

## Format de compression d'image numérique

### Codage hiérarchique par *QuadTree*

Le point de départ de ce projet est la représentation d'une image en niveaux de gris sous la forme d'un quadtree hiérarchique. On utilisera cette structure pour réaliser un schéma de codage compressif avec ou sans perte en partant de l'observation que certaines images présentent des zones uniformes (ou presque).

## Préambule - les images numériques en données brutes (non compressées)

Il existe un grand nombre de formats de fichiers pour stocker les images numériques (GIF, PNG, JPEG, pour les plus connus). La plupart correspondent à des données compressées et sont structurés de manière précise et complexe. Certains formats permettent néanmoins de manipuler les données brutes de l'image (directement les valeurs des pixels, sans aucune compression).

Sous environnement Linux, le format brut standard est le PNM<sup>(1)</sup> (*Portable aNy Map* aussi appelé *Netpbm*).

### Les formats PNM

Le format PNM se décline en plusieurs sous-formats selon la nature des données :

- PBM (*Portable Bit Map*) pour les images binaires, en données brutes ou ASCII
- PGM (*Portable Gray Map*) pour les images en niveaux de gris, en données brutes ou ASCII
- PPM (*Portable Pix Map*) pour les images en couleur RGB, en données brutes ou ASCII

Dans tous les cas, l'image est représentée par une matrice de pixels (ou pixmap) à  $nbl$  lignes et  $nbc$  colonnes, chaque pixel étant représenté par son intensité : un entier codé sur  $n$  bits pour une image sur  $nbg = 2^n$  niveaux de gris ou un vecteur à trois composantes entières (R,V,B, par exemple), chacune étant codée sur  $n$  bits, pour une image couleur.

☞ On ne s'intéresse ici qu'aux images en niveaux de gris (format PGM) dont la structure est :

- un code (*magic number*), 2 caractères ASCII sur une ligne, indiquant le format<sup>(2)</sup> : P2 si les données sont écrites en ASCII, P5 si elles sont brutes (écrites en binaire, *raw*).
- une ou plusieurs lignes de commentaires, commençant toujours par le caractère '#'.  
#
- trois entiers ASCII écrits sur une ou plusieurs lignes, représentant dans l'ordre les nombres de colonnes  $nbc$ , de lignes  $nbl$  et de niveaux de gris  $nbg$  (en général 255, mais pas toujours).
- enfin viennent les données, écrites pixel après pixel, en ASCII ou en binaire selon le code d'en-tête, sans indication de changement de ligne.
  - en mode *raw*, un pixel est codé sur 8 bits. La taille totale des données est donc  $(nbc * nbl)$  octets.
  - en données ASCII, un pixel est représenté par 1, 2 ou 3 caractères suivi(s) d'un espace. La taille totale des données est donc comprise entre  $(2 * nbc * nbl)$  et  $(4 * nbc * nbl)$ , octets.

<sup>(1)</sup> Linux propose nativement ce format – cf. `$>man pnm` et <https://linux.die.net/man/3/libpnm>

<sup>(2)</sup> Les autres codes sont P1 ou P4 pour le format PBM, et P3 ou P6 pour le format PPM

## Exemple

début de fichier en données brutes

```
P5
# CREATOR: XV Version 3.10a
512 512 255
Å70VÅW>"FMZÃÃiIÃÃ²uOYGXc@EwvG7Ã...
```

début de fichier en données ASCII

```
P2
### commentaire
512 512
255
162 162 162 161 162 156 163 160 164 160 161 ...
```

Les objectifs de ce projet sont de réorganiser ces pixels sous une forme différente (hiérarchique) et tirer parti de certaines *corrélations* propres aux images pour tenter de compresser les données.

Dans cette optique, seuls les fichiers en données binaires (code P5) ont un intérêt et pour simplifier les choses, les images à traiter seront toujours de taille  $(2^n \times 2^n)$ , sur 256 niveaux de gris.

Il est par ailleurs *vivement* conseillé de développer les phases de codage et décodage en parallèle !! La plupart des algos. et fonctions à écrire sont en effet parfaitement symétriques et l'ensemble codeur/decodeur (ou codec) se réduit à un assemblage de «boîtes» réversibles.

☞ au final c'est 1 seul et même exécutable (éventuellement dédoublé via un lien symbolique) qui sera soit codeur soit décodeur

## Décomposition en quadtree - réorganisation des données

Une image est une zone bidimensionnelle formée de **pixels** (carrés insécables). Lorsque que 4 pixels voisins sont strictement identiques on peut les grouper et considérer simplement que l'on a affaire à un pixel 'plus gros'.

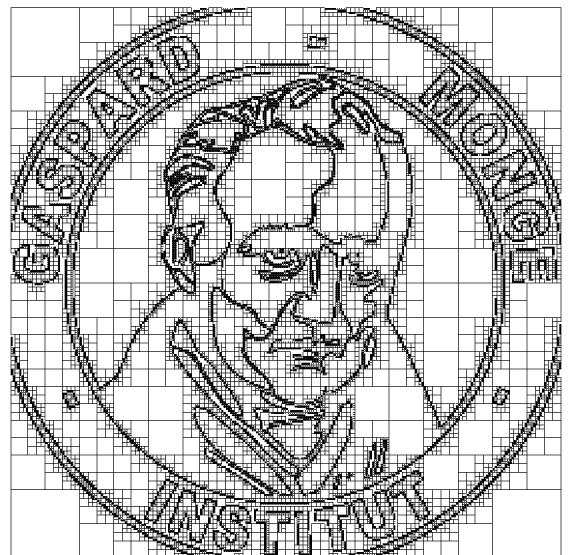
Ainsi une grande zone uniforme pourra être traitée comme un seul très gros pixel, comme dans l'exemple ci-contre.

