Outils et programmation avancée

Préambule

Les codes sont réalisés en langage C++ et doivent être correctement indentés et commentés. Une documentation technique devra être générée (avec Doxygen par exemple).

Les prototypes de fonctions et de procédures sont données à titre indicatif, il est possible d'utiliser les classes de la STL ou d'OpenCV pour manipuler les données.

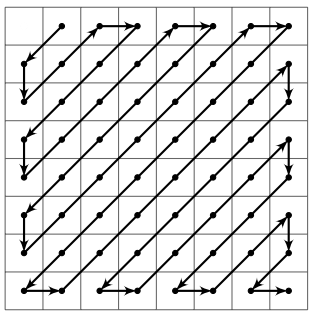
Principe général

JPEG (Joint Photographic Experts Group) est une norme définissant le format d'enregistrement et les algorithmes de codage/décodage pour une représentation numérique compressée d'une image fixe.

La norme communément appelée JPEG, de son vrai nom ISO/CEI 10918-1 UIT-T Recommendation T.81, est de base séquentiel avec pertes *(lossy compression)* et il peut être résumé très succinctement de la façon suivante (cf Figure 1) :

1. L'image subit une transformation de l'espace couleur
2. Un sous-échantillonnage peut être effectué sur les composantes de chrominance
3. L'image est divisée en blocs de 8×8 pixels auxquels on applique une DCT (Discrete Cosine Transform).

   Le processus de base suppose que les niveaux associés à chaque pixel sont codés sur 8 bits. Pour simplifier, on supposera ici que les images traitées sont données en niveaux de gris.
4. Les 64 coefficients de DCT obtenus sont quantifiés.
5. Les blocs sont lus ligne par ligne de haut en bas et de gauche à droite. La valeur moyenne d'un bloc (DC) est le coefficient dans le coin supérieur gauche. On soustrait alors à la valeur moyenne de tous les blocs sauf du premier, la valeur moyenne du bloc précédent.
6. Les 63 autres termes sont lus en « zig zag » selon la figure suivante :



7. La suite de valeur obtenue est codée (codage entropique de Huffman).
8. Chaque coefficient non nul est codé par le nombre de zéros qui le précède, le nombre de bits qui servent à le coder et sa valeur. Les règles de codage sont imposées par la norme.

Dans ce travail, nous nous intéresserons, dans un premier temps, à la compression d'une image de luminance (niveau de gris) puis, nous intégrerons la compression d'une image couleur. Les entêtes de fichiers ne seront pas gérées.

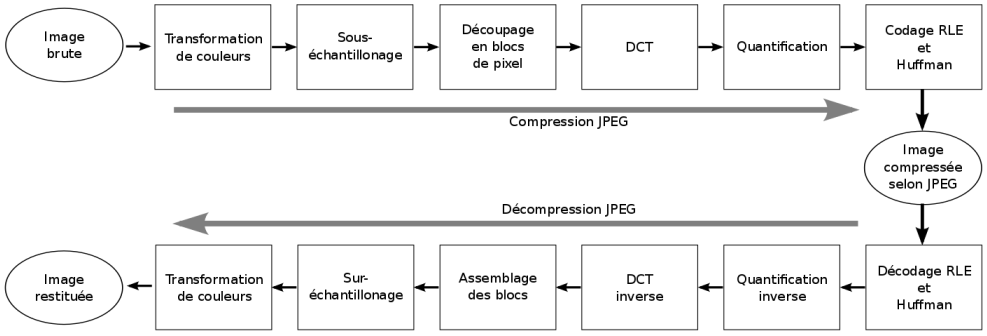


FIGURE 1 – Organigramme de compression (S. Brunner, CC BY-SA 3.0)

1. Calcul de la DCT

Soit p un tableau de pixels de taille 8×8. La DCT est alors donnée par :

$$P(u,v) = \frac{1}{4} C(u)C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} p(x,y) \cos \frac{(2x+1)\pi u}{16} \cos \frac{(2y+1)\pi v}{16},$$

pour tout $u, v \in \{0, \ldots, 7\}$ avec :

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si} \quad k = 0 \\ 1 & \text{si} \quad k = 1, \ldots, 7 \end{cases},$$

et la DCT inverse (IDCT) par :

$$p(x,y) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} P(u,v) C(u)C(v) \cos \frac{(2x+1)\pi u}{16} \cos \frac{(2y+1)\pi v}{16}.$$

On travaille sur des données codées sur un octet (valeur entre 0 et 255) qu'on ramènera entre -128 et 127 avant tout traitement (dans la norme, il est question de «décalage de niveau»).

