Institut Universitaire de Technologie de Tours
Département Génie Électrique et Informatique Industrielle

Mr. Brault
Mme. Dougherty

Les formats de compression d'image

MICHOT Julien
2ème Année, groupe B1

Promotion 2002-2004

Institut Universitaire de Technologie de Tours
Département Génie Électrique et Informatique Industrielle

Mr. Brault
Mme. Dougherty

Les formats de compression d'image

MICHOT Julien
2ème Année, groupe B1

Promotion 2002-2004

Sommaire

Introduction

C’est avec l’apparition des ordinateurs, et surtout avec Internet, que les images sont devenues omniprésentes et prépondérantes. En effet, quoi de plus précis qu’une image ? Il faudrait plusieurs pages de texte pour décrire une simple photographie, ou un schéma… Cependant, si dans la rapidité d’information l’image détrône facilement le texte, celle-ci devient vite lourde de données, donc excessivement longue à transmettre et à traiter. Voilà pourquoi depuis quelques années, les centres de recherche en informatique dépensent de nombreuses heures sur des algorithmes de compression. Afin de limiter la taille, ou le poids, d’une image, nous devons la compresser, c’est-à-dire éliminer les informations inintéressantes ou redondantes. Il existe de nos jours plus d’une vingtaine de formats de compression, spécifiquement dans la compression d’image (.gif, .jpeg, .bmp…), ayant chacun leur propre méthode de codage, ou cumulant plusieurs algorithmes, mais tous sont complémentaires. Étant donné la nature de vulgarisation de ce rapport, ainsi que la limitation du nombre de page qu’il nous est imposé, nous n’expliquerons que succinctement quelques différentes méthodes de compression, en délaissant l’approfondissement mathématique des codages.

Nous présenterons dans un premier temps les caractéristiques d’une image, en tant qu’entité informatique, puis nous analyserons les principes de codage et de compression d’une image, avec quelques exemples de format. Enfin, nous étudierons les deux « futures » méthodes de compression les plus prometteuses : la compression par fractale et par ondelette.

I. Caractéristiques d'une image

Pour comprendre comment fonctionne la compression d'image, nous devons tout d'abord savoir ce qu'est une image, quelles sont les différentes représentations informatiques, par quels moyen peut-on réduire la taille des fichiers, comment représenter les couleurs bref autant de choses qu'il est nécessaire de détailler dans une première partie.

1. Généralités

a. Notion de pixel

En informatique, et ce pour des raisons de gain de place, une image est composée d'un ensemble de points, appelés pixel (abréviation de *PICTure Element*). Ces pixels sont regroupés en lignes et en colonnes afin de former un espace à deux dimensions. Chaque point sera représenté par ses coordonnées (X,Y), avec X l'axe orienté de gauche à droite, et Y l'axe orienté de haut en bas.

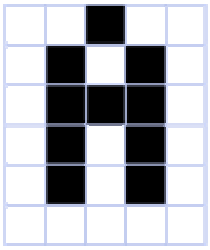


Figure 1. Pixels

b. Représentation de la couleur

En plus de ses coordonnées planaires, un pixel se caractérise par sa pondération, appelée aussi profondeur de codage, qui représente sa couleur ou son intensité. Cette valeur peut être codée sur un nombre n différent de bits (ou octet) selon les méthodes de codage de la couleur utilisées. Les standards les plus répandus sont n=1 bit (noir ou blanc), n=4 (16 couleurs ou 16 niveaux de gris), n=8 bits (256 couleurs ou 256 niveaux de gris) … Les différents codages de la couleur seront évoqués dans la partie I.2.

On appelle la palette de couleur, l'ensemble des couleurs que peut contenir une image. Il est fréquent de voir des images qui n'utilisent jamais certaines couleurs, il devient dès lors intéressant de limiter la palette de couleur en ne sélectionnant que la ou les couleurs utilisées réellement par l'image. Il suffit pour cela, d'attribuer un nombre à la couleur, et de copier la correspondance chiffre-couleur dans l'entête de l'image (utilisée lors du décodage).

c. Taille et définition d'une image

Pour connaître la définition (en octets) d'une image, il est nécessaire de compter le nombre de pixels que contient l'image, cela revient à calculer le nombre de cases du tableau, soit la hauteur de celui-ci que multiplie sa largeur. La taille (ou poids) de l'image est alors le nombre de pixels que multiplie la taille de chacun de ces éléments. Les définitions les plus répandues sont 640 x 480, 600 x 800, 1024 x 768 pixels…

Prenons l'exemple d'une image 1024 x 768, dont la couleur est codée sur 24 bits (1 octet pour les nuances de rouge, 1 pour le bleu et 1 octet pour le vert, codage True color ou RGB)

- Nombre de pixels :

1024 x 768 = 786432 pixels

- Poids de l'image :

786432 x 3 = 2359296 octets

- soit une image de 2359296 / 1024 = 2304 Ko, ou 2304 / 1024 = 2,25 Mo, ce qui est assez conséquent, surtout lorsqu'on veut transmettre l'image…

d. Deux types de compression

L'enjeu de la recherche sur la compression d'image est de trouver un moyen de diminuer la taille d'une image, tout en essayant de limiter la dégradation due à la compression. Il existe deux grandes familles d'algorithme de compression : la non-destructrice et la destructrice.

La première restitue une image totalement identique à l'original, il n'y a donc aucune perte d'information. Cette famille d'algorithme est essentielle dans nos ordinateurs, la totalité des programmes d'archivage sont bâtis sur cette technique de compression sans perte d'information. Le principe courant de ces programmes est de réduire les séquences d'information redondantes.

