propres images** à cette base de tests, par exemple conçues de toute pièce avec `gimp` présentant des caractéristiques spécifiques . En plus de vous aider lors de la mise au point de certaines fonctionnalités du décodeur, vous pourrez vous appuyer sur ces images "*maison*" lors de la soutenance pour démontrer la robustesse de votre implémentation.

! Le piratage c'est du vol

Toutes ces images ne sont pas franchement libres de droits. Mais on les aime bien, alors au moins faisons leur un copyright à notre sauce: relisez vos BDs préférées, parlez l'anglais et l'américain, revoyez Les Temps Modernes, Bêêhh, aimez la physique (à défaut de comprendre), jouez à des jeux d'hier et d'aujourd'hui, Some Like It Hot, et bécotez-vous devant l'hôtel de ville! (mais pas de trop près en ce moment bien sûr. Euh... attends)

! Blague Carambar

L'image "vertical" traduit l'émotion d'un étudiant ou d'une étudiante qui vient d'arriver à finir le *downsampling* à peu près correctement. L'image "horizontal" traduit la réaction de ses enseignants. La première personne qui nous dit de quel(s) album(s) proviennent ces deux images gagne un Carambar! (il y a un piège). Et ce qui est chouette cette année avec le projet en distanciel, c'est que vous gagnez bien le Carambar mais c'est un ou une prof qui le mange!

4.2.3 Découpage en modules & fonctions, spécifications

Par rapport à la plupart des TPs réalisés cette année, une des difficultés de ce projet est sa *taille*, c'est-à-dire la quantité des tâches à réaliser. Une autre est que c'est principalement à vous de définir *comment* vous allez résoudre le problème posé. Avant de partir tête baissée

dans l'écriture de code, il est essentiel de bien définir un découpage en *modules* et en *fonctions* pour les différents éléments à réaliser. Ils doivent être clairement **spécifiés** : rôle, structures de données éventuelles, fonctions proposées et leur signature précise.

Dans le cadre de ce projet en équipe, vous serez amenés à *utiliser* les modules programmés par vos collègues. Il est donc nécessaire de bien comprendre leur usage, même si vous ne connaissez ou ne maîtrisez pas leur contenu. Une bonne spécification est donc fondamentale pour que la mise en commun des différentes parties du projet se fasse sans trop de heurts (vous verrez, ce n'est pas toujours simple...).

Prenons pour exemple l'étape DCT. Il est naturel⁴ d'écrire une fonction spécifique qui réalise cette opération uniquement. Beaucoup de questions sont à soulever :

- Quel nom donner à cette fonction ?
- Dans quels fichiers sera-t-elle déclarée (`.h`) et définie (`.c`) ?
- Quel est son rôle ? Réalise-t-elle le calcul sur un seul bloc, sur plusieurs ?
- Quels sont ses paramètres : un bloc d'entrée et un de sortie ? Un seul bloc qui est modifié au sein de la fonction ? Faut-il allouer de la mémoire ? ...
- Comment est représenté un bloc en mémoire : tableau 1D de 64 valeurs ou tableau 2D de taille 8x8 ?⁵ Adresse dans un tableau de plus grande taille ? ...
- Quel est le type des éléments d'un bloc à ce stade du décodage ?
- ...

Attention, ce travail est difficile ! Mais il est vraiment fondamental, même si vous serez certainement amenés à modifier ces spécifications au fur et à mesure de l'avancement dans le projet (et c'est bien normal).

Dans la mesure du possible (pas toujours facile), chaque module devra être testé de manière autonome c'est-à-dire hors contexte de son utilisation dans le décodeur. Il s'agit de tester avec des entrées contrôlées et de vérifier que les sorties sont bien conformes à la spécification. Ceci sera facile à réaliser pour certaines étapes "simples" du décodage (zig-zag, ...). Sinon vous devrez tester les étapes séquentiellement (les sorties de l'une étant les entrées de la suivante), en comparant notamment les données à l'aide de l'outil `jpeg2b1ab1a` distribué (voir sa description juste après !).

❗ Point de passage obligatoire

Pour éviter les déconvenues du genre *"On est à une semaine du rendu, mon projet est cassé et je n'arrive plus à le corriger parce que mes structures de données que j'ai construites au départ sont mal foutues."* (toute ressemblance avec ce qu'on entend tous les ans est fortuite), vous **devrez** organiser un point d'étape avec un enseignant, où vous lui présenterez l'architecture logicielle de votre projet (vos structures de données, le découpage du projet en modules, le découpage des modules en fonctions). De notre côté, on s'engage à dire tout le mal qu'on pense de ce que vous nous montrerez, ce qui vous

permettra de rectifier le tir avant de vous lancer dans l'implémentation. Bien entendu, plus ce point d'étape a lieu tôt dans le projet, mieux c'est, mais **ce sera à vous de venir nous voir quand vous vous sentirez prêts.**

4.3 Outils et traces pour la mise au point

Cette section introduit quelques outils disponibles pour vous aider à mettre au point votre décodeur.

`jpeg2blabla`

Ce programme est en fait une version « bavarde » de `jpeg2ppm`, réalisée par nos soins, qui fournit deux types d'informations :

- **Paramètres de l'image** : l'option `-v` (pour verbose) affiche dans la console les caractéristiques de l'image JPEG à décoder (celles utiles en mode baseline séquentiel DCT). Un exemple est donné figure 4.2. L'option `-vh` (ou `-v -h`) permet en plus d'afficher la structure des arbres de Huffman, c'est-à-dire les chemins menant aux différents symboles. Un exemple est donné figure 4.3.
- **Traces de toutes les étapes du décodage** : en plus de décoder l'image, cet utilitaire crée également un fichier d'extension `.blabla` qui fournit les valeurs numériques de tous les blocs de chaque MCU, étape après étape. Un exemple est donné figure 4.4.

Ces traces pourront s'avérer trèèèèèè utiles (sur des images de taille réduite) pour valider votre décodeur étape après étape, si nécessaire.

```

[SOI]   marker found
[APP0]  length 16 bytes
        JFIF application
        other parameters ignored (9 bytes).
[DQT]   length 67 bytes
        quantization table index 0
        quantization precision 8 bits
        quantization table read (64 bytes)
[SOF0]  length 11 bytes
        sample precision 8
        image height 225
        image width 300
        nb of component  1
        component Y
            id 1
            sampling factors (hvx) 1x1
            quantization table index 0
[DHT]   length 31 bytes
        Huffman table type DC
        Huffman table index 0
        total nb of Huffman symbols 12
[DHT]   length 81 bytes
        Huffman table type AC
        Huffman table index 0
        total nb of Huffman symbols 62
[SOS]   length 8 bytes
        nb of components in scan 1
        scan component index 0
            associated to component of id 1 (frame index 0)
            associated to DC Huffman table of index 0
            associated to AC Huffman table of index 0
        other parameters ignored (3 bytes)
        End of Scan Header (SOS)

... (image decompression) ...
... Done
[EOI]   marker found
bitstream empty

```

Figure 4.2 : Trace de `./jpeg2blabla -v bisou.jpeg`. L'image est ici en niveaux de gris, donc avec une seule composante de couleur (la luminance Y) et pas de sous-échantillonnage, et utilise une seule table de quantification et deux tables de Huffman (une AC et une DC) définies dans des sections DHT séparées.

```

...
[DHT] length 30 bytes
      Huffman table type DC
      Huffman table index 0
      total nb of Huffman symbols 11
      path: 000 symbol: 2
      path: 001 symbol: 3
      path: 010 symbol: 4
      path: 011 symbol: 5
      path: 100 symbol: 6
      path: 101 symbol: 7
      path: 110 symbol: 8
      path: 1110 symbol: 1
      path: 11110 symbol: 9
      path: 111110 symbol: 0
      path: 1111110 symbol: a
...

```

Figure 4.3 : Trace (partielle) de `./jpeg2blabla -vh bisou.jpeg`. L'option `-h` permet de voir la structure des arbres de Huffman, ici un de type DC.

Voyons maintenant un exemple de trace complète de décodage, dans un fichier `.blabla`. Pour chaque composante de MCU et pour chaque bloc, on trouvera d'abord une première ligne comme:

```

[ DC/AC] 07(4) / 03(6) 04(3) 51(7) 01(3) f0(8) 21(5) 11(4) 41(6) a1(8)
11(4) 81(8) 00(4) -- total 92 bits

```

Comment lire ceci ?

- la magnitude du coefficient DC est 0x07, elle a été lue en consommant 4 bits (== profondeur du symbole 0x07 dans l'arbre de Huffman)
- le premier coefficient AC est 0x03 (6 bits lus), le second 0x4 (3 bits lus), etc.
- au total 92 bits ont été consommés dans le flux pour récupérer tous les coefficients du bloc (nb de bits lus dans les arbres + les magnitudes trouvées)

Ensuite, et toujours pour chaque bloc on trouvera:

- les blocs décompressés, puis après quantification inverse, zig-zag inverse et iDCT
- la composante de la MCU reconstruite à partir du ou des blocs, après éventuel *upsampling*.