Q. 1 Écrire une classe `cCompression` *avec les membres privés suivant :*
- `unsigned int mLargeur`, *la largeur de l'image,*
- `unsigned int mHauteur`, *la hauteur de l'image,*
- `unsigned char **mBuffer`, *l'image,*
- `unsigned int mQualite`, *la qualité de la compression (entre 0 et 100), mise à 50 par défaut.*

Q. 2 Écrire les mutateurs et assesseurs nécessaires pour les données membres.

Q. 3 Écrire la fonction membre de calcul de la DCT d'un bloc de taille 8×8 en C++ de prototype :

```
void Calcul_DCT_Block(char **Bloc8x8, double **DCT_Img);
```

Q. 4 Écrire la fonction membre de calcul de la DCT inverse de prototype :

```
void Calcul_iDCT(double **DCT_Img, char **Bloc8x8 );
```

Cette fonction sera nécessaire pour évaluer le taux de compression d'un bloc et pour vérifier la réversibilité de la DCT.

2. Quantification

Une fois la DCT calculée, on quantifie la matrice P selon la formule suivante :

$$P_q(u,v) = \text{round}\left(\frac{P(u,v)}{Q_{tab}(u,v)}\right).$$

La table de quantification Q_{tab} dépend d'une table de référence Q prévue pour quantifier la luminance dans la norme JPEG et d'un facteur de qualité F_q exprimé en pourcent selon la formule suivante :

$$Q_{tab}(i,j) = \begin{cases} 1 & \text{si} \quad \lfloor \frac{Q(i,j)\cdot\lambda+50}{100} \rfloor < 1 \\ 255 & \text{si} \quad \lfloor \frac{Q(i,j)\cdot\lambda+50}{100} \rfloor > 255 \\ \lfloor \frac{Q(i,j)\cdot\lambda+50}{100} \rfloor & \text{sinon} \end{cases},$$

avec

$$\lambda = \begin{cases} \frac{5000}{F_q} & \text{si} \quad F_q < 50 \\ 200 - 2F_q & \text{si} \quad F_q \geqslant 50 \end{cases}.$$

et

$$Q = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}.$$

On remarque qu'on a $Q_{tab} = Q$ pour $F_q = 50\%$.

Q. 5 Écrire une fonction membre :

```
void quant_JPEG(double **img_DCT, int** Img_Quant);
```

qui prend en argument une image après DCT `DCT_Img` *connaissant un facteur de qualité* `mQualite` *et calcule l'image quantifiée* `Img_Quant`.

Q. 6 Écrire une fonction :

```
void DCT_Img = dequant_JPEG(int** Img_Quant, double **img_DCT);
```

qui prend en argument une image quantifiée connaissant un facteur de qualité `mQualite` *et calcule l'image déquantifiée.*

Q. 7 Écrire une fonction membre :

```
double EQM(int **Bloc8x8);
```

qui prend en argument une image `Bloc8x8` *connaissant un facteur de qualité* `mQualite` *et renvoie l'écart quadratique moyen calculé entre l'image approchée par l'étape de quantification et l'image d'origine .*

Q. 8 Écrire une fonction membre :

```
double Taux_Compression(int **Bloc8x8);
```

qui prend en argument une image `Bloc8x8` *connaissant un facteur de qualité* `mQualite` *et renvoie le taux de compression calculée entre le bloc issu de l'étape de quantification et l'image d'origine*[1].

Un exemple vous est donné dans l'annexe A.