La méthode destructrice délivre après compression une image différente, elle contient beaucoup moins d'information que l'original, certains détails auront été éliminés lors du codage. Cette modification est plus ou moins visible, selon le degré de compression. L'attrait de cette famille est qu'elle peut obtenir un rendement formidablement grand. Cependant, on ne peut utiliser ce genre de compression que pour des sons, vidéos et images, mais pas sur des fichiers ou sur du texte puisque ceux-ci ne doivent subir aucune dégradation.

On appelle ratio le rapport entre la taille d'une image non compressée et la taille de la même image, mais compressée. Le taux de compression étant l'inverse du ratio, un ratio de n : 1 est équivalent à un taux de $\frac{1}{n}$. Ainsi, avec une image de 2 Mégaoctets non compressée, et

avec un algorithme de ratio 4 : 1 (ou par équivalence, avec un taux de compression de $\frac{1}{4}$), nous devrions obtenir une image compressée de taille 0,5 Mo ($2 \times \frac{1}{4} = 0{,}5$) .

.

2. Le codage de la couleur

Pour représenter sur un périphérique externe, un écran par exemple, la couleur d'un pixel, nous avons besoin de codifier la couleur sur un ou plusieurs octets. En effet, la lumière est une onde, et la couleur varie en fonction de la longueur de cette onde. Comme tout signal analogique, nous avons besoin de numériser cette donnée. Nous allons donc parcourir les codages de couleur les plus répandus, qui se différencient par leur mode de représentation de la couleur.

- Le codage RGB[1] correspond à la façon dont les couleurs sont codées informatiquement, ou plus exactement à la manière dont les tubes cathodiques des écrans d'ordinateurs représentent les couleurs. Ce codage réserve un octet pour chaque couleurs (Rouge, Bleu et Vert). Aussi, nous obtenons 256 intensités de rouge (2^8), 256 intensités de vert et 256 intensités de bleu, soient 16 777 216 possibilités théoriques de couleurs différentes, c'est-à-dire plus que ne peut en discerner l'œil humain (environ 2 millions). Cependant, cette valeur n'est que théorique car elle dépend fortement du matériel d'affichage utilisé.

Etant donné que le codage RGB repose sur trois composantes proposant la même gamme de valeur, on le représente généralement graphiquement par un cube dont chacun des axes correspond à une couleur primaire :

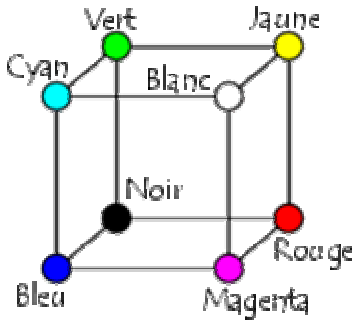


Figure 2. Représentation graphique du codage RGB

- Le codage TSL[2] (ou HSL), inventé grâce aux travaux du peintre Albert H.Munsell , est un modèle de représentation dit "naturel", c'est-à-dire proche de la perception physiologique

[1] *Rouge, Vert, Bleu ou en anglais :Red, Green, Blue*

[2] *Teinte, Saturation, Luminance ou en anglais : Hue, Saturation, Luminance*

de la couleur par l'œil humain. Le modèle RGB aussi adapté soit-il pour la représentation informatique de la couleur ou bien l'affichage sur les périphériques de sortie, ne permet pas de sélectionner facilement une couleur. En effet, le réglage de la couleur en RGB dans les outils informatiques se fait généralement à l'aide de trois glisseurs ou bien de trois cases avec les valeurs relatives de chacune des composantes primaires, or l'éclaircissement d'une couleur demande d'augmenter proportionnellement les valeurs respectives de chacune des composantes. Ainsi le modèle HSL a-t-il été mis au point afin de pallier à cette lacune du modèle RGB.

Nous avons donc vu que le code HSL décompose une couleur en trois critères physiologiques :

- La teinte correspond à la couleur de base (T-shirt mauve ou orange)

- La saturation, décrivant la pureté de la couleur, c'est-à-dire son caractère vif ou terne (T-shirt neuf ou délavé)

- La luminance, indiquant la brillance de la couleur, c'est-à-dire son aspect clair ou sombre (T-shirt au soleil ou à l'ombre)

Voici une représentation graphique du modèle HSL, dans lequel la teinte est représentée par un cercle chromatique, la luminance et la saturation par deux axes :

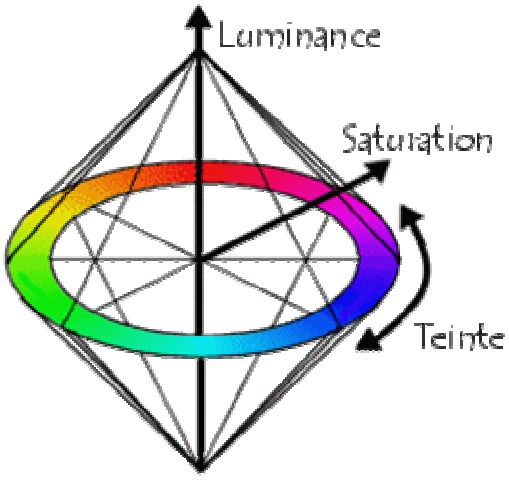


Figure 3. Représentation graphique du codage HSL

- Le codage CMYK[3] décompose les couleurs en trois couleurs (*Cyan*, *Magenta* et *jaune*). Ce codage réalise ensuite le même procédé que le codage RGB. La lettre K désigne le terme *Noir pur*, cette dernière a été ajoutée car pour réaliser du noir, on devait additionner le Cyan, le Magenta et le jaune, ce qui n'était pas économique…

[3] *Cyan, Magenta, Yellow, Noir pur*

- Le codage CIE [4] utilise la *Chromaticité* et la *Luminace*. Puisque les couleurs peuvent être perçues différemment selon les individus et qu'elles peuvent être affichées différemment selon les périphériques d'affichage, la commission internationale de l'éclairage a décidé de créer un nouveau code dont les critères sont basés sur la perception de la couleur par l'œil humain, donc indépendants du périphérique utilisé.