La figure ci-dessous fournit un exemple complet d'une MCU en couleur.

[illegible]

```

ffff 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ffff 0 0 0 0 0 0 0 0 0
[ izz] 6 fff7 ffff ffff 0 0 0 0 4 1 ffff 1 1 0 0 1 fff9 0 0 ffff 0 0 0 0 2 1 ffff 0 0 0 0 ffff
ffff 0 0 0 0 0 0 0 1 0 ffff 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
[ idct] 7f 7f 80 81 82 81 81 81 7f 7f 80 80 81 81 82 82 80 80 81 81 83 82 83 83 80 80 81 81 83 83
83 83 80 81 81 82 82 83 83 83 7f 80 80 81 81 82 82 83 7e 7f 80 80 80 81 81 82 7c 7d 7e 7e 7e 7f
80 81
* component mcu
[ mcu] 7f 7f 7f 7f 80 80 81 81 82 82 81 81 81 81 81 81 7f 7f 7f 7f 80 80 80 80 81 81 81 81 82 82
82 82 80 80 80 80 81 81 81 81 83 83 82 82 83 83 83 83 80 80 80 80 81 81 81 81 83 83 83 83 83
83 83 80 80 81 81 81 81 82 82 82 82 83 83 83 83 83 83 7f 7f 80 80 80 80 81 81 81 81 82 82 82 82
83 83 7e 7e 7f 7f 80 80 80 80 80 80 81 81 81 81 82 82 7c 7c 7d 7d 7e 7e 7e 7e 7e 7f 7f 80 80
81 81
...

```

Figure 4.4 : Extrait d'un fichier `.bLAbLa` généré par `jpeg2bLAbLa` sur une image avec sous-échantillonnage horizontal. Pour chaque composante de chaque MCU (la 787e ici, de taille 2x1 blocs) on trouve le(s) blocs décompressé(s), puis après quantification inverse, zig-zag inverse et iDCT, et enfin la composante de la MCU reconstruite après upsampling. Ça pique les yeux au début, mais après quelques jours vous apprécierez !

❗ Monsiiieeur, Madaaammme, j'ai pas pareil.

Il se peut que les valeurs numériques diffèrent légèrement entre votre décodeur et `jpeg2bLAbLa`. Il peut s'agir d'une erreur bien sûr, mais aussi de différences purement numériques en particulier lors de calculs en valeurs flottantes : précision, pas les même arrondis, etc.

Pour le cas particulier des résultats après DCT, sachez que `jpeg2bLAbLa` s'appuie sur l'algorithme de Loeffler pour calculer la DCT. La DCT Loeffler est plus rapide mais ne fournit pas exactement les mêmes valeurs que la DCT naïve.

`hexdump`

Cette application, en particulier avec l'option `-C`, affiche de manière textuelle le contenu octet par octet d'un fichier (un exemple de sortie est disponible en [section 3.1.2](#). Dans ce projet, c'est très utile pour regarder les données d'un fichier JPEG, comprendre sa structure et vérifier que les données lues sont correctes (notamment lorsque vous implémenterez la lecture de l'entête JPEG). Vous pourrez également l'utiliser pour regarder le contenu bit à bit d'un fichier et vérifier que les bits lus par votre décodeur correspondent.

`identify`

C'est un utilitaire de la suite `ImageMagick`, qui décrit le format et les caractéristiques d'une image. À utiliser avec notamment l'option `-verbose`.

`gimp`

C'est un des outils de manipulation d'images les plus connus. Il permet entre autres de générer des fichiers au format JFIF baseline (supporté par votre décodeur) en jouant sur différents paramètres, de faire des modifications dans les fichiers, ou encore d'afficher des images JPEG ou PPM. Pour l'affichage seul, préférez `eog` (Eye Of Gnome), moins gourmand en ressources et plus rapide à invoquer.

`cjpeg`

`cjpeg` est un encodeur JPEG. Développé initialement par le Independent JPEG Group 5, cet encodeur se positionne comme l'implémentation de référence du standard JPEG. Il vous permettra notamment de contrôler certains aspects "ceinture noire" de la norme lors de la conversion des images, comme les facteurs d'échantillonnage, les tables de quantification utilisées ou encore l'utilisation du mode progressif. Si vous souhaitez étendre la base d'images de tests fournie avec le projet, par exemple pour stresser votre décodeur (ou accessoirement, votre trinôme), ou simplement briller en société, c'est l'outil qu'il vous faut !

4.4 Des idées d'améliorations et extensions

Si vous lisez cette section, c'est que vous êtes venus à bout de votre décodeur pour toutes les configurations d'images JPEG.⁶ Félicitations ! Mais pourquoi s'arrêter en si bon chemin, alors voici déjà deux améliorations possibles à regarder. Et si jamais vous vous ennuyez après ceci, venez nous voir⁷ on a toujours d'autres trucs sous le coude!

4.4.1 A la recherche du temps perdu

Votre décodeur decode, cool. Pourtant, il est possible que le temps d'exécution de votre `jpeg2jpeg` soit (nettement) plus important que celui du `jpeg2blabla` distribué, qui écrit pourtant de nombreuses informations dans un fichier texte (ce qui est trèèèès gourmand en temps). Alors?

On s'intéresse d'abord à l'*optimisation* de votre décodeur. Par optimisation, on entend ici *tout ce qui peut le rendre meilleur qu'un décodeur aux fonctionnalités équivalentes*. A niveau de fonctionnalité équivalent, on préférera évidemment un décodeur plus rapide ou qui consomme moins de mémoire⁸.

Y'a les bons et les mauvais décodeurs

Le mauvais décodeur, il voit une image, il la decode...

Afin d'améliorer le temps d'exécution de votre décodeur, commencez par étudier l'impact des optimisations à la compilation. Pour ce faire, jouez avec l'option `-O` du compilateur pour générer différentes versions utilisant différents niveaux d'optimisation (`-O0`, `-O1`, jusqu'à `-O3`). Pensez à toujours vérifier l'intégrité des images générées, les optimisations appliquées à partir du niveau 3 ne garantissant pas toujours l'exactitude des résultats numériques (!).

On utilisera ensuite un outil de *profiling* pour détecter dans quelle partie du décodeur on passe le plus de temps. Vous pouvez par exemple utiliser l'outil GNU `gprof` :

- recompilez votre programme en ajoutant l'option de compilation `-pg` (pour compiler les objets ET pour l'édition de liens) ;
- exécutez votre programme normalement. Un fichier `gmon.out` a normalement été créé ;
- étudiez le résultat à l'aide de la commande: `gprof ./ppm2jpeg gmon.out`. Vous trouverez en particulier la ventilation du temps d'exécution sur les différentes fonctions de votre programme. Sympa, non ?

L'optimisation d'un programme peut être vue comme un processus cyclique, illustré ici sur l'amélioration du temps d'exécution du décodeur :

- Analysez les traces du profiler pour identifier les parties à optimiser en priorité : quelles sont les étapes les plus coûteuses en temps ? Sont-elles améliorables ?
- Posez-vous des questions sur vos structures de données, efficacité des accès mémoires, nombres d'allocations, algorithmes, etc. Puis implémentez ces améliorations et évaluez les.
- **Vérifiez** que votre décodeur fonctionne toujours aussi bien! Ben oui, si ça va plus vite mais que ça marche plus, ça sert à rien. Une suite de tests (dits de non-régression) qui vérifie le bon fonctionnement de votre décodeur pourrait à ce stade être de bon goût.
- **GOTO 1.** Et oui, on peut faire ça à l'infini (ou en gros, jusqu'à qu'on ait soit atteint un niveau de performance souhaité, soit qu'on n'ait plus d'idée.).

Bien entendu, l'approche est transposable à d'autres types d'optimisation que l'amélioration du temps d'exécution.

Exemple d'optimisation: une DCT plus rapide

Sauf surprise, il est très probable qu'au moins 80% du temps soit consommé par l'étape de DCT.

En regardant de plus près l'algorithme présenté en [section 2.4](#), il est clairement de complexité quadratique. Même en bossant depuis votre canapé ces dernières semaines vous aurez compris que ce n'est pas terrible. Donc la DCT est une étape prioritaire à optimiser.

Déjà, on se rend compte qu'on recalcule plusieurs fois les mêmes valeurs de cosinus. Une première optimisation de votre décodeur consiste donc à précalculer et stocker tous les cosinus nécessaires. Même si la complexité de l'algorithme est inchangée, vous allez déjà gagner énormément !

Pour aller plus loin, il faudra s'attaquer à l'algorithme en lui-même pour réduire le nombre d'opérations, les multiplications en particulier. Comme vous l'avez vu peut-être en cours d'Algorithmique, il existe des méthodes de type "Diviser pour régner" pour écrire une version efficace de la transformée de Fourier (*Fast Fourier Transform*, FFT) en $\mathcal{O}(n \log n)$. Des

versions optimales existent en plus dans le cas particulier de la DCT sur un bloc 8x8. Reste plus qu'à les trouver, les comprendre, les implémenter, les valider et se congratuler. Mais comme vous avez des profs extra, allez donc voir en [annexe D](#).