☞ dans ce genre d'application, le **quadtree** est utilisé comme une structure *réductrice* pour la **compression**, mais les applications sont nombreuses et variées : recherche de formes (*pattern*), extraction de contours, détection de mouvement ...

Bien sûr, le cas ci-contre est **très** simple : c'est un logo, facilement réductible à une image binaire (blanc/noir)

☞ notre *compresseur* sera très performant.

Sur une image photographique ce sera nettement moins spectaculaire et des traitements annexes seront nécessaires.



### Cas général d'une image en niveaux de gris

Le principe de base de la représentation en **quadtree** est donc de subdiviser récursivement le *pixmap* en quatre et d'associer à chaque nœud la moyenne des intensités de ses 4 fils.

Ainsi, la racine de l'arbre représente l'intensité moyenne sur *toute* l'image et les feuilles représentent directement les pixels, comme le montre l'exemple suivant pour une image  $4 \times 4$

### Structure du quadtree

- Pour une image de taille  $(2^n \times 2^n)$ , le **quadtree** aura  $(n + 1)$  niveaux, chaque niveau  $(0 \leq k \leq n)$  pouvant être vu comme un *pixmap* de taille  $(2^k \times 2^k) = 4^k$ , réorganisé, et dont les «pixels» sont des nœuds du **quadtree**.

☞ la taille totale du **quadtree** est donc parfaitement connue dès le départ :  $\sum_{k=0}^{n} 4^k$  nœuds.

- Le **quadtree** sera construit de manière *ascendante* (parcours *suffixe*), en commençant par le niveau  $n$ , qui contient les valeurs des pixels de l'image, réorganisés par groupes de 4 voisins, comme sur la *fig.??*.

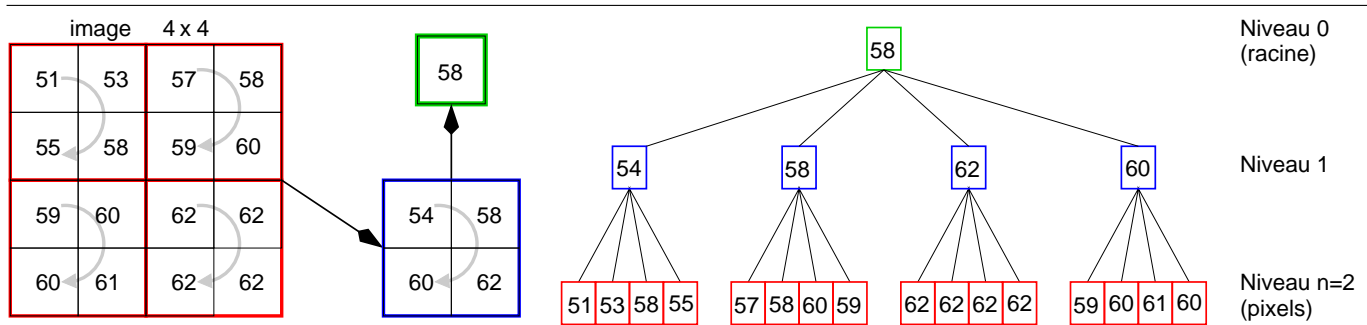


Figure 1: Exemple de décomposition en quadtree pour une image (4x4)

Le sens de lecture des 4 voisins (ici, sens des aiguilles d'une montre) est arbitraire mais **imposé** (le même à chaque niveau, pour le codeur et le décodeur)  
 ➡ il s'agit de définir un **format d'échange de fichiers** : n'importe quel décodeur doit pouvoir décompresser un fichier produit par n'importe quel encodeur.

- Un nœud du quadtree est défini comme un « pixel surchargé » : une structure contenant la valeur moyenne  $m$  (`unsigned char`) de ses quatre fils, ainsi que d'autres champs utiles ( $\epsilon$ ,  $u$ , ...), détaillés dans la suite, plus ce qu'il faut pour naviguer dans l'arbre.

L'architecture globale du quadtree est laissée à votre appréciation : gestion de la mémoire, accès aux nœuds fils/frères/père ....

Quelques conseils :

- la taille du quadtree (nbre et taille des niveaux) est connue dès le départ et il faudra nécessairement créer et remplir le quadtree en intégralité (des feuilles, jusqu'à la racine).
- une fois créé, le quadtree ne change pas de forme (pas de nœud créé, supprimé ou déplacé en cours de route) et les nœuds sont parfaitement et simplement ordonnés (les 4 fils d'un nœud de niveau ( $k$ ) sont toujours voisins dans le niveau ( $k+1$ )) et leur position dans ce niveau se calcule facilement par rapport à la position du nœud père au niveau ( $k$ ).

Et idem dans l'autre sens, d'un nœud vers son père.

➡ hors de question d'allouer chaque nœud individuellement avec des pointeurs partout !

Un seul « gros » `malloc` peut suffire (à la limite, un par niveau, mais pas plus).

➡ au final, ce quadtree n'est rien d'autre qu'un "tas d'orde 4" constant (cf. cours Algo des Arbres en L2).

## Informations complémentaires sur les nœuds

On voit bien qu'avec cette structure de quadtree, on est loin d'avoir compressé quoi que ce soit : si une image de taille ( $2^n \times 2^n$ ) « pèse »  $4^n$  octets, les données de *moyennes* du quadtree pèsent à elles seules  $\sum_{k=0}^n 4^k$  octets.