Voici le diagramme de chromaticité de ce codage CIE (Y : luminance, et x : chromaticité):

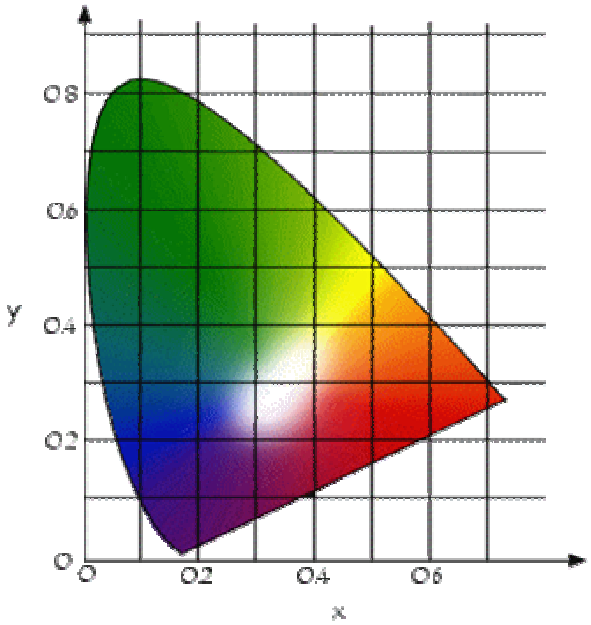


Figure 4. Représentation graphique du codage CIE

- Le codage YUV [5] est utilisé dans les standards PAL[6] et SECAM[7]. Ce codage a été conçu pour être reconnu par les télévisions en noir et blanc, qui convertissaient alors les couleurs en un dégradé de gris.

Relations en fonction du codage RGB :

$$Y = 0.299\ R + 0.587\ G + 0.114\ B$$

$$U = -0.147\ R - 0.289\ G + 0.463\ B = 0.492\ (B - Y)$$

$$V = 0.615\ R - 0.515\ G - 0.100\ B = 0.877\ (B - Y)$$

- Enfin, il existe un autre codage, le codage YIQ [8], utilisé dans le standard vidéo NTSC (Etats-Unis et Japon).

Relations :

---

[4] *Commission Internationale de l'Eclairage*

[5] *Luminance et Chrominance*

[6] *Phase Alternation Line*

[7] *Séquentiel Couleur avec Mémoire*

[8] *Luminance Interpolation et Quadrature*

$$Y = 0.299\ R + 0.587\ G + 0.114\ B$$
$$I = 0.596\ R - 0.275\ G - 0.321\ B$$
$$Q = 0.212\ R - 0.523\ G + 0.311\ B$$

3. Différentes formes d'images

Nous avons vu auparavant qu'une image était constituée d'un ensemble de points, nommés pixels, de position et de couleur différente. C'est ce qu'on appelle une image du type « bitmap », ou en français « tableau de données binaires ».

Cependant, il existe une autre forme d'image, appelée image vectorielle. Les images vectorielles sont des représentations géométriques (cercle, rectangle, droite, etc.…). Cette propriété spécifique aux images vectorielles, nous permet de coder une image entière en formules mathématiques (exemple : un cercle est défini par son centre et son rayon, le rectangle lui peut être déterminé par ses deux points extremums…). C'est le processeur qui sera ensuite chargé de "traduire" ces formes en informations interprétables par la carte graphique.

Le grand avantage des images vectorielles est le suivant : puisque ces images sont constituées d'entités mathématiques, il est possible de leur appliquer des transformations géométriques. Lorsqu'on applique un grossissement sur une image du type bitmap, on s'aperçoit que l'image n'est plus nette, et selon le degré de grossissement, on peut observer les pixels, surtout sur les contours des objets (cf. figure 5). Avec les images vectorielles, la précision ne dépend pas du facteur d'agrandissement.

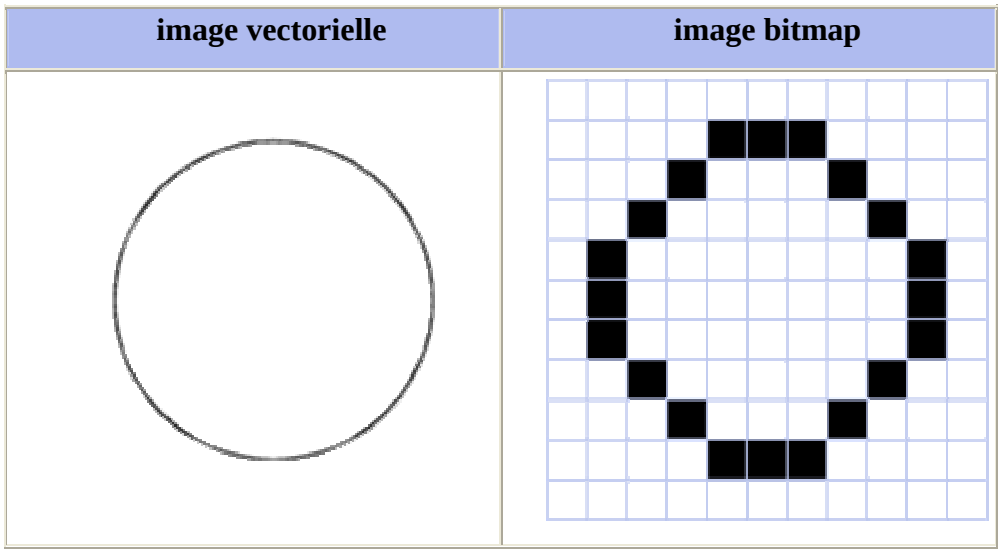


Figure 5. Comparaison d'un grossissement d'une image vectorielle et d'une image Bitmap