4.4.2 Mode progressif

Comme vous pouvez le constater, tout vous est donné pour réaliser sereinement un décodeur JPEG en mode séquentiel : des explications et exemples aux petits oignons, une proposition de progression incrémentale dans les spécifications et les tests, des belles images, des profs bienveillants, etc.

Pour le mode progressif, on change de paradigme : **débrouillez-vous**. Il vous faudra notamment comprendre la norme, spécifier les restructurations à apporter à votre code, les faire, générer des tests adéquats et valider votre décodeur. L'objectif final est bien d'avoir un programme qui décode les modes séquentiel ET progressif.

Mais attention le progressif c'est (très) dur. Donc :

1. Commencez par finir votre décodeur séquentiel et le valider ;
2. Ensuite seulement abordez la partie progressive. Inutile d'aborder ce mode à la soutenance si votre décodeur panique sur la 2ème image séquentielle testée...

4.5 Informations supplémentaires

Voici quelques documents ou pages Web qui vous permettront d'aller un peu plus loin en cas de manque d'information, ou simplement pour approfondir votre compréhension du JPEG si vous êtes intéressés.

1. En premier lieu, toute l'information sur la norme est évidemment disponible dans le document [ISO/IEC IS 10918-1 | ITU-T Recommendation T.81](#) disponible ici : <http://www.w3.org/Graphics/JPEG/itu-t81.pdf>. La norme ne se limite pas à notre spécification (mode baseline séquentiel uniquement), mais fondamentalement tout y est. Ce document restera votre livre de chevet préféré pour les 42 années à venir ;
2. <http://www.impulseadventure.com/photo> Ce site fournit une approche par l'exemple pour qui veut construire un décodeur JPEG baseline. On y retrouve des illustrations des différentes étapes présentées dans ce document. Les détails des tables de Huffman, la gestion du sous-échantillonnage, etc., sont expliqués avec des schémas et force détail, ce qui permet de ne pas galérer sur les aspects algorithmiques ;
3. Pour une compréhension plus poussée du sous-échantillonnage des chrominances, consulter <http://dougkerr.net/pumpkin/articles/Subsampling.pdf> ;
4. Pour les informations relatives au format JFIF, aller voir du côté de <http://www.ijg.org/>.

Parmi les informations que vous pourrez trouver sur le web, il y aura du code, mais il aura du mal à rentrer dans le moule que nous vous proposons. **L'examen du code lors de la soutenance sera sans pitié pour toute forme de plagiat.** Mais surtout assimiler ce type de code vous demandera au moins autant d'effort que de programmer vous-même votre propre décodeur !

1. Pour la gestion en C des paramètres optionnels d'un exécutable, renseignez-vous sur la famille des fonctions `getopt`. D'accord la `man page` est ce qu'elle est, mais entre comprendre cette doc et tout faire à la mimine, le choix devrait être simple... ↩
2. Vos enseignants sont parfois très joueurs! ↩
3. Au-delà de ce projet, ceci sera surtout valable dans votre vraie vie d'ingénieur, si si! Pensez à nous remercier le moment venu. ↩
4. Si ce n'est pas le cas encore, ça devrait ! ↩
5. Tip: moyennant une petite gymnastique sur les indices, cette représentation 1D simplifiera beaucoup l'expression et la manipulation des différentes fonctions à implémenter! ↩
6. Si ce n'est pas le cas, commencez par finir votre décodeur avant d'attaquer les extensions ! ↩
7. de pas trop près hein, on n'a toujours pas le droit... ↩
8. Vous verriez les performances du `jpeg2blabla` à côté de ce qu'on a en interne... Hein, Big Brother? ↩

Travail demandé

Objectif

L'objectif de ce projet est de développer un décodeur `jpeg2ppm` répondant aux spécifications édictées en section 4.1.

Même si une approche incrémentale, progressive, est fortement conseillée, il est réellement attendu que vous fournissiez un décodeur complet ! Il devra être capable de décoder toutes les images fournies, en supportant au minimum les sous-échantillonnages les plus répandus, cités en fin de section 2.5.1 (`4:4:4`, `4:2:2`, `4:2:0`).

Rendu

La date limite de rendu est fixée au **mercredi 24 mai 2023 à 12:00**. A cette heure, vos enseignants lanceront `git pull` sur la branche `master` de votre dépôt. Et c'est fini. **Tout ce que vous ajoutez après ne sera pas utilisé en soutenance.**

Votre dépôt devra contenir :

- un unique répertoire avec la même structure que le dépôt déployé initialement : sous-répertoires `include`, `src`, etc. Merci de respecter cette structure quel que soit l'environnement dans lequel vous ayez travaillé pendant le projet (IDE) ;
- la totalité de votre code source, fichiers `.h` et `.c` dans les bons sous-répertoires ;
- un fichier `Makefile` permettant de compiler l'application en tapant simplement `make` ;
- toutes les données de test supplémentaires que vous aurez pu utiliser (scripts, tests unitaires,...). N'ajoutez pas les fichiers image au dépôt, ce serait bien trop volumineux (sauf éventuellement des petits tests très particuliers). Votre banque d'image de tests sera de toute façon utilisée lors de la démonstration en soutenance. Vous pouvez si vous le souhaitez uploader vos images tests quelque part et indiquer où les télécharger dans un fichier `README.md` intégré au dépôt ;
- tout autre élément pertinent de votre projet.

Quelques remarques supplémentaires :

- logiquement, le code devrait fonctionner avec le même environnement que les salles machines de l'Ensimag (Linux 64 bits). Si vous n'avez pas moyen de garantir ceci¹, faites le point avec nous avant la fin du projet.

- il vous est demandé de respecter les [conventions de codage du noyau Linux](#) déjà utilisées dans la phase de préparation au C (exception pour les tabulations de largeur 4) ;
- utilisez également les types entiers C99 définis dans `stdint.h` et `stdbool.h` ;
- ne considérez pas le `Makefile` comme une contrainte mais comme une aide ! Le Makefile fourni devrait quasiment suffire pour tout le projet, à adapter bien sûr selon vos besoins. Utilisez-le dès le début et tout au long du projet. Le temps gagné est non négligeable, et les encadrants s'agaceront rapidement de devoir vous demander à chaque fois comment votre programme se compile si ce n'est pas fait. Et un encadrant agacé est un encadrant qui sera peu enclin à répondre à vos questions.
- il n'est PAS demandé de rapport écrit. Mais si vous avez rédigé des documents de travail ou schémas qui vous paraissent faciliter la compréhension de votre projet, n'hésitez pas à les joindre à l'archive rendue.

Soutenance

Les soutenances auront lieu les **jeudi 25 mai** et **vendredi 26 mai**. Un planning d'inscription sera mis en ligne sur `Teide`. Voici quelques éléments d'information, qui pourront être précisés si besoin d'ici la fin du projet.

- la durée de votre soutenance est de 40 minutes ;
- vous aurez 10-15 minutes pour nous présenter votre projet : ce qui fonctionne, les limites, comment vous avez conçu et testé votre code, les points importants/spécifiques de votre implémentation, etc. À vous de "vendre" votre travail de la manière qui vous paraît adéquate !
- la suite sera passée avec l'enseignant pour rentrer plus en détails dans votre projet, réaliser des tests supplémentaires, regarder le code, etc.
- si certains le souhaitent, vous pouvez préparer quelques transparents, schémas, etc. sur lesquels appuyer votre présentation. Mais ce n'est pas demandé et pas forcément utile, encore une fois c'est à vous de voir.
- l'évaluation sera faite à partir du travail rendu le mercredi. Toute modification ultérieure ne sera pas prise en compte.

Tout est dit, il ne nous reste plus qu'à vous souhaiter bon courage!

1. Pas de troll sur votre OS ; même si... ↩

Le format JPEG

Principe du format JFIF/JPEG

Le format d'un fichier JFIF/JPEG est basé sur des sections. Chaque section permet de représenter une partie du format. Afin de se repérer dans le flux JPEG, on utilise des marqueurs, ayant la forme `0xff??`, avec le `??` qui permet de distinguer les marqueurs entre eux. **La norme impose que les marqueurs soient toujours alignés dans le flux sur un multiple d'octets.**

Chaque section d'un flux JPEG a un rôle spécifique, et la plupart sont indispensables pour permettre le décodage de l'image. Nous vous donnons dans la suite de cette annexe une liste des marqueurs JPEG que vous pouvez rencontrer.

Petit point sur les indices et identifiants

Afin de faire les associations entre éléments, le JPEG utilise différents types d'indices. On en distingue trois :

- les **identifiants** des composantes de couleur, qu'on notera i_C ;
- les **indices** de table de Huffman, qui sont en fait la concaténation de deux indices $(i_{AC/DC}, i_H)$;
- et les **indices** de table de quantification, qu'on notera i_Q .