Le schéma de compression proposé ici repose sur 3 critères :

### a) interpolation au décodage :

la valeur d'un nœud interne (pas une feuille) représentant la moyenne de ses quatre fils, on peut se contenter de ne coder que trois de ses fils, le dernier pouvant être, en théorie, reconstruit par interpolation :  $m = \frac{m_1 + m_2 + m_3 + m_4}{4} \Rightarrow m_4 = 4m - (m_1 + m_2 + m_3)$ .

Le nombre de valeurs à stocker est alors  $\left(1 + \sum_{k=1}^n (4^k - 4^{k-1})\right) = 4^n$  comme pour l'image brute.

⑥ la valeur d'erreur  $\epsilon$  :

le fait de travailler avec des valeurs entières va provoquer des erreurs d'arrondi lors des calculs de moyennes (division entière). Ainsi, à la reconstruction, la valeur *interpolée* au niveau  $k$  risque d'être faussée et cette erreur se propagera aux niveaux  $p > k$

Si l'on veut assurer une compression sans perte (le fichier après décodage est *exactement* le même que l'original), il faut prendre en compte ces erreurs pour ne pas les répercuter au décodage sur l'interpolation de la valeur du quatrième fils.

Il faut donc ajouter aux nœuds internes du quadtree un champ  $\epsilon$  pour encoder et transmettre cette erreur.

Celle-ci est facilement quantifiable : on note  $s = (m_1 + m_2 + m_3 + m_4)$  la somme des 4 valeurs

alors  $m = s // 4$  (div. entière) et  $\epsilon = s \% 4$  (modulo) et donc  $\epsilon \in \{0, 1, 2, 3\}$  ☞  $\epsilon$  est codable sur 2bits .

☞ La valeur interpolée pour le 4<sup>o</sup> fils du nœud courant devient **exacte** :  $m_4 = (4m + \epsilon) - (m_1 + m_2 + m_3)$

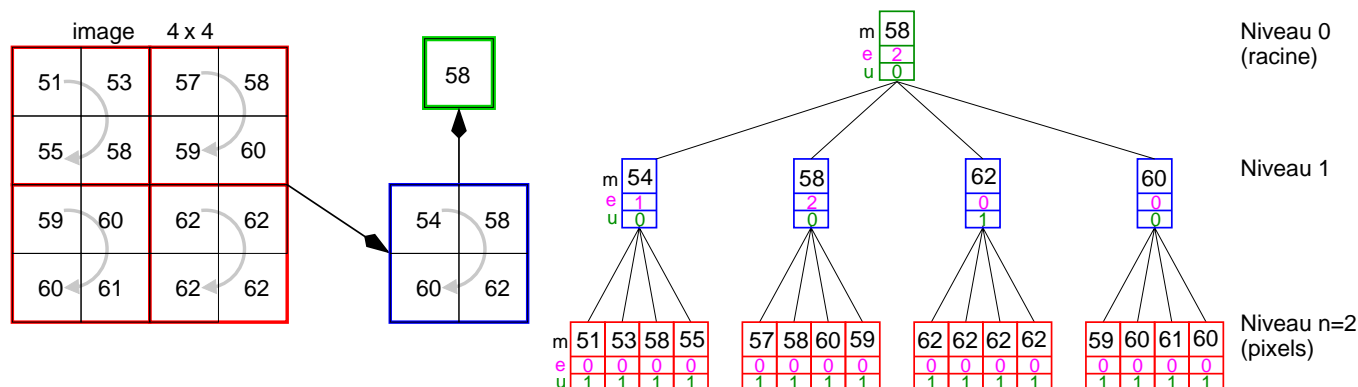
⑦ le bit d'uniformité  $u$  :

lorsqu'un bloc est uniforme à un niveau  $k$ , il l'est à tous les niveaux  $p > k$ . Ainsi, si un nœud représente un bloc uniforme, le sous-arbre dont il est la racine ne contient aucune information supplémentaire : il pourra être éliminé au moment du codage.

Il faudra donc prévoir, pour chaque nœud autre qu'une feuille, un champ  $u$  codable sur 1 bit , précisant si le bloc représenté est uniforme ou non.

Le processus de compression n'aura donc pas besoin de descendre dans le sous-arbre correspondant, et lors de la décompression, on reconstruira directement un bloc uniforme à la bonne taille.

Avec l'image 4x4 précédente, la structure de quadtree obtenue serait donc telle que :