Les images vectorielles, appelées aussi « Cliparts » permettent donc de définir une image

avec très peu d'information, d'où le gain de place. En comparaison, une image vectorielle de 1m sur 1m a le même poids qu'une image du type Bitmap de taille 0,1mm sur 0,1mm. De plus, contrairement au Bitmap, il est très facile de modifier l'image, par exemple une couleur ou une forme. Ces grandes qualités de compression et de maniabilité de ces images ont intéressé les programmeurs, et notamment sur Internet. Un langage d'animation, le Flash (de la société Macromedia), utilise comme de nombreux programmes de CAO[9] et de création 3D[10], la représentation vectorielle. Cependant, ce format ne convient pas pour des images trop complexes, comme des photographies par exemple aussi, d'autres algorithmes ont été inventés.

II. Codages et compressions d'une image

1. Le format RLE[11]

Le principe de compression de RLE est assez simple à mettre en œuvre. Il repose sur le fait que dans une image, il existe de nombreuses répétitions d'un même pixel, ou d'une même séquence de pixel, tous juxtaposés. Ainsi, au lieu de coder chaque pixel d'une image, le RLE propose de coder d'une part le nombre de répétitions, et d'autre part la séquence ou l'octet à répéter. Lorsqu'il s'agit d'une répétition de séquence, on ajoute entre le nombre de répétitions et la séquence, un octet représentant le nombre d'octets de la séquence. (cf. tableau 1) Si la taille de celle-ci est impaire, alors on ajoute un octet nul (00) à la fin.

10 10 10 10 10	est codé 05 10
0A 0A 0A 0A 0A 0A 0A 0A 0A 0A	est codé 0A 0A
23 65 55 34 22	est codé 00 05 23 65 34 22 00
10 10 89 23	est codé 00 04 10 10 89 23

Tableau 1. Exemple de codage du format RLE

En plus de chaque pixel, le code RLE doit aussi coder un saut de ligne (sur deux octets : 00 00), et la fin de l'image (sur deux octets aussi : 00 01). Ce codage présente une autre particularité puisqu'il peut aussi coder le déplacement du « pointer » (lors du traitement de

[9] Conception assisté par ordinateur

[10] 3 dimensions

[11] Run Length Encoding

l'image), et ainsi omettre volontairement des pixels. La syntaxe est la suivante : 00 02 XX YY, avec XX le nombre de pixel sur une ligne et YY le nombre de pixel sur une colonne (en hexadécimale). Les icônes de Windows sont codées en RLE, ceci explique pourquoi certaines icônes sont transparentes à certains endroits.

Ce système de compression est assez bon dans certain cas (images monochromes ou 256 couleurs), mais il s'avère très coûteux pour des images dont la couleur est codée sur 16 ou 24 bits… Cette méthode est néanmoins largement utilisé, dans les formats (BMP, PCX, TIFF, ICO,..). Il est à noter qu'il existe plusieurs variantes de ce codage, certains types encodent l'image en décelant des séquences redondantes selon les lignes, les colonnes, ou même en zigzag (cf. figure 6) .

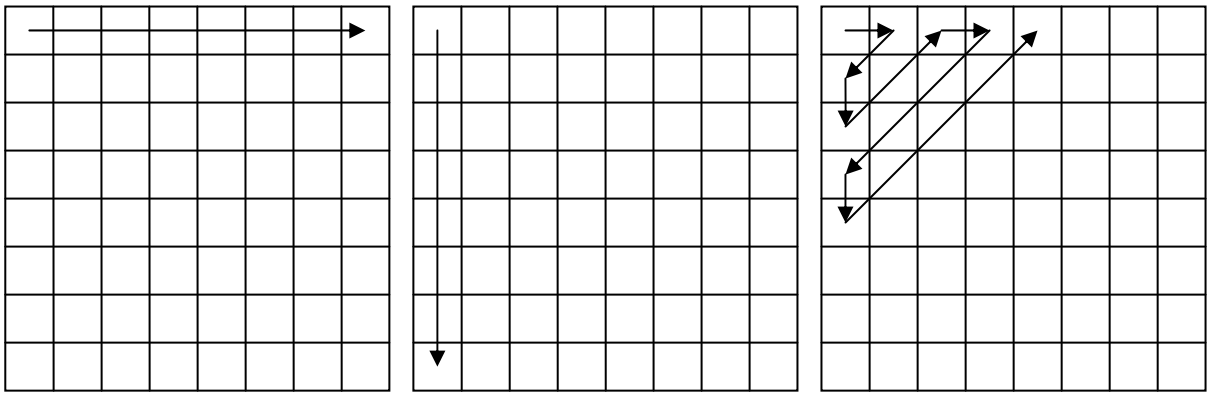


Figure 6. linéarisation en ligne, en colonne et en zig-zag

2. La compression RL[12]

La compression RL est une variante de la compression RLE. Au lieu de coder la répétition d'une même séquence sur une ligne, comme le fait le codage vu précédemment, la compression RL utilise ce qu'on désigne par codage par zone. Le procédé est simple : le programme tente de déterminer dans toute l'image des rectangles dont les pixels sont identiques. Ainsi, il suffira de coder le nombre de répétitions du rectangle à réaliser lors de la décompression, les deux points extremums (P1, P2) de chaque rectangle, et enfin la séquence du rectangle. (cf. figure 7)