L'identifiant d'une composante est un entier entre 0 et 255. Les indices des tables sont eux des "vrais" indices : 0, 1, 2, etc. Une table de Huffman se repère par le type de coefficients qu'elle code, à savoir les constantes DC ou les coefficients fréquentiels AC, et par l'indice de la table dans ce type, i_H .

Afin de pouvoir décoder chaque composante de l'image, l'en-tête JPEG donne les informations nécessaires pour :

- associer une table de quantification i_Q à chaque i_C ;
- associer une table de Huffman $(i_{AC/DC}, i_H)$ pour chaque couple $(i_{AC/DC}, i_C)$.

Sections JPEG

Le format général d'une section JPEG est le suivant :

Offset	Taille (octets)	Description
0x00	2	Marqueur pour identifier la section
0x02	2	Longueur de la section en nombre d'octets, y compris les 2 octets codai
0x04	?	Données associées (dépendent de la section)

La longueur de la section indique combien d'octets on doit lire *au décodage*, une fois qu'on a lu le marqueur de section, pour lire l'intégralité de la section. Elle tient donc compte des deux octets permettant de coder cette longueur.

Marqueurs de début et de fin d'image

Toute image JPEG débute par un marqueur SOI (*Start of Image*) 0xffd8 et termine par un marqueur EOI (*End of Image*) 0xffd9. Ces deux marqueurs font exception dans le JPEG puisqu'ils ne suivent pas le format classique décrit ci-dessus : ils sont utilisés sans aucune autre information et servent de repères.

Bien qu'il soit possible qu'un fichier contienne plusieurs images (format de vidéo MJPEG, pour *Motion JPEG*), nous nous limiterons dans ce projet au cas d'une seule image par fichier.

APPx - *Application data*

Cette section permet d'enregistrer des informations propres à chaque *application*, application signifiant ici format d'encapsulation. Dans notre cas, on ne s'intéressera qu'au marqueur APP0, qui sert pour l'encapsulation JFIF. On ne s'intéresse pas aux différentes informations dans ce marqueur. Les seules choses qui nous intéressent sont la séquence des 4 premiers octets de la section, qui doit contenir la phrase JFIF, et le numéro de version JFIF X.Y codé sur deux octets (un pour X, un pour Y).

Offset	Taille (octets)	Description
0x00	2	Marqueur APP0 (0xffe0)
0x02	2	Longueur de la section
0x04	5	'J' 'F' 'I' 'F' '\0'
0x09	1	Version JFIF (1.1) : doit valoir 1
0x0A	1	Version JFIF (1.1) : doit valoir 1
0x0B	7	Données spécifiques au JFIF, non traitées : tout mettre à 0

COM - Commentaire

Afin de rajouter des informations textuelles supplémentaires, il est possible d'ajouter des sections de commentaires dans le fichier (par exemple, on trouve parfois le nom de l'encodeur). Notez que cela nuit à l'objectif de compression, les commentaires étant finalement des informations inutiles.

Offset	Taille (octets)	Description
0x00	2	Marqueur COM (0xfffe)
0x02	2	Longueur de la section
0x04	?	Données

DQT - *Define Quantization Table*

Cette section permet de définir une ou plusieurs tables de quantification. Il y a généralement plusieurs tables de quantification dans un fichier JPEG (souvent 2, au maximum 4). Ces tables sont repérées à l'aide de l'indice i_Q défini plus haut. C'est ce même indice, défini dans une section DQT, qui est utilisé dans la section SOF pour l'association avec une composante.

Un fichier JPEG peut contenir soit une seule section DQT avec plusieurs tables, soit plusieurs sections DQT avec une table à chaque fois. C'est la longueur d'une section qui permet de déterminer combien de tables elle contient. Si une table de quantification a le même indice i_Q qu'une table de quantification précédemment lue, alors cette table est redéfinie avec les nouvelles données lues.

Offset	Taille (octets)	Description
0x00	2	Marqueur DQT (0xffdb)
0x02	2	Longueur de la section
...	4 bits	Précision (0 : 8 bits, 1 : 16 bits)
	4 bits	Indice i_Q de la table de quantification
...	64	Valeurs de la table de quantification, stockées au format zig-zag (cf. sec

SOFx - *Start Of Frame*

Le marqueur SOF définit le début effectif d'une image, et donne les informations générales rattachées à cette image. Il existe plusieurs marqueurs SOF selon le type d'encodage JPEG utilisé. Dans le cadre de ce projet, nous ne nous intéressons qu'au JPEG *baseline sequential*, *DCT*, *Huffman*, *8 bits*, soit SOF0 (0xffc0). Pour information, les autres types sont récapitulés en section 6.3.

Les informations générales associées à une image sont la précision des données (le nombre de bits codant chaque coefficient, toujours 8 dans notre cas), les dimensions de l'image, et le nombre de composantes de couleur utilisées (1 en niveaux de gris, 3 en YCbCr).

Pour chacune de ces composantes sont définis :

- un identifiant i_C entre 0 et 255, qui sera référencé dans la section SOS ;
- les facteurs d'échantillonnage horizontal et vertical. Comme décrit section 2.5.1, ces facteurs h x v indiquent le *nombre de blocs par MCU* codant la composante (dans le cas de Y, il s'agit donc de la taille de la MCU) ;
- l'indice i_Q de la table de quantification associée à la composante.

Dans cette section SOF (et ce n'est garanti qu'ici), l'ordre des composantes est toujours le même : d'abord Y, puis Cb puis Cr. Les identifiants sont normalement fixés à 1, 2 et 3. Cependant, certains encodeurs ne suivent pas cette obligation et donc un décodeur doit savoir gérer des identifiants quelconques entre 0 et 255.

Finalement, une section SOF suit le format :

Offset	Taille (octets)	Description
0x00	2	Marqueur SOFx : 0xffc0 pour le SOF0
0x02	2	Longueur de la section
0x04	1	Précision en bits par composante, toujours 8 pour le baseline
0x05	2	Hauteur en pixels de l'image
0x07	2	Largeur en pixels de l'image
0x09	1	Nombre de composantes N (Ex : 3 pour le YCbCr, 1 pour les niveaux de
0x0a	3N	N fois : <ul style="list-style-type: none">• 1 octet : Identifiant de composante i_C, de 0 à 255• 4 bits : Facteur d'échantillonnage (<i>sampling factor</i>) horizontal, de 1 à 4• 4 bits : Facteur d'échantillonnage (<i>sampling factor</i>) vertical, de 1 à 4• 1 octet : Table de quantification i_Q associée, de 0 à 3

DHT - *Define Huffman Table*

La section DHT permet de définir une (ou plusieurs) table(s) de Huffman, selon le format décrit en section 2.8.1. Pour chaque table sont aussi définis ses indices de repérage $i_{AC/DC}$ et i_H .

Comme pour DQT, une section DHT peut contenir une ou plusieurs tables. Dans ce dernier cas, il y a en fait répétition des 3 dernières cases du tableau suivant. La longueur en octets de la section représente la taille nécessaire pour stocker toutes les tables contenues. Au décodage, pour déterminer si une section contient plusieurs tables, il faut donc regarder combien d'octets ont été lus pour construire une table. S'il en reste, la section contient encore (au moins) une autre table. A l'encodage, libre à vous, lorsque plusieurs tables de Huffman sont utilisées, de les définir dans autant de sections DHT, ou de les rassembler dans une seule section.

Dans un fichier, il ne peut pas y avoir plus de 4 tables de Huffman par type AC ou DC (sinon, le flux JPEG est corrompu).

Offset	Taille (octets)	Description
0x00	2	Marqueur DHT (0xffc4)
0x02	2	Longueur de la section
0x04		Informations sur la table de Huffman :
	3 bits	☐ non utilisés, doit valoir 0 (sinon erreur)
	1 bit	☐ type (0=DC, 1=AC)
	4 bits	☐ indice (0..3, ou erreur)
0x05	16	Nombres de symboles avec des codes de longueur 1 à 16. La somme de
0x15	?	Table contenant les symboles, triés par longueur (cf 2.8.1)

SOS - Start Of Scan

Une section SOS contient des données brutes encodant l'image. Un fichier séquentiel ne contient qu'une seule section SOS, mais un fichier progressif peut en contenir plusieurs.

L'en-tête d'une section SOS (on parle de *Scan Header*) contient:

- le nombre de composantes du *scan* ;
- les identifiants de(s) composante(s) dans l'ordre où elles apparaissent dans le flux ;
- les associations entre composantes et tables de Huffman ;
- les informations de sélection (fréquences) et d'approximation des coefficients AC pour le mode progressif. Pour plus d'informations rendez-vous en section B.2.3 (page 37) de la norme JPEG (document [itu-t81.pdf](#) distribué avec le sujet). Dans le cadre de ce projet, en mode *baseline sequential*, ces trois octets doivent prendre les valeurs respectives 0, 63 et 0.