Quelques remarques importantes

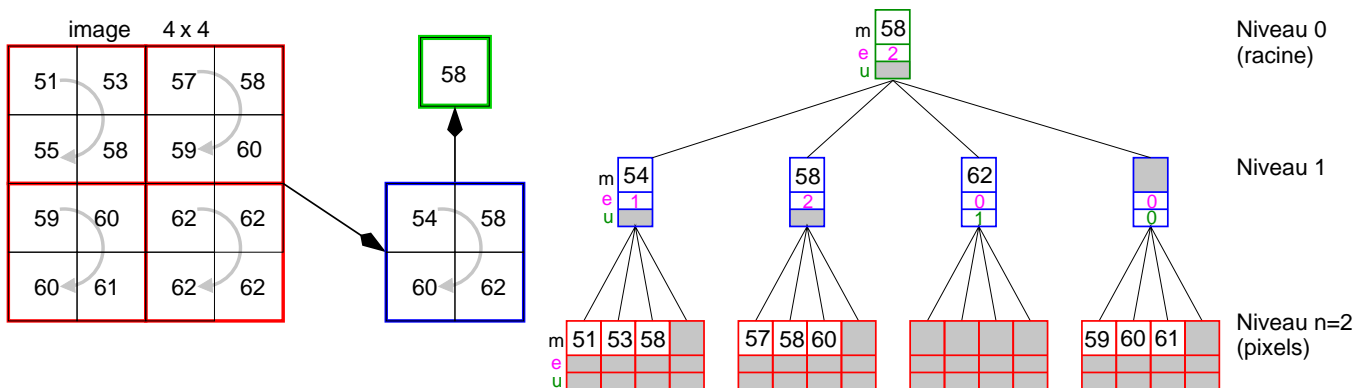
- Au niveau terminal (pixels), les nœuds sont forcément uniformes ( $u = 1$ ), d'erreur nulle ( $\epsilon = 0$ ).  
☞ il faudra penser à les initialiser ainsi, puisque ça ne sera pas fait par calcul.
- Si le bloc courant est uniforme, l'erreur  $\epsilon$  commise en calculant sa moyenne est forcément nulle. On en déduit que si l'erreur associée à la moyenne d'un bloc est non nulle, c'est que le bloc n'est pas uniforme.  
☞ il n'y a que dans le cas où  $\epsilon = 0$  qu'il est nécessaire de coder le bit d'uniformité  $u$
- Pour un nœud interne, on peut ne coder que trois des valeurs  $m_i$  de ses fils (le décodeur interpolera la 4<sup>o</sup>), mais il ne pourra pas deviner si ce 4<sup>o</sup> fils est uniforme ou non et quelle erreur  $y$  est associée  
☞ il est tout de même nécessaire de coder la valeur  $\epsilon$  (et donc éventuellement le bit  $u$ ) du 4<sup>o</sup> fils, comme dans l'exemple précédent, pour le dernier bloc du niveau 1.

- Un nœud interne peut très bien avoir ses 4 fils identiques (leur champs  $m$  sont égaux) que le bloc peut être considéré comme *uniforme*. Il faut également que toute sa descendance (les nœuds du sous-arbre dont il est racine) ait été déclarée *uniforme*.

👉 le critère d'«erreur nulle» ou même le fait que les fils vérifient ( $m_1 = m_2 = m_3 = m_4$ ) de suffit pas. Il faut aussi que ( $u_1 = u_2 = u_3 = u_4 = 1$ ). Si un seul de ses fils n'est pas uniforme, le nœud ne l'est pas.

## Schéma de codage compressif sans perte

C'est bien sûr l'exploitation des blocs uniformes qui va surtout permettre de *compresser* l'image. Cela suppose évidemment que l'image à traiter en contienne suffisamment, sinon les données codées à partir du quadtree risquent d'être tout de même plus volumineuses que les données originales.



Toujours sur le même exemple, l'exploitation de toutes ces caractéristiques fait qu'au final seules les cases non grisées du quadtree doivent être encodées : les blocs uniformes disparaissent, de même que les bits d'uniformité des nœuds d'erreur non nulle, ainsi que les champs  $u$  et  $\epsilon$  du niveau terminal (pixels).

Les données à encoder se réduisent à : 13 ( $m$ ) (sur 8 bits)+ 5 ( $\epsilon$ ) (sur 2 bits)+ 2 ( $u$ ) (sur 1 bit).

👉 soit au total  $13 \times 8 + 5 \times 2 + 2 \times 1 = 116$  bits contre  $16 \times 8 = 128$  bits pour l'image elle-même

👉 taux de compression :  $\tau = 116/128 \approx 90\%$

## Écriture des données du quadtree

Avec le schéma précédent, les données du quadtree s'écrivent<sup>(3)</sup> comme une suite hétérogène de mots de 8 bits (moyennes  $m$ ), de 2 bits (erreurs  $\epsilon$ ) et de 1 bit (marqueurs  $u$ ), de la racine, vers les feuilles.

Sur l'exemple courant, la séquence codée serait donnée par la deuxième ligne de la table suivante. La troisième ligne indique quant à elle la taille, en bits, des valeurs codées. Bien sûr, pour le codage du niveau terminal 2, on ne code plus que des valeurs de moyenne sur 8 bits.

👉 ce sont ces 116bits qui constituent la séquence encodée qu'il faudra ensuite écrire sur un fichier formaté :

champs	$m$	$\epsilon$	$m$	$\epsilon$	$m$	$\epsilon$	$m$	$\epsilon$	$u$	$\epsilon$	$u$	$m$	$m$	$m$	$m$	$m$	$m$	$m$
valeurs	58	2	54	1	58	2	62	0	1	0	0	51	53	58	57	58	60	59
tailles	8b	2b	8b	2b	8b	2b	8b	2b	1b	2b	1b	8b	8b	8b	8b	8b	8b	8b
	racine		niveau 1										niveau 2					

Selon le type d'image, les performances pourront être meilleures ou moins bonnes.

<sup>(3)</sup> cela sera vu en TP

On peut évaluer la pire situation possible :

une image sans aucun bloc uniforme et dont toutes les valeurs d'erreur  $\epsilon$  sont nulles :  $\rightarrow$  il faut encoder tous les nœuds complets ( $m, \epsilon = 0, u = 0$ ) : 11b, sauf pour le niveau terminal (juste ( $m$ ) : 8b).

ce cas extrême<sup>(4)</sup> «coûte»  $11 * (1 + \sum_{k=1}^{n-1} (4^k - 4^{k-1})) + 8 * (4^n - 4^{n-1})$  soit 109,375% de la taille initiale.

Dans tous les cas ces données brutes ne suffiront pas au décodeur : il faut lui donner quelques informations supplémentaires.

### Données d'en-tête $\rightarrow$ format

Comme pour tout format d'image, quelques données supplémentaires seront nécessaires au décodeur pour reconstruire l'image initiale.