[12] Run Length

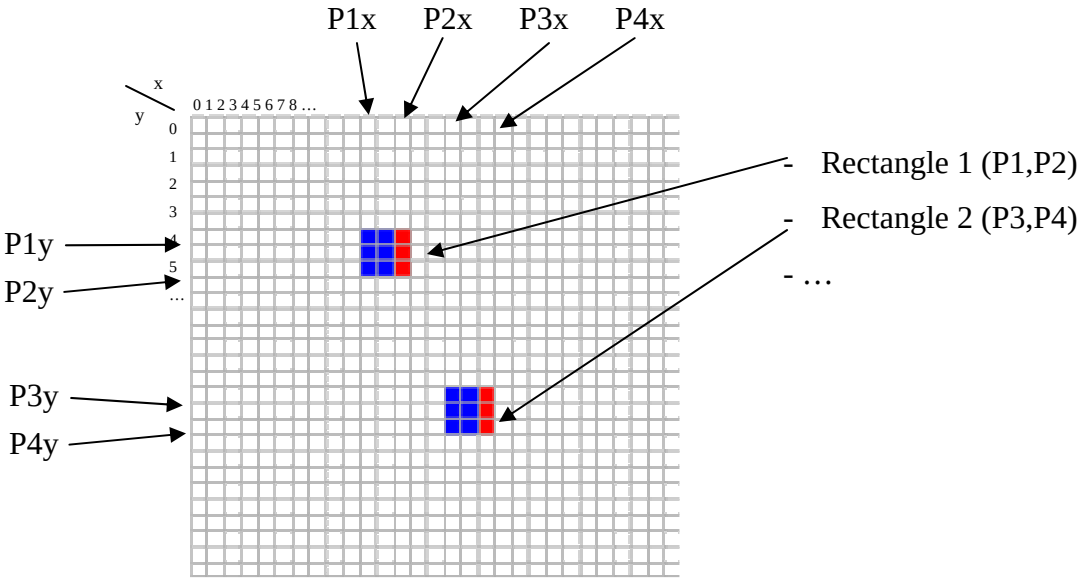


Figure 7 . Exemple d'image au format RL

La difficulté majeure de ce mode de compression est de trouver des rectangles identiques dans l'image. Les algorithmes actuels sont récursifs, c'est-à-dire qu'ils partent d'une taille de rectangle n x m, avec n et m maximum, puis ils font tendre n et m vers 1 (1 pixel). Le problème est que cette manière de procéder est excessivement coûteuse en temps, lors de la compression, mais aussi lors de la décompression puisqu'il faut redessiner (presque) complètement l'image.

3. Le codage de Huffman

En 1952, Huffman inventa une nouvelle méthode de compression appelée compression à arbre. Le principe est simple, une image est formée par de nombreux caractères différents, mais certains reviennent plus souvent que d'autres. Aussi, l'algorithme de huffman établit un arbre contenant les signes, les symboles les plus fréquents, ainsi que leur fréquence d'apparition. Puis, à chaque caractère est assigné un code. Le signe le plus souvent utilisé, placé à la base de l'arbre, reçoit le code le plus court. Le caractère le moins fréquent aura donc le plus long code binaire. Par conséquent, la suite finale des mots codés sur des longueurs variables (petite pour les symboles courants et longs pour les mots peu fréquents), sera mathématiquement plus petite qu'avec un codage sans compression, donc sur des mots de taille standard.

Exemple d'arbre

- Source du fichier image :

7B 33 46 EE 4F 90 33 7B 1C D3 33 46 EE 7B……..

- Arbre :

Séquence	Fréquence	Code attribué
7B	3	0
33 46 EE	2	1
…	…	…

Tableau 2. Exemple d'arbre

Lors de la compression, l'algorithme écrira le tableau des équivalences Code-Séquence(cf. tableau 2), le fichier contiendra donc non plus les séquences binaires de l'image, mais les différents codes attribués par le programme de compression. Pour décompresser, il suffira de remplacer tous les codes par les séquences équivalentes (présentent dans l'entête du fichier).

Cette méthode de codage donne de bon taux de compression, principalement sur des images monochromes, pour les fax par exemple, mais il est aussi utilisé dans la compression JPEG, et ceux, pour tous les types d'image.

4. La compression LZW [13]

a) Introduction

Le système de compression d'image le plus utilisé à travers le monde reste la compression LZW (acronyme de ses inventeurs Lempel-Ziv-Welch). Cet algorithme (cf. Annexe n°1 p 29) utilise, comme la compression de Huffman vue précédemment, un tableau, un dictionnaire pour réaliser une compression du type non-destructrice. Contrairement au codage précédent, la compression LZW n'encode pas dans le fichier le dictionnaire, celui-ci sera reconstruit lors de la décompression. Le LZW est un dérivé du codage LZ. Les concepteurs, Abraham Lempel et Jakob Ziv utilisaient principalement le principe de compression LZ dans un autre format, nommé LZ77, dédié aux programmes d'archivage. Les formats ZIP, ARJ et LHA basent leur compression sur cet algorithme. En 1978 ils créèrent le compresseur LZ78 spécialisé dans la compression d'images et de fichiers binaires. Puis en 1984, Terry Welch modifia le format LZ78 pour pouvoir l'utiliser dans les contrôleurs de disques durs, le format LZW est né.