3. Run-Length Encoding

Il reste maintenant à réaliser la dernière étape avant le codage entropique à savoir l'encodage « Run-Length Encoding » qui consiste à rassembler en paquets les symboles d'un signal qui, tout à la fois se suivent et sont identiques. Pour mémoire dans le cadre de JPEG, pour chaque coefficient, on code le couple longueur de plage de zéros et amplitude.

Un exemple vous est donné dans l'annexe B.

Q. 9 Écrire une fonction membre :

```
void RLE_Block(int** Img_Quant, int DC_precedent, char *Trame);
```

qui prend en argument un bloc quantifié `Img_Quant`*, la valeur moyenne du bloc précédent* `int DC_precedent` *et calcule la trame* `Trame`.

Q. 10 Écrire une fonction membre

```
void RLE(int *Trame);
```

qui concatène des trames de tous les blocs 8×8 *à partir d'une image* `mBuffer` *de hauteur et largeur multiple de 8.*

1. On supposera que la taille du bloc après les étapes de DCT et de quantification sera évaluée par le nombre de coefficients non nuls dans ce bloc quantifié.

4. Codage de Huffman

L'objectif de cette étape est d'implanter le codage de Huffman et de proposer un objet pour manipuler les noeuds d'un arbre binaire adaptée à ce codage. Vous implanterez l'encodage de la trame après le RLE.

4.1. Préambule

Soit les fréquences d'apparition des lettres de 'a' à 'f' dans le tableau 1,

	a	b	c	d	e	f
Occurences	5	3	7	1	10	2

TABLE 1 – Table des occurences

Avant de travailler sur la trame issue du RLE, vous debuggerez votre code à partir de cet exemple.

Q. 11 A partir du tableau 1, quel est l'arbre binaire que vous devriez théoriquement obtenir lors d'un codage de Huffman ?

Q. 12 Quels sont les codes binaires des six lettres ?

4.2. Construction de l'arbre

```
struct sNoeud {
  char mdonnee;     // code
  double mfreq;     // frequence du code
  sNoeud *mgauche; // fils gauche
  sNoeud *mdroit;  // fils droit

  sNoeud(char d, double f) {
    mgauche = NULL;
    mdroit = NULL;
    this->mdonnee = d;
    this->mfreq = f;
  }
};
```

Q. 13 Écrire une classe `cHuffman` contenant
- *un pointeur sur une trame `mtrame` de type `char`,*
- *de longueur `mLongueur`*
- *l'adresse de la racine du noeud de l'arbre du codage de Huffman.*

Q. 14 Écrire ses constructeurs, son destructeur, ses assesseurs et mutateurs nécessaires.

Q. 15 Écrire une fonction membre qui construit l'arbre de codage à partir d'un vecteur de données `Donnee` de type `char`, de ses fréquences `Frequence` de type `double` et de leur longueur `Taille` :

```
void HuffmanCodes(char *Donnee, double *Frequence, unsigned int Taille);
```

Pour cela, nous utiliserons une file de priorité, une structure de données qui permet de retourner la plus petite valeur d'un vecteur de type `vector`. La STL en propose une sous le nom `std::priority_queue`.

```
priority_queue<noeud*, vector<noeud*>, compare> queue;
```

Le code de comparaison des fréquences de 2 noeuds est le suivant :

```
struct compare {
  bool operator()(noeud *gauche, noeud *droit) {
    return gauche->s_freq > droit->s_freq;}
};
```

La file est chargée avec `queue.push();` [2]
L'élément en queue est extrait par `queue.top();`
L'élément en queue est supprimé par `queue.pop();`

4.3. Affichage des codes binaires

Implanter une fonction membre qui affiche le codage de l'arbre construit par le codage de Huffman :

```
void AfficherHuffman(struct sNoeud *Racine)
```

Tester votre code avec les fréquences du tableau 1.

4.4. Compression de l'image

Q. 16 Écrire une fonction membre privée de la classe `cCompression` qui calcule l'histogramme de la trame après l'étape du RLE :

```
unsigned int Histogramme(char* Trame, unsigned int Longueur_Trame, char *Donnee, double *
    Frequence);
```

La taille des vecteurs `Donnee` et `Frequence` est retournée en sortie.