- Le fichier commencera par un **code d'identification**<sup>(5)</sup> (par exemple les deux caractères "Q1").
- Outre ce code, les seules informations indispensables sont les dimensions de l'image. Dans le cadre de ce projet, on ne manipulera que des images de taille  $2^n \times 2^n$ . Il suffira donc de coder la **valeur n**, qui correspond en fait au nombre de niveaux de l'arbre (1 octet suffit).
- Métadonnées** : en plus des données d'en-tête nécessaires au décodage, le fichier compressé devra contenir, en commentaire, les **date et heure de création** du fichier et le **taux de compression** des données image. Ces commentaires seront, comme dans le cas des images PGM, des lignes de texte ASCII en nombre quelconque commençant par le caractère spécial #.

code	com.	nb.niv.	m	ε	m	ε	m	ε	m	ε	u	ε	u	m	m	m	m	m	m	m	m	m
Q1	#.....	2	58	2	54	1	58	2	62	0	1	0	0	51	53	58	57	58	60	59	60	61
16b	?	8b	8b	2b	8b	2b	8b	2b	8b	2b	1b	2b	1b	8b	8b	8b	8b	8b	8b	8b	8b	8b
en-tête			racine		niveau 1								niveau 2									

## Schéma de décodage

Pour reconstituer l'image, il faut bien sûr disposer d'un décodeur adapté qui devra :

- lire et évaluer le code d'identification du format
- extraire les données de dimension (hauteur de l'arbre) n.
- extraire les données compressées du flux binaire (blocs hétérogènes de 8, 2 ou 1 bit(s)) et, parallèlement, construire les niveaux successifs du quadtree.
- recréer l'image bitmap originale et la sauver sous un format d'image brut (PGM).

### Reconstruction du quadtree

Cette opération est le symétrique de la phase d'écriture du codeur. Il faut décomposer le flux entrant en blocs de p (8, par défaut) bits, suivis de blocs de 2 bits (les valeurs d'erreurs  $\epsilon$ , sauf pour le niveau terminal), eux mêmes pouvant être suivis de bits isolés (champ d'uniformité u, lorsque l'erreur  $\epsilon$  est nulle).

On rappelle que pour le quatrième fils d'un nœud, la valeur de m n'a pas été codée, contrairement aux valeurs  $\epsilon$  et éventuellement u. Il faudra donc prendre soin de construire le quadtree en même temps pour être en mesure d'identifier les *quatrième fils* : toute valeur lue dans flux binaire est immédiatement intégré au quadtree.

Pour un tel nœud, il faudra interpoler sa valeur  $m_4$  à partir de celles de son père et de ses trois frères et de l'erreur  $\epsilon$  de son père :  $m_4 = (4m + \epsilon) - (m_1 + m_2 + m_3)$

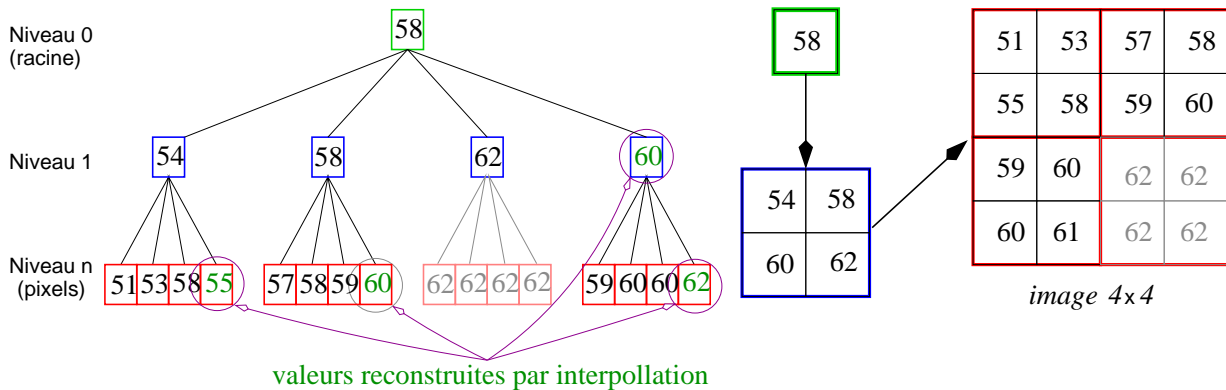
<sup>(4)</sup>un exemple très simple : un damier de pixels noirs et de pixels blancs (cf. `chessboard.256.pgm`)

<sup>(5)</sup>on appelle souvent ce code «magic number», tous les formats en ont un

## Reconstruction de l'image

À partir du quadtree décompressé, la dernière étape consistera à reformer la matrice carrée  $2^n \times 2^n$  des pixels, simplement en réorganisant le tableau contenant le niveau 2 du quadtree (cf. fig.).

Enfin, le  *pixmap*  ainsi reformé sera sauvé sur fichier pour reformer une image au format PGM. En commentaire de l'image devront figurer les dates et heures de compression (lues dans le fichier compressé) et de décompression.



## Adaptation pour la compression avec pertes

Vous remarquerez assez vite que, sur des images de type photographique, le traitement proposé ici produit en sortie des fichiers souvent plus volumineux que les originaux.

C'est normal : de telles images ne contiennent généralement que très peu de blocs parfaitement uniformes mais de nombreux blocs relativement homogènes. C'est ceux-là qu'il faut exploiter. Pour améliorer les performances, il faut introduire des pertes dans le processus de compression.

Un processus de compression est dit *avec pertes*, ou *destructifs* s'il ne permet pas de reconstituer les données à l'identique. L'intérêt de telles techniques est d'améliorer le taux de compression en acceptant de perdre une part de l'information originale jugée non indispensable.