Pour comprendre plus aisément le principe de compression, nous allons étudier le code LZW dans sa forme initiale (il existe plusieurs variantes), au travers d'un exemple.

[13] Lempel, Ziz et Welch

b) Etude d'un exemple de compression

Prenons la séquence suivante : « AIDE TOI LE CIEL T AIDERA» (séquence non hexadécimale, mais ASCII, afin de mieux comprendre le procédé de codage).

Nous pouvons remarquer que cette séquence contient quelques redondances (lettres coloriées) :

AIDE TOI LE CIEL T AIDERA

Afin de mieux comprendre le principe de compression utilisant un dictionnaire, la figure ci-dessous montre à chaque étape ce qu'il ce passe dans le fichier compressé, dans le dictionnaire, et dans la mémoire tampon (ou buffer) du programme.

étape	lu	émis (déc)	émis (bin)	tampon	adresse	séquence
1	A			A	0...255	ascii 0...255
2	I	65	0100 0001	AI	256	AI
3	D	73	0100 1001	ID	257	ID
4	E	68	0100 0100	DE	258	DE
5	blanc	69	0100 0101	E blanc	259	E blanc
6	T	32	0010 0000	blanc T	260	blanc T
7	O	84	0101 0100	TO	261	TO
8	I	79	0100 1111	OI	262	OI
9	blanc	73	0100 1001	I blanc	263	I blanc
10	L	32	0100 0000	blanc L	264	blanc L
11	E	76	0100 1100	LE	265	LE
12	blanc	*SP*	1111 1111	E blanc		
13	C	259	1 0000 0011	E blanc C	266	E blanc C
14	I	67	0 0100 0011	CI	267	CI
15	E	73	0 0100 1001	IE	268	IE
16	L	69	0 0100 0101	EL	269	EL
17	blanc	76	0 0100 1100	L blanc	270	L blanc
18	T			blanc T		
19	blanc	260	1 0000 0100	blanc T blanc	271	blanc T blanc
20	A	32	0 0010 0000	blanc A	272	blanc A
21	I			AI		
22	D	256	1 0000 0000	AID	273	AID
23	E			DE		
24	R	258	1 0000 0010	DER	274	DER
25	A	82	0 0101 0010	RA	275	RA
26		65	0 0100 0001			

Figure 8. Principe de compression à dictionnaire (LZW)

Au début de la compression, le dictionnaire est initialisé, et contient les 256 codes ASCII, c'est-à-dire qu'il renferme déjà les valeurs : A B C… Ainsi, l'algorithme ajoutera ses propres séquences à partir de l'adresse de la « case » 256.

Pour déceler les redondances de la séquence, le programme lit séquentiellement chaque octet un par un, et le place dans une mémoire tampon. Cet algorithme de compression fonctionne de manière itérative. Au départ, le programme met un caractère dans le buffer puis, si cette valeur est déjà contenue dans le dictionnaire, le programme met deux caractères dans

le buffer. Ainsi, à chaque fois que le buffer contient une valeur connue (exemple : étapes 12, 18, 21, 23), l'algorithme rajoute un caractère (exemple : étapes 19, 22, 24).. Dès que la valeur n'est pas déjà compactée dans le dictionnaire, le programme exécute plusieurs tâches : il ajoute premièrement cette valeur à la fin du tableau, il recopie dans le fichier l'adresse de la dernière séquence connue, et enfin, il recharge la mémoire tampon avec deux caractères. Il est important de noter qu'entre chaque itération, le programme n'écrit rien, d'où le gain de place (exemple : étapes 18, 21, 23).

c) Etude détaillée de l'exemple

Lors de la première étape, l'algorithme lit dans le fichier non-compressé, le caractère 'A', et l'ajoute dans la mémoire tampon. Comme le dictionnaire possède déjà tout le code ASCII, l'algorithme ajoute le caractère suivant, ce qui donne la valeur 'AI' dans le buffer. Puisque le dictionnaire ne contient pas cette valeur, le programme l'ajoute en fin de table, inscrit l'adresse de la valeur précédemment répertoriée (donc l'adresse de 'A', soit 65), et recharge le buffer du caractère suivant : 'D' (avec le caractère précédent :'I'). L'algorithme procède de la même manière ensuite.

Nous pouvons remarquer qu'à l'étape 12, le programme écrit un mot spécial '*SP'*. Cette valeur $(1111\ 1111)_2$ a une signification bien précise, elle indique un changement de codage. En effet, avant l'étape 12, les adresses retransmises dans le fichier compressé étaient codées sur 8 bits, mais pour coder des adresses supérieures à 255, il faut non plus 8, mais 9 bits. Cette information est capitale pour le décompresseur. Ce principe nous donne ainsi la possibilité de pouvoir agrandir infiniment le nombre de bits des adresses du dictionnaire, sans pour autant coder les premières cases avec des bits inutiles.

Voici une récapitulation des différents éléments du dictionnaire à la fin de la compression de la séquence (figure 9).

Dictionnaire

adresse	séquence
0...255	ascii 0...255
256	AI
257	ID
258	DE
259	E blanc
260	blanc T
261	TO
262	OI
263	I blanc
264	blanc L
265	LE
266	E blanc C
267	CI
268	IE
269	EL
270	L blanc
271	blanc T blanc
272	blanc A
273	AID
274	DER
275	RA

Figure 9. Constitution du dictionnaire après compression

d) Conclusion

Ce type de compression possède de nombreux avantages. Le principal étant la reconstruction du dictionnaire pendant la décompression, ce qui permet de ne pas le coder dans le ficher. Les formats les plus répandus utilisent en plus de leur propre algorithme, cette méthode de compression (notamment les formats GIF, TIFF…).