Si un *scan* contient plusieurs composantes, elles sont entrelacées MCU par MCU. D'après la norme, l'**ordre** des composantes devrait être le même que dans la section SOF, soit Y, Cb puis Cr. Mais puisque certains encodeurs ne respectent pas cette convention, seul l'identifiant i_C d'une composante permet de l'associer correctement à la bonne composante de couleur. Les composantes apparaissent donc dans le flux dans l'ordre des indices i_C de cette section SOS.

Après le *Scan Header*, les données brutes sont ensuite stockées par blocs 8x8 encodés RLE + Huffman, dans l'ordre des composantes indiqué dans l'en-tête. Le nombre de blocs par composante de MCU dépend des facteurs d'échantillonnage lus en section SOF. Leur ordre est spécifié en 2.5.2.

Offset	Taille (octets)	Description
0x00	2	Marqueur SOS
0x02	2	Longueur de la section (données brutes non comprises)
0x04	1	N = Nombre de composantes. La longueur de la section vaut $2N + 6$
0x05	2N	N fois : <ul style="list-style-type: none"> 1 octet : identifiant i_C de la composante 4 bits : indice de la table de Huffman (i_H) pour les coefficients DC (4 bits : indice de la table de Huffman (i_H) pour les coefficients AC (i
...	1	Ss : Premier indice de la sélection spectrale : doit valoir 0 en mode bas
...	1	Se : Dernier indice de la sélection spectrale : doit valoir 63 mode basel
...	1	Approximation successive : <ul style="list-style-type: none"> Ah : 4 bits, poids fort : doit valoir 0 mode baseline Al : 4 bits, poids faible : doit valoir 0 en mode baseline

Récapitulatif des marqueurs

Code	Nom	Description
0x00		Byte stuffing (ce n'est pas un marqueur !)
0x01	TEM	
0x02 ... 0xbf	Réservés (not used)	
0xc0	SOF0	Baseline DCT (Huffman)
0xc1	SOF1	DCT séquentielle étendue (Huffman)

Code	Nom	Description
0xc2	SOF2	DCT Progressive (Huffman)
0xc3	SOF3	DCT spatiale sans perte (Huffman)
0xc4	DHT	Define Huffman Tables
0xc5	SOF5	DCT séquentielle différentielle (Huffman)
0xc6	SOF6	DCT séquentielle progressive (Huffman)
0xc7	SOF7	DCT différentielle spatiale (Huffman)
0xc8	JPG	Réservé pour les extensions du JPG
0xc9	SOF9	DCT séquentielle étendue (arithmétique)
0xca	SOF10	DCT progressive (arithmétique)
0xcb	SOF11	DCT spatiale (sans perte) (arithmétique)
0xcc	DAC	Information de conditionnement arithmétique
0xcd	SOF13	DCT Séquentielle Différentielle (arithmétique)
0xce	SOF14	DCT Différentielle Progressive (arithmétique)
0xcf	SOF15	Progressive sans pertes (arithmétique)
0xd0 ... 0xd7	RST0 ... RST7	Restart Interval Termination
0xd8	SOI	Start Of Image (Début de flux)
0xd9	EOI	End Of Image (Fin du flux)
0xda	SOS	Start Of Scan (Début de l'image compressée)
0xdb	DQT	Define Quantization tables
0xdc	DNL	
0xdd	DRI	Define Restart Interval
0xde	DHP	
0xdf	EXP	
0xe0 ... 0xef	APP0 ... APP15	Marqueur d'application
0xf0 ... 0xfd	JPG0 ... JPG13	
0xfe	COM	Commentaire

Exemple d'entête JPEG

Cette annexe décortique un entête de fichier jpeg, c'est à dire tous les paramètres et tables décrits en [annexe A](#) qui viennent avant l'image encodée proprement dite. Conseil: regardez en parallèle ces deux annexes A et B!

On utilise les outils présentés en [section outils](#) et sur l'image fournie `poupoupidou_bw.jpg`. En version "pop art/pixel art monochrome" (élargie ici pour voir quelque chose) cette image ressemble à ça:



En version "sont fada ces profs de C, et dire que dans 3 semaines je lirai moi aussi dans la Matrice", la trace de `hexdump` est:

```
$ hexdump -C poupoupidou_bw.jpg

00000000  ff d8 ff e0 00 10 4a 46  49 46 00 01 01 00 00 01  |.....JFIF.....|
00000010  00 01 00 00 ff fe 00 10  3c 33 20 6c 65 20 70 72  |.....<3 le pr|
00000020  6f 6a 65 74 20 43 ff db  00 43 00 03 02 02 02 02  |objet C...C.....|
00000030  02 03 02 02 02 03 03 03  03 04 06 04 04 04 04 04  |.....|
00000040  08 06 06 05 06 09 08 0a  0a 09 08 09 09 0a 0c 0f  |.....|
00000050  0c 0a 0b 0e 0b 09 09 0d  11 0d 0e 0f 10 10 11 10  |.....|
00000060  0a 0c 12 13 12 10 13 0f  10 10 10 ff c0 00 0b 08  |.....|
00000070  00 10 00 10 01 01 11 00  ff c4 00 15 00 01 01 00  |.....|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 07 00 ff  |.....|
00000090  c4 00 22 10 00 02 02 01  04 02 03 01 00 00 00 00  |.."......|
000000a0  00 00 00 00 01 03 02 04  05 06 11 12 21 07 13 00  |.....!...|
000000b0  14 31 33 ff da 00 08 01  01 00 00 3f 00 47 f0 7b  |.13.....?.G.{|
000000c0  31 16 f4 d0 c1 d7 7b 31  56 68 29 a3 35 5d 22 0a  |1.....{1Vh).5]".|
000000d0  75 87 cf d6 c4 3b db ca  73 87 14 15 9d 87 0f eb  |u....;.s.....|
000000e0  d8 00 ec 6f 39 54 f1 ce  63 0f 7f 30 9c 72 9d a9  |...o9T..c..0.r..|
```

(et ça continue...)

Analyse section par section

Dans un élan de bienveillance infinie, voici la même chose mais en surlignant les marqueurs des différentes sections:

```
$ hexdump -C poupoupidou_bw.jpg

00000000 ff d8 ff e0 00 10 4a 46 49 46 00 01 01 00 00 01 | .....JFIF.....|
00000010 00 01 00 00 ff fe 00 10 3c 33 20 6c 65 20 70 72 | .....<3 le pr|
00000020 6f 6a 65 74 20 43 ff db 00 43 00 03 02 02 02 02 | ojet C...C.....|
00000030 02 03 02 02 02 03 03 03 03 04 06 04 04 04 04 04 | .....|
00000040 08 06 06 05 06 09 08 0a 0a 09 08 09 09 0a 0c 0f | .....|
00000050 0c 0a 0b 0e 0b 09 09 0d 11 0d 0e 0f 10 10 11 10 | .....|
00000060 0a 0c 12 13 12 10 13 0f 10 10 10 ff c0 00 0b 08 | .....|
00000070 00 10 00 10 01 01 11 00 ff c4 00 15 00 01 01 00 | .....|
00000080 00 00 00 00 00 00 00 00 00 00 00 00 07 00 ff | .....|
00000090 c4 00 22 10 00 02 02 01 04 02 03 01 00 00 00 00 | ..".....|
000000a0 00 00 00 00 01 03 02 04 05 06 11 12 21 07 13 00 | .....!...|
000000b0 14 31 33 ff da 00 08 01 01 00 00 3f 00 47 f0 7b | .13.....?.G.{|
000000c0 31 16 f4 d0 c1 d7 7b 31 56 68 29 a3 35 5d 22 0a | 1.....{1Vh).5]".|
000000d0 75 87 cf d6 c4 3b db ca 73 87 14 15 9d 87 0f eb | u....;..s.....|
000000e0 d8 00 ec 6f 39 54 f1 ce 63 0f 7f 30 9c 72 9d a9 | ...o9T..c..0.r..|

(et ça continue...)
```

C'est déjà mieux non? Alors plus qu'à regarder en détail.

Prélude

Le premier marqueur est `ff d8`, c'est bien le *Start Of Image* (SOI).

APP0

Le second marqueur est `ff e0`, section *Application Data* (APP0) indiquant que l'image est de type JFIF. Les deux octets suivant le marqueur donnent la taille de la section, ici `00 10` (16) octets en comptant ces deux là. On peut vérifier que les cinq octets suivants sont bien `'J'`, `'F'`, `'I'`, `'F'`, `'\0'`, suivis de deux fois `'1'`, puis sept octets que l'on ignorera.

C'est conforme à la spécification vue en annexe [annexe A](#), donc on peut continuer.

Tout à fait Thierry

La prochaine section `ff fe` (COM) est un commentaire. La section est de taille 16 octets, donc vous trouverez quatorze octets à interpréter comme des caractères. Méditez ce commentaire.