Dans le cas présent, le processus de compression est basé sur la recherche de blocs *uniformes* : plus ils sont nombreux et de grandes tailles, plus le processus sera performant.

Ainsi, il suffira, à la compression, de *dégrader* les blocs *presques uniformes* pour les rendre *parfaitement uniformes*. Pour cela, lors de la construction du quadtree, il faudra évaluer, en plus de la *moyenne* des quatre voisins, un paramètre capable d'indiquer si le bloc correspondant (i.e. l'intégralité du sous-arbre) peut être déclaré *presque uniforme*.

Cela passe par une évaluation montante de la **variance** du bloc d'image dont le nœud est la racine.

## Variance : définition mathématique et adaptation au cas du quadtree

Pour un bloc de  $n$  pixels adjacents d'intensités  $(x_k)_{0 \leq k < n}$ , sa variance  $\nu$  se calcule en théorie par :

$$\nu^2 = \sum_{k=0}^{k < n} (m - x_k)^2 \quad \text{où} \quad m = \frac{1}{n} \sum_{k=0}^{k < n} x_k \quad \text{est la moyenne du bloc}$$

Cette grandeur mesure donc le degré de *dispersion* des valeurs autour de la moyenne : plus la variance est faible, plus le bloc est homogène. Pour un bloc parfaitement uniforme, la variance est nulle.

Dans le cas du quadtree nous utiliserons une formule légèrement différente et beaucoup plus économique.

En notant  $(m, \nu)$  la moyenne et la variance du nœud courant et  $(m_k, \nu_k)_{0 \leq k < 4}$  celles de ses nœuds fils, on calculera  $\nu$  par :

$$\mu = \sum_{k=0}^{k < 4} (\nu_k^2 + (m - m_k)^2) \Rightarrow \nu = \frac{\sqrt{\mu}}{4}$$



On calcule ainsi une variance locale qui prend en compte la dispersion des valeurs dans tout le sous-arbre. Par exemple un grand bloc globalement homogène mais traversé par un trait (un contour), gardera une variance élevée.

## Filtrage du quadtree

Ce filtrage va consister à décréter qu'un nœud dont la variance est «faible» (i.e. inférieure à un certain seuil) peut être déclaré uniforme : son champ binaire  $u$  est simplement mis à 1 (et son champs d'erreur  $\epsilon$  à 0) et tout le sous-arbre sera ignoré lors de l'écriture..

La question du réglage de ce seuil de segmentation dépasse largement le cadre de ce projet : pour être réellement efficace il doit être évolutif et adaptatif au fur et à mesure que l'on descend dans l'arbre.

Nous nous contenterons d'une version primaire permettant d'illustrer le principe :

- pour que l'algorithme garde un comportement à peu près stable (pas trop dépendant de l'image à traiter) il faudra extraire les variances moyenne ( $\text{medvar}$ ) et maximale ( $\text{maxvar}$ ) sur l'ensemble des nœuds du quadtree.
- on choisira comme seuil de départ  $\sigma$  la valeur  $\sigma = \frac{\text{medvar}}{\text{maxvar}}$ .
- il faudra également disposer d'un *paramètre*  $\alpha$  pour faire évoluer le seuil durant le parcours de l'arbre.

☞ cet  $\alpha$  sera à fixer en paramètre de la ligne de commande.

Les résultats obtenus varient un peu d'une image à l'autre, mais globalement, sur une image photographique :

- $\alpha \leq 1.0 \rightarrow$  : aucun filtrage, donc aucun gain supplémentaire
- $\alpha \approx 1.5 \rightarrow$  : résultats intéressants, filtrage moyen, gain correct
- $\alpha \geq 2.0 \rightarrow$  : filtrage excessif, image très abîmée.

☞ ci dessous, pour une même image (512x512), 4 tests pour des valeurs de  $\alpha \in \{1.4, 1.6, 1.8, 2.0\}$  qui donnent des taux de compression respectifs de  $\tau \in \{60\%, 32\%, 14\%, 6\%\}$

Sur cette même image, l'algorithme sans pertes donne un taux de  $\tau = 109,12\%$ , soit pratiquement la pire situation possible.





- l'algorithme de filtrage récuratif-suffixe se déroule alors comme suit :

```
pseudo code pour la fonction de filtrage
fonction filtrage(noeud* node, reel  $\sigma$ , reel  $\alpha$ )  $\rightarrow$  {0 ou 1}
{
    si (node->u == 1)  $\rightarrow$  1;      /* noeud déjà uniforme */
    si (est_feuille(node))  $\rightarrow$  1; /* noeud feuille */
    /* descente dans les niveaux inférieurs : *
    * il faut les faire tous et jusqu'au bout */
    entier s = 0;
    s += filtrage(fils(node,1), ( $\sigma * \alpha$ ),  $\alpha$ );
    s += filtrage(fils(node,2), ( $\sigma * \alpha$ ),  $\alpha$ );
    s += filtrage(fils(node,3), ( $\sigma * \alpha$ ),  $\alpha$ );
    s += filtrage(fils(node,4), ( $\sigma * \alpha$ ),  $\alpha$ );
    /* le noeud courant est 'uniformisé' seulement si : *
    * - ses 4 fils ont déjà été 'uniformisés'          *
    * - sa variance vérifie les conditions du filtre */
    si ( (s < 4) ou (node->v >  $\sigma$ ) )  $\rightarrow$  0;
    node->e = 0;
    node->u = 1;
     $\rightarrow$  1;
}

/* filtrage du quadtree : sur la racine (root) */
filtrage(root, (medvar/maxvar),  $\alpha$ );
```

Pour mieux évaluer l'effet de ce filtrage, il est très simple et très pratique de disposer d'une fonction permettant de produire une image PGM montrant la «grille de segmentation», comme dans les exemples ci-dessus. L'édition de cette "grille" pourra être activée au codage **et/ou** au décodage par une option du **codec**.