Cette méthode de compression peut obtenir des ratios supérieurs à 2 : 1, et ce, sans aucune altération de l'image.

5. Le mode de compression JPEG[14]

Le problème majeur de tous les codages précédents était que ces méthodes de compression ne délivraient de bon taux de compression que pour des images ciblées, c'est-à-dire que pour des images vectorielles ou pour des images monochromes bref, pas pour des photographies, ou toute autre image où les détails sont nombreux. Ce pourquoi, en 1982, une réunion entre un groupe d'experts en photographie et l'ITU-T abouti sur la création du JPEG (comité conjoint d'experts de la photographie).

[14] Joint Photographic Expert Group

A la différence du codage LZW, la compression JPEG est une compression avec pertes, ou destructrice, et qui par conséquent peut obtenir des ratios de 20 : 1 à 25 : 1, sans perte notable de qualité.

a) Principe de compression

Le principe de compression du JPEG est constitué de plusieurs étapes :

- la préparation
- la transformation
- la quantification
- l'encodage

Prenons l'exemple suivant : une image 640x480 RGB 24 bits par pixel.

→ La première étape consiste à transformer le codage de la couleur RGB, de l'image bitmap de départ, en un codage YIQ, avec les combinaisons linéaires vues dans la partie I.2 :

$$Y = 0.299\ R + 0.587\ G + 0.114\ B$$

$$I = 0.596\ R - 0.275\ G - 0.321\ B$$

$$Q = 0.212\ R - 0.523\ G + 0.311\ B$$

Mais pour gagner un peu de place, on arrondit généralement les coefficients de la luminance Y et de la chrominance (I et Q), pour obtenir :

$$Y = 0.3\ R + 0.59\ G + 0.11\ B$$

$$I = 0.6\ R - 0.28\ G - 0.32\ B$$

$$Q = 0.21\ R - 0.52\ G + 0.31\ B$$

Les résultats sont ensuite regroupés en matrice 640x480. Nous obtenons donc trois matrices de taille 640x480 représentant la luminance et la chrominance (2 matrices). Une des astuces du code JPEG est d'éliminer les variations de la chrominance entre deux pixels. En effet, l'œil de l'être humain n'est que peu sensible à ces minuscules variations. Aussi, l'algorithme rétrécit la taille des matrices de I et Q, en effectuant la moyenne des deux composantes de la chrominance, dans des carrés de 2x2 pixels. Nous obtenons alors des matrices de chrominance de 320x240.

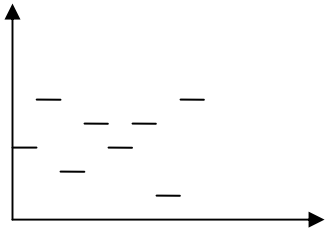
→ La deuxième action est ce qu'on appelle la transformation, ou DCT[15]. L'algorithme de compression découpe premièrement les matrices en blocs de 8x8 pixels, et leur applique ensuite la fonction DCT. Cette fonction DCT est une transformation en série (Fourier), qui délivre une représentation non plus spatiale, mais dans le domaine fréquentiel. En effet, la ligne et la colonne de la matrice représentent les axes X et Y dans le domaine spatial, et la valeur d'une case particulière représente la valeur de Z. Nous raisonnons donc en 3

[15] Discrete Cosinus Transform

dimensions. La transformation de la matrice donne une nouvelle matrice contenant cette fois-ci, les différentes puissances spectrales pour chaque fréquence. L'élément (0,0) représente la valeur moyenne du signal.

Pour mieux comprendre, étudions ce qu'il ce passe dans la première ligne d'une matrice 8x8.

- La variation de couleur nous donne ce signal :



- la transformation DCT en somme de fonctions sinusoïdales nous donne ceci :

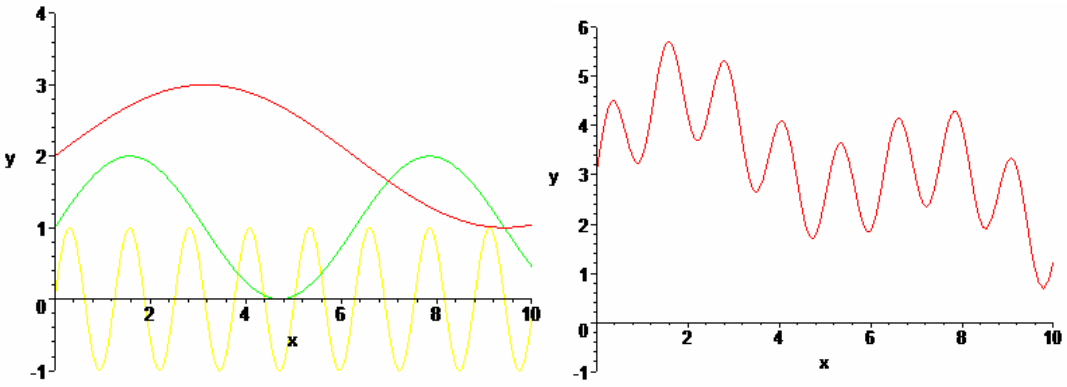


Figure 10. Fonctions sinusoïdales et la somme de celles-ci

- Ainsi, à la première case de la matrice correspond une fréquence particulière (la plus petite : en rouge), dont l'amplitude est déterminée par la valeur de cette case, et plus on s'éloigne de la première case, plus les fréquences sont élevées et plus les amplitudes sont réduites.

- En appliquant ces transformations en deux dimensions, nous obtenons une matrice carrée.