*Q. 17 **[HARD]** Écrire une fonction membre publique de la classe `cCompression` qui enregistre sur le disque dur le flot de bits des codes de la trame issues du codage du Huffman :*

2. `http://www.cplusplus.com/reference/queue/priority_queue/`

```
void Compression_JPEG(char*Trame, char *Nom_Fichier);
```

Cette fonction calcule l'histogramme des données de la trame après l'étape de RLE, puis effectue le codage de Huffman et enfin enregistre le flot de données. La difficulté est de bien gérée le flot en fonction de la longueur des codes.

Q. 18 Ecrire une fonction pour charger l'image `Lenna.dat` *de taille* 256×256 *et faire un graphique du taux de compression en fonction de la qualité.*

5. Fonction de décompression

Q. 19 Écrire une fonction membre publique de décompression `Decompression()` *qui prend en argument une suite de trames et un facteur de qualité et renvoie l'image décompressée. On suppose que l'image de départ est carrée pour pouvoir la reconstruire.*

```
char** Decompression_JPEG(char *Nom_Fichier_compresse);
```

On considère la table du codage de Huffman connue (pas de gestion des entêtes).

Q. 20 Tester votre algorithme de compression-décompression JPEG sur l'image `lenna.img` *pour différentes valeurs de qualités.*

6. [OPTION] Compression d'une image couleur

Pour compresser une image couleur, on privilégie une représentation de type luminance/chrominance. L'algorithme convertit l'image d'origine codé dans l'espace colorimétrique RVB (rouge, vert, bleu) vers le modèle de type YCbCr avec Y l'information de luminance, Cb et Cr sont deux informations de chrominance. L'œil humain étant peu sensible à la chrominance, les informations Cr et Cb peuvent être sous-échantillonnées (les blocs de luminance ne sont pas modifiés). L

Pour cela, est utilisée la notation *J :a :b* :

— *J* représente la largeur de la plus petite matrice de pixels considérée (généralement 4)
— *a* représente le nombre de composantes de chrominance dans la première ligne
— *b* représente le nombre de composantes de chrominance supplémentaires dans la deuxième ligne

Q. 21 Écrire des classes `cCompressionCouleur` *et* `cDecompressionCouleur` *qui héritent respectivement des classes* `cCompression` *et* `cDecompression` *qui permettent de compresser et décompresser des images couleurs.*

Q. 22 La classe intègrera le fonctions membres de changement d'espace colorimétrique RVB vers YCrCb et les sous-échantillonnages 4 :2 :0 et 4 :4 :4 (pas de sous-échantillonnage) des matrices de chrominances de la Figure 2.

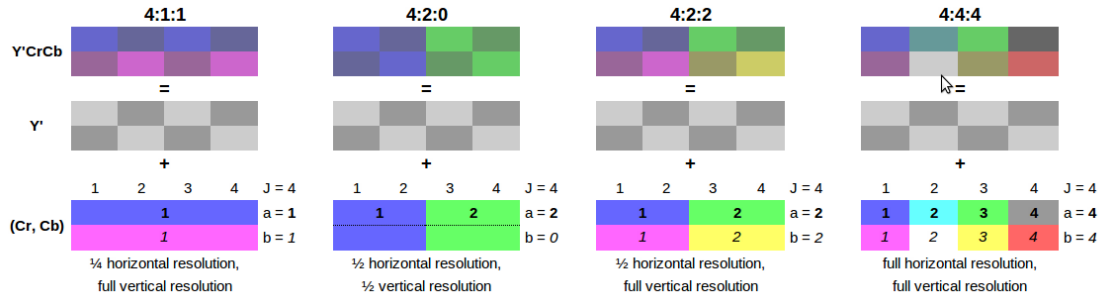


FIGURE 2 – Sous-échantillonnage de la chrominance (CC BY-SA 3.0)