De son côté, le décompresseur n'a pas à savoir que le codage a introduit des pertes : il formera des blocs parfaitement uniformes. Il n'y a donc rien de plus à faire de ce côté.

Remarque : (pour les curieux....)

Avec ce processus simple, les résultats sont peu convaincants : on passe d'une dégradation faible (donc gain faible) à une dégradation trop forte, assez brutalement.

Une adaptation beaucoup plus performante consiste à faire évoluer le seuil  $\alpha$  de niveau en niveau avec un second paramètre  $\beta$  :  $\alpha \leftarrow \alpha^\beta$ . La segmentation est alors beaucoup plus fine mais aussi beaucoup plus sensible aux valeurs initiales de  $\alpha$  ( $> 1.$ ) et  $\beta$  ( $< 1.$ ).

Et comme ce traitement n'a strictement aucune incidence sur le format final ni sur le décodage, c'est là qu'un encodeur peut s'avérer bien meilleur qu'un autre : il augmentera le taux de compression en limitant les dégradations visibles.

## Conditionnement - bibliothèque partagée

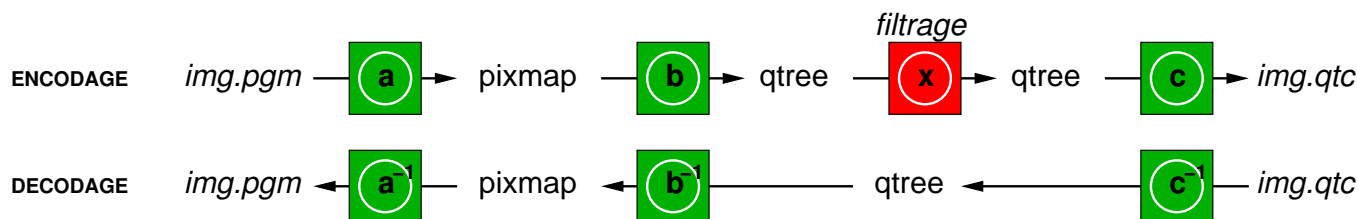
### Résumé de la chaîne de codec

Comme toute application de type **encodeur-décodeur**, ce projet se résume à une chaîne de «boîtes de traitement» connectées (la sortie de l'une devient l'entrée de la suivante, et inversement).

Ici, il y aurait donc 4 de ces «boîtes». 3 sont parfaitement réversibles, la 4<sup>e</sup> (introduction des pertes) ne l'est pas :

- Ⓐ du fichier source .pgm vers les données image (le pixmap) – **réversible**
- Ⓑ du pixmap vers le quadtree – **réversible**
- ⓧ filtrage du quadtree – **non réversible**
- Ⓒ du quadtree vers le fichier encodé .qtc – **réversible**

Dans cette chaîne, seuls les points d'entrée et de sortie sont **formatés** : les fichiers .pgm et .qtc. Tout le reste est libre.



## Bibliothèque partagée ([qtc.h/libqtc.so](http://qtc.h/libqtc.so)), Makefile(s) et programme applicatif

Au final, tous ces modules devront être assemblés sous la forme d’une bibliothèque partagée `libqtc.so` associée à une interface `qtc.h` regroupant tous les éléments publics de cette lib (et uniquement ceux-là).

C’est un programme annexe, constitué essentiellement d’une fonction `int main(int argc, char** argv)` – plus quelques fonctions périphériques simples (aide, analyse de la ligne de commande...) – qui permettra de mettre en œuvre les fonctionnalités de la bibliothèque.

La compilation de la bibliothèque (assemblage des modules dans un `shared object` `libqtc.so`) devra donc se faire de manière totalement indépendante de celle du prog. applicatif de test. Il peut être intéressant à ce niveau de disposer de 2 fichiers `Makefile` différents.

De même il est possible de restructurer le répertoire de travail pour bien séparer la bibliothèque des prog. applicatifs.

## Ce qu’il vous est demandé

Il vous est demandé de développer un codec (codeur/décodeur) pour ce format : de l’image `.pgm` au fichier compressé `.qtc` et inversement.

Le choix entre codeur et décodeur ainsi que le choix des options (le taux de perte par exemple) se feront par analyse des arguments de la ligne de commande.

### paramètres

Parmi les options de la ligne de commande :

- le sens de traitement ( `-c` : encodeur, `-u` décodeur )
- le nom du fichier d’entrée `-i input.{pgm|qtc}` (en cohérence avec l’option précédente)
- le nom du fichier de sortie :
  - par défaut le fichier de sortie sera nommé `{QTC|PGM}/out.{qtc|pgm}`
  - pour renommer le fichier de sortie on utilisera l’option `-o output.{qtc|pgm}`
- édition de la grille de segmentation ( `-g` ) pour le codeur **et** le décodeur.
- les classiques : `-h` (aide), `-v` (mode bavard – à vous de choisir les informations pertinentes)
- d’autres options sont bien sûr possibles et bienvenues tant qu’elles sont bien documentées.