Table de quantification

La section suivante `ff db` (DQT) définit une table de quantification. Sa taille est `00 43` soit 67 octets. Le premier octet après la taille est `00` : la précision est sur 8 bits et l'indice i_Q de la table est 0. Les 64 octets suivants `03 02 02 ...` sont les coefficients de cette table de quantification.

Ici on a bien lu la taille, un octet précision/indice et 64 valeurs. Ça fait 67 tout pile donc c'est fini. Tiens et si la taille de la section avait été plus grande? Allez, plus tard...

Start of Frame

La section suivante `ff c0` (SOF0) donne les informations relatives à l'image. Elle est de taille `00 0b` soit 11 octets. En continuant:

- le premier octet `08` indique une précision de 8 bits, ce qui est attendu en mode *baseline*.
- on trouve ensuite deux fois 2 octets pour la taille de l'image, ici 16x16 pixels.
- le prochain octet `01` indique qu'il n'y a qu'une seule composante, ce sera donc Y (monochrome).
- la table contient donc encore 3 octets: l'identifiant de la composante i_C (1), les facteurs d'échantillonnage (1x1), et l'indice de table de quantification associé à cette composante (0; ça tombe bien la table lue précédemment était d'indice $i_Q = 0$).

Table de Huffman

La prochaine section `ff c4` (DHT) définit une table de Huffman. La section est de taille `00 15` soit 21 octets. Le premier octet `00` indique une table pour une composante DC, d'indice 0. Les octets suivent le [format d'une table de Huffman JPEG](#):

- 16 octets donnant le nombre de codes de longueurs 1 à 16. Ici il n'y aura que deux codes de longueurs 1 et 2, respectivement.
- ensuite on lit les symboles eux-mêmes, ici 7 et 0.

Rebelote

Une autre section DHT? Pas de panique, vous savez la lire: AC, 15 symboles 1, 3, 2, 4, ..., 31, et 33.

SOS (*Start Of Scan*, pas Au Secours)

Ah voilà, la section suivante `ff da` (SOS) est celle qui va contenir les vraies données encodées. Elle est de taille 8, c'est bien $2N+6$ avec une seule composante.

Après la taille on trouve les octets `01 01 00 00 3f 00`. La spécification est super compliquée mais bon on retrouve qu'il n'y a bien qu'une seule composante dans ce scan et que son indice i_C est 1. Les deux derniers octets `3f 00` sont bien ceux attendus en mode *baseline*.

Et la suite? Et bien les octets restants `47 f0 7b ...` ne font plus partie de l'entête, c'est vraiment l'image encodée! Donc on sort du cadre de cette annexe, circulez.

This is The End

En réalité il reste encore un marqueur qu'on trouvera après avoir décodé l'image, c'est le `ff` `d9` (EOI) *End Of Image*.

Blablabla

Là nous venons de regarder le fichier binaire brut "à la dure" avec `hexdump`. Mais dans les outils nous vous avons aussi parlé de l'utilitaire `jpeg2blabla` codé avec soin et amour par nous-même. Il vous donnera les informations d'un entête de manière un peu plus interprétée, et aussi des détails sur les arbres de Huffman reconstruits.

En fait ces deux outils sont complémentaires, à vous de vous en servir à bon escient.

Et maintenant?

Cet exemple sur une image simple devrait être largement suffisant pour commencer. Ensuite ce sera à vous de savoir gérer les cas qui se présenteront pour des images plus générales.

Pour commencer ce projet, faites juste en sorte de lire les données d'entête correctement et de stocker les informations qui seront nécessaires pour la suite (même si vous ne les comprenez pas encore, ce qui viendra en grandissant).

Exemple d'encodage d'une MCU

Cette annexe donne un exemple d'encodage d'une MCU, de la compression des pixels RGB initiaux jusqu'à l'encodage bit par bit dans le flux de données JPEG. À étudier en parallèle du chapitre 2.

On travaille sur la version couleur de l'image étudiée dans l'[annexe sur l'entête](#) au format ppm, `poupoupidou.ppm` (élargie ici pour l'affichage):



C'est une image 16x16 que l'on souhaite encoder avec sous-échantillonnage horizontal (4:2:2 horizontal). On considère donc deux MCU de taille 16x8, et on s'intéresse ici à l'encodage de la première.

MCU en RGB

La première MCU en RGB est:

d83d67	d6407d	c0407b	d88170	fcc35c	e4b357	e4bf4c	e7c365	e6bd53	f4b25b	c45685	d83e87	d34378	d244
d5407a	c34b78	d06e6d	ffc745	dcbe44	f8dd5c	b68945	9e8437	e5d84b	f5d13d	fb06c	ce4980	cb4486	cb40
c73a7c	d78e65	ffc6b4	eec643	756335	a79435	e2bf47	ae9f40	eac866	f3d460	eacd43	ffcd52	f8c454	d885
c64271	934b56	e4c44d	cdae44	7f6849	dcc867	f1cc64	e5cd6b	eecf63	b7974c	f5df55	f9d65e	f7d557	ffce
c94574	b16a4e	ebd462	d5bf47	deadb0	e7b6af	e7b696	eeb3ab	f7b3c0	e9b7ac	e6bd9d	d3b764	d6be52	e7ca
ca4a63	82344e	8a6f42	be9b75	eaaba6	f0b6ab	edb5b6	f3adab	fba5b0	f6b5cb	e7bbb2	a29046	d4c66e	a190
c7437d	b6875d	7a693e	a77a81	e7acae	edb2b6	edb3a8	edadc5	efb0cb	e5b8b3	e7b2b8	ba965c	d4b753	b490
dc6470	b8a556	2b1c1f	b18386	ad7d8b	e2b1a3	f1afcb	cb9f96	bb828b	bd988f	a1767d	836751	bdab47	9a89

Représentation YCbCr

La représentation de cette MCU en YCbCr est:

[Y]:

70 74 6d 99 c8 b7 bd c3	bd bc 7c 74 74 74 77 72
73 74 8b c9 b9 d6 8f 83	cc cb bf 77 74 77 6d 76
6c 9f cf c3 63 8f bc 99	c7 d0 c6 ce c7 9c 7b 73
6f 62 c0 ab 6b c3 cb c9	cc 98 d6 d3 d1 d2 90 6f
72 7c ce b8 bc c4 c1 c4	c9 c5 c6 b6 b9 c5 82 7e
73 4e 72 a1 bd c6 c6 c2	c0 cb c7 8d c0 8b 4b 79
71 90 69 88 be c4 c3 c3	c6 c5 c3 9a b4 9c 66 8a
89 a2 21 91 8d be c6 ab	94 a2 84 6d a5 88 5b 8e

[Cb]:

7b 85 88 69 43 4a 40 4b	44 49 85 8b 82 7f 84 81
84 82 6f 36 3e 3b 56 55	37 30 51 85 8a 7e 81 87
89 5f 44 38 66 4d 3e 4e	49 41 36 3a 3f 6a 7c 8c
81 79 3f 46 6d 4c 46 4b	45 55 37 3e 3b 48 75 85
81 66 43 40 79 74 68 72	7b 72 69 52 46 41 63 88
77 80 65 67 73 71 77 73	77 80 74 58 52 51 7f 7f
87 63 68 7c 77 78 71 81	83 76 7a 5d 49 5b 6e 77
72 55 7f 7a 7f 71 83 74	7b 75 7c 70 4b 63 70 77

[Cr]:

ca c6 bb ad a5 a0 9c 9a	9d a8 b3 c7 c4 c3 d0 bf
c6 b8 b1 a7 99 98 9c 93	92 9e ab be be bc c8 bb
c1 a8 a2 9f 8d 91 9b 8f	99 99 9a a3 a3 ab c2 c6
be a3 9a 98 8e 92 9b 94	98 96 96 9b 9b a0 b4 c1
be a6 95 95 98 99 9b 9e	a1 9a 97 95 95 98 a7 bf
be a5 91 95 a0 9e 9c a3	aa 9f 97 8f 8e 90 9e c1
bd 9b 8c 96 9d 9d 9e 9e	9d 97 9a 97 97 91 91 b4
bb 90 87 97 97 9a 9f 97	9c 93 95 90 91 8d 8e b7

Sous échantillonnage

Cette MCU est sous-échantillonnée horizontalement, pour ne conserver qu'un seul bloc 8x8 par composante de chrominance.