A. Exemple

Si un bloc 8×8 extrait d'une image prend la forme suivante :

$$p = \begin{pmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 156 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 162 & 162 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 162 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{pmatrix}.$$

En ramenant les valeurs entre -128 et 127, on a :

$$p = \begin{pmatrix} 11 & 16 & 21 & 25 & 27 & 27 & 27 & 27 \\ 16 & 23 & 25 & 28 & 31 & 28 & 28 & 28 \\ 22 & 27 & 32 & 35 & 30 & 28 & 28 & 28 \\ 31 & 33 & 34 & 32 & 32 & 31 & 31 & 31 \\ 31 & 32 & 33 & 34 & 34 & 27 & 27 & 27 \\ 33 & 33 & 33 & 33 & 32 & 29 & 29 & 29 \\ 34 & 34 & 33 & 35 & 34 & 29 & 29 & 29 \\ 34 & 34 & 33 & 33 & 35 & 30 & 30 & 30 \end{pmatrix}.$$

On applique ensuite une DCT :

$$P = \begin{pmatrix} 235.62 & -1.03 & -12.08 & -5.20 & 2.13 & -1.67 & -2.71 & 1.32 \\ -22.59 & -17.48 & -6.24 & -3.16 & -2.86 & -0.07 & 0.43 & -1.19 \\ -10.95 & -9.26 & -1.58 & 1.53 & 0.20 & -0.94 & -0.57 & -0.06 \\ -7.08 & -1.91 & 0.22 & 1.45 & 0.90 & -0.08 & -0.04 & 0.33 \\ -0.62 & -0.84 & 1.47 & 1.56 & -0.13 & -0.66 & 0.61 & 1.28 \\ 1.75 & -0.20 & 1.62 & -0.34 & -0.78 & 1.48 & 1.04 & -0.99 \\ -1.28 & -0.36 & -0.32 & -1.46 & -0.49 & 1.73 & 1.08 & -0.76 \\ -2.60 & 1.55 & -3.76 & -1.84 & 1.87 & 1.21 & -0.57 & -0.45 \end{pmatrix}.$$

Puis on quantifie avec $F_q = 50\%$:

$$P_q = \begin{pmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

On passe donc de 64 coefficients à 7 coefficients significatifs, d'où une compression de 89%. En déquantifiant, on trouve :

$$P' = \begin{pmatrix} 240 & 0 & -10 & 0 & 0 & 0 & 0 & 0 \\ -24 & -12 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & -13 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

En appliquant la DCT inverse, en arrondissant puis en ramenant entre 0 et 255 :

$$p' = \begin{pmatrix} 142 & 144 & 147 & 150 & 152 & 153 & 154 & 154 \\ 149 & 150 & 153 & 155 & 156 & 157 & 156 & 156 \\ 157 & 158 & 159 & 161 & 161 & 160 & 159 & 158 \\ 162 & 162 & 163 & 163 & 162 & 160 & 158 & 157 \\ 162 & 162 & 162 & 162 & 161 & 158 & 156 & 155 \\ 160 & 161 & 161 & 161 & 160 & 158 & 156 & 154 \\ 160 & 160 & 161 & 162 & 161 & 160 & 158 & 157 \\ 160 & 161 & 163 & 164 & 164 & 163 & 161 & 160 \end{pmatrix}.$$

On trouve alors un écart quadratique moyen d'environ 5.

B. Exemple du RLE

Séquence	**Commentaires**
15	Différence avec moyenne bloc précédent (ici 0)
1,−2	1 zéro précède le -2
0,−1	Aucun zéro ne précède le -1
0,−1	Aucun zéro ne précède le -1
0,−1	Aucun zéro ne précède le -1
2,−1	2 zéros précédant le -1
0,−1	Aucun zéro ne précède le -1
0,0	Il ne reste plus que des zéros

Le bloc 8×8 vu comme un tableau comprenant 64 valeurs se réduit à la trame :

```
[15, 1, -2, 0, -1, 0, -1, 0, -1, 2, -1, 0, -1, 0, 0].
```