### Quelques exemples :

- `$> codec -c -i PGM/toto.pgm` : encode le fichier `PGM/toto.pgm` dans `QTC/out.qtc`
- `$> codec -c -a 1.5 -g -i PGM/toto.pgm -o QTC/toto.qtc` : encode `PGM/toto.pgm` avec perte ( $\alpha = 1.5$ ) dans `QTC/toto.qtc` et édite la grille de segmentation correspondante dans `PGM/toto_g.pgm`
- `$> codec -u QTC/toto.qtc -g` : décode `QTC/toto.qtc` dans `PGM/out.pgm` et édite la grille de segmentation correspondante dans `PGM/out_g.pgm`

Votre programme devra bien évidemment gérer correctement les erreurs sur la ligne de commande (option non reconnue, tentative de décompression d’un fichier PGM...), ainsi que toutes les erreurs qui pourraient se produire en cours d’exécution (éviter au maximum le recours à la fonction `exit` dans votre code) : si votre programme échoue, il doit se terminer proprement en donnant quelques explications.

Bref, inspirez vous du comportement classique des programmes et commandes `LINUX` pour rendre votre programme convivial et intuitif, robuste aux erreurs et facile à utiliser.

## modularité

Le code devra être structuré en différents modules (`src/*.c` `include/*.h`) mais le choix du découpage vous appartient.

☞ une grande attention sera portée à cette modularité : bon usage des *headers*, des données privées et publiques... En particulier chaque module devra être compilable (production de son fichier objet) indépendamment des autres.

## Portabilité

le plus important est de respecter scrupuleusement le format de codage proposé ici. En particulier, votre décodeur devra être capable de lire et reconstruire un fichier compressé par le codeur d'un autre groupe ou de vos enseignants, et inversement vos fichiers compressés devront être lisibles par les autres décompresseurs.

Le programme doit fonctionner sur les machines de l'université. En conséquence n'essayez pas de l'intégrer dans une "jolie" interface graphique : ça ne marche jamais! Il doit simplement lire et écrire des fichiers formatés en ligne de commande et surtout être totalement indépendant de toute bibliothèque externe (du pur C, bien standard).

Des fichiers PGM et des fichiers compressés (`.qtc`) de test seront mis à votre disposition (en temps utile).

## Rendre le projet

Votre projet devra être rendu sur la zone de dépôt eLearning sous la forme d'une archive précisant vos noms `L3.2024.ProgC-Nom1.Nom2.[tgz|zio|rar|...]`

☞ **rappel** : les systèmes Unix-like détestent les espaces, accents et autre caractères bizarres dans les noms de répertoire/fichiers. Au besoin, utiliser le caractère **underscore** `'_'`.

## Documentation et structuration : quelques conseils

Il ne vous est pas demandé de rapport mais votre projet devra être bien documenté (commenter les modules, les fonctions, expliciter les variables, sans en faire trop...). Une documentation [Doxygen](#) sera bienvenue mais n'est pas imposée.


D'autre part, il devra être accompagné d'un fichier texte (brut) **Readme** précisant clairement vos noms, prénoms, groupe de TD et expliquant le fonctionnement de votre projet : comment le compiler, où trouver le fichier exécutable, quelles sont les options de la ligne de commande (quelques exemples d'utilisation).... bref tout ce qui peut faciliter l'usage de votre projet et mettre vos «clients» (les correcteurs!) dans de bonnes dispositions.

Dans ce même **Readme**, et puisqu'il s'agit d'un travail de groupe (2, c'est déjà plusieurs...) vous préciserez quelle a été la répartition des tâches sur les différentes étapes ainsi que les difficultés rencontrées.

## Important

Si vous voulez être évalués équitablement respectez les règles suivantes :

- Structurez votre projet correctement :
  - tout doit se décompresser dans un répertoire `L3.2024.ProgC-Nom1.Nom2/` .
  - ce répertoire doit être structuré de manière *standard* : des sous-répertoire `src/` `include/` , un **Makefile** , un **Readme** , un sous-répertoire `PGM/` pour les fichiers images et un répertoire `QTC/` et les fichiers encodés.
  - Au moment de l'archivage votre répertoire ne doit contenir ni fichiers objets, ni exécutable, ni pdf ni images ni fichiers codés (en particulier, **vider** les répertoires `PGM/` et `QTC/`).L'archive déposée ne doit contenir que des *sources* et ne doit peser que quelques ko.  
On doit retrouver cette architecture au désarchivage (y compris les répertoires vides).

- Votre programme doit **impérativement** fonctionner (compilation/exécution) sur les machines de l'université (salle de TP, système **Linux**).  
 l'argument "*ça marche très bien sur mon Mac*" est irrecevable. Il existe de très bons émulateurs **linux** pour tout système...
- N'oubliez que les correcteurs disposent d'outils de détection de plagiat.

Ces conseils sont évidemment valables pour les autres projets que vous aurez à faire cette année et les suivantes...

## Evaluation

Les critères d'évaluation sont :


- la bonne réalisation de ce qui est demandé (en particulier le respect du format d'encodage)
- la qualité de la programmation (architecture, clarté du code, etc.)
- la qualité de la documentation,
- l'éventuelle présence d'options,

De manière générale la note attribuée à un projet est *inversement proportionnelle* au temps nécessaire pour son évaluation : un bon projet s'évalue vite ... et inversement.

## Options

Les améliorations possibles sont nombreuses. Vous avez carte blanche pour les options mais n'implantez pas d'option tant que le reste n'est pas parfaitement fini. Mieux vaut un projet sans option bien propre et qui fonctionne qu'un projet avec beaucoup d'options qui est mal construit ou qui ne fonctionne pas.

Typiquement, la partie *Compression avec pertes* se traite en dernier, une fois que l'on est sûr que les formats d'entrée/sortie sont bien respectés.

 Procédez par paliers et conservez toujours une copie de la "dernière version qui marche bien" avant d'attaquer la suite.