[Y]:

70 74 6d 99 c8 b7 bd c3	bd bc 7c 74 74 74 77 72
73 74 8b c9 b9 d6 8f 83	cc cb bf 77 74 77 6d 76
6c 9f cf c3 63 8f bc 99	c7 d0 c6 ce c7 9c 7b 73
6f 62 c0 ab 6b c3 cb c9	cc 98 d6 d3 d1 d2 90 6f
72 7c ce b8 bc c4 c1 c4	c9 c5 c6 b6 b9 c5 82 7e
73 4e 72 a1 bd c6 c6 c2	c0 cb c7 8d c0 8b 4b 79
71 90 69 88 be c4 c3 c3	c6 c5 c3 9a b4 9c 66 8a
89 a2 21 91 8d be c6 ab	94 a2 84 6d a5 88 5b 8e

[Cb]:

80 78 46 45 46 88 80 82
83 52 3c 55 33 6b 84 84
74 3e 59 46 45 38 54 84
7d 42 5c 48 4d 3a 41 7d
73 41 76 6d 76 5d 43 75
7b 66 72 75 7b 66 51 7f
75 72 77 79 7c 6b 52 72
63 7c 78 7b 78 76 57 73

[Cr]:

c8 b4 a2 9b a2 bd c3 c7
bf ac 98 97 98 b4 bd c1
b4 a0 8f 95 99 9e a7 c4
b0 99 90 97 97 98 9d ba
b2 95 98 9c 9d 96 96 b3
b1 93 9f 9f a4 93 8f af
ac 91 9d 9e 9a 98 94 a2
a5 8f 98 9b 97 92 8f a2

DCT : passage au domaine fréquentiel

La DCT est ensuite appliquée à chacun des blocs de données, après avoir soustrait 128¹ aux valeurs. Les basses fréquences sont en haut à gauche et les hautes fréquences en bas à droite. On représente maintenant des *entiers signés 16 bits*. Ainsi, `0xffff` représente la valeur `-1` ici, et pas `65535` !

[Y]:

00f2 ff36 ffca 0008 fff2 001f 0000 ffd	00fb 00cb ffe1 0021 0004 ffc1 0027 000c
0005 0028 ffc b fcd fff6 0017 0023 000b	fffb 001a 001d 0000 ffec 0030 ffe8 fff5
ffda fff5 0015 005e 0027 ffca ffb b fddf	ff82 fff5 005a fff2 0000 ffe0 ffe4 0000
0018 ffc2 ffe0 0023 0031 ffde 0010 0030	ffe8 0018 0024 0000 fff6 fff0 0000 0000
000c ffe8 0022 0000 0000 0000 0000 0000	fff1 ffc8 fffa 0021 0000 0015 0000 fff1
fffc ffe4 002c ffe6 fff1 0000 0000 0011	0022 fff9 ffe0 000d 000f 0014 0000 0000
ffed 0000 001f ffe f 0000 0017 ffe8 0000	ffec 0000 fff0 0000 0014 ffe9 ffe8 0000
001b 0023 0000 ffd9 ffeb ffed ffeb 0013	001b 0000 0000 0000 0000 0000 ffec ffed

[Cb]:

ff2b fffc 003f fff8 0037 fffd 002c fffd
ffcd ffd9 005d 001c fff0 0023 fffb fff7
0034 fff1 0007 001d ffc b fff9 ffe2 ffd8
002c ffff fff1 0017 fff7 fff7 0000 fff5
0000 0003 fff6 0004 fff7 fff2 0008 0000
fff6 000f 000b fffd 0000 fff0 0000 0008
0000 0001 0000 fff7 0000 fff2 0000 0007
0000 0000 ffed fffb 0001 fffd 0007 000a

[Cr]:

0114 fff2 0046 0009 0029 0002 0014 fffe
003d ffe8 0029 0011 ffec ffff fff7 fff7
0016 0000 0000 000a ffec 0006 0000 ffff
0016 0007 fffa 0003 0000 0001 0005 0005
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 fffb 0008 fffa 0000 0003
0000 0000 0000 0000 0000 0000 0000 0000

Zig-zag

On réordonne ensuite les blocs en zig-zag. Cette étape permettra de regrouper les coefficients des plus haute-fréquences, souvent nuls après la phase de quantification, en "fin" de bloc. Pour des questions pratiques, on effectue cette réorganisation ZZ avant la quantification, puisque *les tables de quantification sont stockées au format zig-zag dans l'en-tête JPEG*. On pourrait bien entendu fusionner ces deux étapes (réordonnancement zig-zag + quantification) en une seule opération, comme indiqué sur la figure présentée en fin de section [2.1](#).

[Y]:

00f2 ff36 0005 ffda 0028 ffca 0008 ffc b	00fb 00cb fffb ff82 001a ffe1 0021 001d
fff5 0018 000c ffc2 0015 ffc d fff2 001f	fff5 ffe8 fff1 0018 005a 0000 0004 ffc1
fff6 005e ffe0 ffe8 fffc ffed ffe4 0022	ffec fff2 0024 ffc8 0022 ffec fff9 fffa
0023 0027 0017 0000 ffd d 0023 ffca 0031	0000 0000 0030 0027 000c ffe8 ffe0 fff6
0000 002c 0000 001b 0023 001f ffe6 0000	0021 ffe0 0000 001b 0000 fff0 000d 0000
ffde ffb b 000b ffd f 0010 0000 fff1 ffe f	fff0 ffe4 fff5 0000 0000 0015 000f 0000
0000 ffd9 0000 0000 0000 0030 0000 0000	0000 0000 0014 0014 0000 0000 fff1 0000
0017 ffeb ffed ffe8 0011 0000 ffeb 0013	ffe9 0000 0000 ffe8 0000 0000 ffec ffed

[Cb]:

ff2b fffc ffcd 0034 ffd9 003f fff8 005d
fff1 002c 0000 ffff 0007 001c 0037 fffd
fff0 001d fff1 0003 fff6 0000 000f fff6
0017 ffc b 0023 002c fffd fffb fff9 fff7
0004 000b 0001 0000 0000 0000 fffd fff7
fff7 ffe2 fff7 ffd8 0000 fff2 0000 fff7
ffed fffb 0000 fff0 0008 fff5 0000 0000
fff2 0001 fffd 0000 0008 0007 0007 000a

[Cr]:

0114 fff2 003d 0016 ffe8 0046 0009 0029
0000 0016 0000 0007 0000 0011 0029 0002
ffec 000a fffa 0000 0000 0000 0000 0000
0003 ffec ffff 0014 fffe fff7 0006 0000
0000 0000 0000 0000 0000 0000 0000 0000
0001 0000 fff7 ffff 0005 0000 0000 fffb
0000 0000 0008 0000 0000 0005 0000 0000
ffa 0000 0000 0000 0000 0003 0000 0000

Quantification

La quantification consiste à diviser les blocs par les tables de quantification, une pour la luminance et une pour les deux chrominances. Dans cet exemple, les deux tables sont issues du projet *The Gimp*. Lors du décodage, les tables utilisées sont bien sûr lues dans l'entête du fichier.

[table que quantification Y]

05	03	03	05	07	0c	0f	12
04	04	04	06	08	11	12	11
04	04	05	07	0c	11	15	11
04	05	07	09	0f	1a	18	13
05	07	0b	11	14	21	1f	17
07	0b	11	13	18	1f	22	1c
0f	13	17	1a	1f	24	24	1e
16	1c	1d	1d	22	1e	1f	1e

[table que quantification chrominances Cb/Cr]

05	05	07	0e	1e	1e	1e	1e
05	06	08	14	1e	1e	1e	1e
07	08	11	1e	1e	1e	1e	1e
0e	14	1e	1e	1e	1e	1e	1e
1e	1e	1e	1e	1e	1e	1e	1e
1e	1e	1e	1e	1e	1e	1e	1e
1e	1e	1e	1e	1e	1e	1e	1e
1e	1e	1e	1e	1e	1e	1e	1e

Après quantification, les blocs de la MCU sont finalement :

[Y]:

0030 ffb0 0001 fff9 0005 fffc 0000 fffe	0032 0043 ffff ffe7 0003 fffe 0002 0001
fffe 0006 0003 fff6 0002 fffd 0000 0001	fffe fffa fffd 0004 000b 0000 0000 fffd
fffe 0017 fffa fffd 0000 ffff ffff 0002	fffb fffd 0007 fff8 0002 ffff 0000 0000
0008 0007 0003 0000 fffe 0001 fffe 0002	0000 0000 0006 0004 0000 0000 ffff 0000
0000 0006 0000 0001 0001 0000 0000 0000	0006 fffc 0000 0001 0000 0000 0000 0000
fffc fffa 0000 ffff 0000 0000 0000 0000	fffe fffe 0000 0000 0000 0000 0000 0000
0000 fffe 0000 0000 0000 0001 0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
0001 0000 0000 0000 0000 0000 0000 0000	ffff 0000 0000 0000 0000 0000 0000 0000

[Cb]:

ffd6 0000 fff9 0003 ffff 0002 0000 0003
fffd 0007 0000 0000 0000 0000 0001 0000
fffe 0003 0000 0000 0000 0000 0000 0000
0001 fffe 0001 0001 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 ffff 0000 ffff 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000

[Cr]:

0037 fffe 0008 0001 0000 0002 0000 0001
0000 0003 0000 0000 0000 0000 0001 0000
fffe 0001 0000 0000 0000 0000 0000 0000
0000 ffff 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000

Codage différentiel DC

La composante continue DC est la première valeur d'un bloc.

Premier bloc

la MCU de cet exemple est la première de l'image (en haut à gauche), la valeur `0x0030` (= 48, l'âge du capitaine) doit être encodée en premier. Elle appartient à la classe de magnitude 6 :

`-63, ..., -32, 32, ..., 63`. Dans cette classe, l'indice de 48 est `110000`

Pour continuer il faut connaître les tables de Huffman à utiliser pour l'encodage, qui seront ensuite incluses dans l'entête et lues par le décodeur. On considère ici les tables "statistiques standard" définies dans la norme [ISO/IEC IS 10918-1 | ITU-T Recommendation T.81](#) (section K.3) et aussi accessibles via `./jpeg2blabla -vh poupoupidou.jpg`

Le code de Huffman de la magnitude 6 est `1110` (elle est portée par une feuille de profondeur 4). Finalement, la suite de bits à inclure dans le flux de l'image compressée est le code de la magnitude puis l'indice dans la classe de magnitude, soit ici : `1110110000`

Bloc suivant

La valeur DC du bloc suivant de la composante Y est `0x0032` (= 50). Par contre, la valeur à encoder est la différence par rapport à la valeur DC du bloc précédent (de la même composante de lumière), soit ici $50 - 48 = 2$. Avec `011` le code de Huffman de la magnitude 2, l'encodage de ce coefficient DC dans le flux de bits est `01110`.

Codage AC avec RLE

On s'intéresse au codage des 63 coefficients AC du bloc de Cr. La composante continue `0x0037` a déjà été encodée dans le flux. Il reste donc la séquence :

```
0xfffe 0x0008 0x0001 0x0000 0x0002 0x0000 0x0001 ... 0x0000
```

- Un premier symbole RLE sur un octet est calculé pour le premier coefficient `0xfffe` (-2, de magnitude 2 et d'indice 2):
 - les quatre bits de poids forts sont nuls, car aucun coefficient nul ne précède ce coefficient.
 - les quatre bits de poids faible contiennent la magnitude de -2, qui est 2. Le symbole RLE est donc `0x02` ;
 - on n'insère pas directement ce symbole dans le flux mais plutôt son code de Huffman, ici `100` (3 bits). Attention à bien lire dans le bon arbre, celui des AC / chrominances!
 - finalement l'indice du coefficient sera inséré, ici `10` (2 bits)
- Le prochain coefficient non nul est `0x0008`, de magnitude 4 et d'indice 8.
 - Aucun coefficient nul ne précède ce coefficient, donc le symbole RLE correspondant est `0x04` ;
 - le flux contiendra le code de Huffman du symbole, `11000` (5 bits), puis l'indice `1000` (4 bits).
- Le prochain coefficient non nul est `0x0001`, de magnitude 1 et d'indice 1.
 - Aucun coefficient nul ne précède ce coefficient, donc le symbole RLE correspondant est `0x01` ;
 - le flux contiendra le code de Huffman du symbole, `01` (2 bits), puis l'indice `1` (1 bits).
- Le prochain coefficient non nul est `0x0002`, de magnitude 2 et d'indice 2.
 - Un coefficient le précède, donc le symbole RLE correspondant est `0x12` ;
 - le flux contiendra le code de Huffman du symbole, `111001` (6 bits), puis l'indice `10` (2 bits).
- Le code des six prochains coefficients non nuls est:
 - `0x0001` -> symbole RLE `0x11`, encodé `1011` puis `1`

- `0x0003` -> symbole RLE `0x12`, encodé `111001` puis `11`
- `0x0001` -> symbole RLE `0x41`, encodé `111010` puis `1`
- `0xffffe` -> symbole RLE `0x12`, encodé `111001` puis `01`
- `0x0001` -> symbole RLE `0x01`, encodé `01` puis `1`
- `0xfffff` -> symbole RLE `0x71`, encodé `1111010` puis `0`
- Tous les coefficients suivants étant nuls, il suffit de mettre une balise EOB `0x00` pour terminer le codage du bloc; son code de Huffman est `00`.

Un gain de place?

Au final l'ensemble des 63 coefficients AC du bloc Cr est totalement encodé dans le flux par les 66 bits: `100 10 11000 1000 01 1 111001 10 1011 1 111001 11 111010 1 111001 01 01 1 1111010 0 00`.

Sans codage de Huffman, la séquence aurait été `00000002 10 00000004 1000 00000001 1 00000012 10 00000011 1 00000012 11 00000041 1 00000012 01 00000001 1 00000071 0 00`, soit 99 bits.

Et brutalement, sans RLE ni magnitude, il aurait fallu 63 octets soit 504 bits... Et nous n'avons parlé que du stockage des données compressées, mais par rapport à l'image PPM, il faut en plus considérer le gain lié à l'encodage du signal lui-même.

Y'a plus qu'à!

Voilà, ça pique un peu les yeux mais cette annexe devrait bien vous aider à comprendre l'encodage.

Ah oui, mais nous on doit faire un décodeur! Ok, ben faites dans l'autre sens alors.

Mais si en testant sur `poupoupidou.jpg` vous obtenez une image plus proche de `pompompidou.jpg`, ça vaut le coup de relire encore une fois cette annexe!



1. en fait 2^{P-1} , avec ici une précision $P = 8$ ↩

Pistes d'implémentations pour la DCT rapide

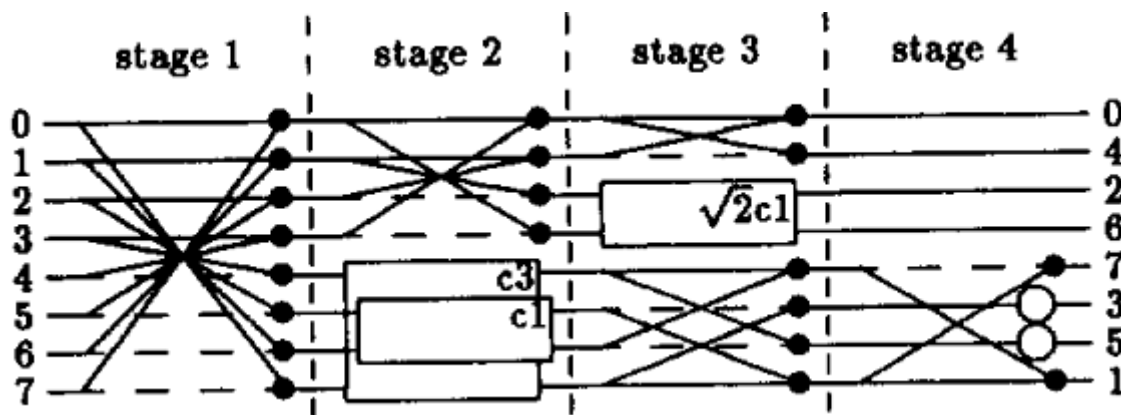
iDCT rapide : comment ça marche ?

L'article de référence sur l'algorithme de Loeffler : [Loeffler.pdf](#)

L'idée est toujours la même : on explique le sens encodage (DCT), à vous faire le chemin inverse comme des glands.

fast DCT

Le schéma qu'il est beau est le suivant :



Loeffler aurait pu être prof de C, car il a rajouté des coquilles dans son papier (!):

- en figure 1 : le coefficient de la rotation à l'étape 3 est: $\sqrt{2}c_6$ (et non $\sqrt{2}c_1$) ;
- en figure 2 : les sorties du butterfly et de la rotation sont sur O_0 (1ère ligne) et O_1 (2e ligne), et pas deux fois O_0 .

Au-delà de ça, il restes quelques points difficiles à comprendre :

- sens des papillons: la "borne inférieure" (O_1/I_1) est marquée d'un trait pointillé, ne pas se mélanger ;
- le schéma proposé code la DCT 1D à un facteur près, lequel ?

La définition de la DCT 1D¹ est:

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

avec $a_0 = \frac{1}{\sqrt{2}}$ et $a_i = 1$ si $i > 1$

Le schéma de Loeffler intègre uniquement $\sqrt{2}$ comme coefficient devant la somme. Ceci permet d'équilibrer le a_0 et de gagner une multiplication au final. Par contre tout le vecteur de sortie doit ensuite être normalisé pour retomber sur la définition de la DCT. Alors, normalisé par quel coefficient ?

- tout ce qui précède est en 1D. Mais la vie est bien faite, en 2D il suffit de faire la DCT 1D sur les lignes puis sur les colonnes.

Oui mais nous on veut la DCT inverse

Ben il suffit de tout faire dans l'autre sens! Donc inverser les opérations, coefficients...

Comme on est sympa, on vous donne la rotation inverse :

$$I_0 = O_0 \frac{1}{k} \cos\left(\frac{n\pi}{16}\right) - O_1 \frac{1}{k} \sin\left(\frac{n\pi}{16}\right)$$

$$I_1 = O_1 \frac{1}{k} \cos\left(\frac{n\pi}{16}\right) + O_0 \frac{1}{k} \sin\left(\frac{n\pi}{16}\right)$$

1. si ça vous dit la définition est ici : [DCT-II](#) ↩