→ La quantification réside dans le faite que l'algorithme attribue à chaque fréquence, à chaque cellule de la matrice 8x8 pixels, un coefficient de perte. L'algorithme annulera ou diminuera les hautes fréquences qui, représentent les plus petits détails de l'image. L'atténuation de l'amplitude des différentes fréquences est déterminée par le ratio demandé par l'utilisateur.

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Tableau 3. exemple de coefficient d'atténuation

Cet exemple nous montre que les hautes fréquences sont atténuées de 64, alors que l'amplitude des basses fréquences (valeur moyenne de l'image) n'est pas modifiée (coefficient de 1). Chaque cellule sera donc divisée (et arrondit) par le coefficient de la case correspondante de la matrice de coefficients.

→ Enfin, la matrice obtenue est linéarisée en zigzag, selon le codage RLE (cf. chapitre II.1), et est compressée avec la méthode de Huffman (cf. chapitre II.3).

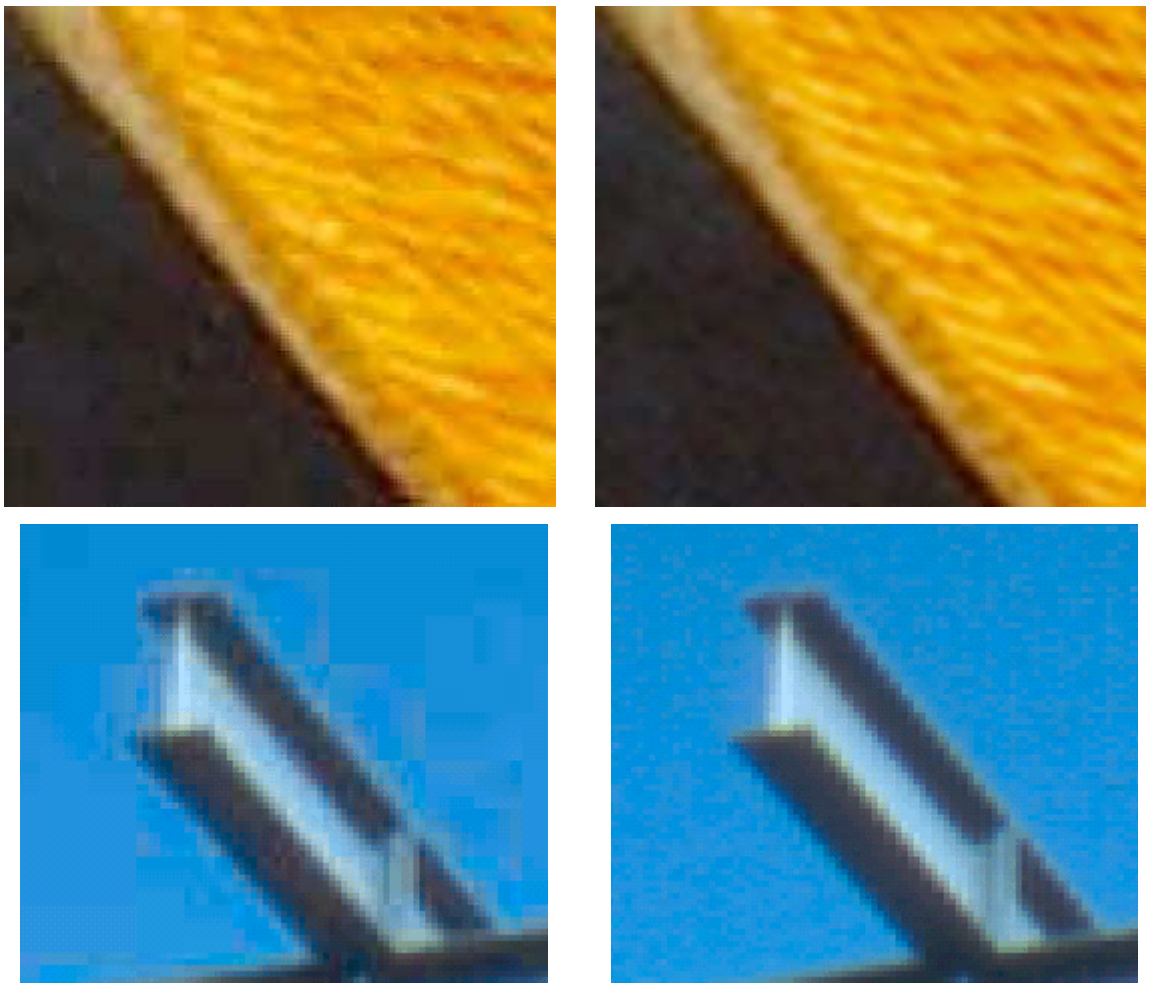


Figure 11. Comparaison figures compressées en JPEG - figures non compressées

La figure 11 montre deux images codées en jpeg (à gauche), selon un ratio moyen, et grossie plusieurs fois. Les images positionnées à droite sont les mêmes images que celles de gauche, mais sans compression.

Avec le grossissement de l'image, nous pouvons apercevoir les blocs de 8x8 pixels traités par l'algorithme du jpeg, dans la première image, et qui ne se retrouvent pas dans l'image de droite. De plus, dans la deuxième image de gauche, les bordures de l'objet sont moins nettes que celles présentent dans l'image de droite. Le JPEG élimine donc les plus petits détails de l'image, mais il tend aussi à mélanger le fond de l'image avec les bordures. Ainsi, cette compression altère dangereusement les textes, et les transforme en une simple « tâche » sur le fond.

La compression JPEG permet néanmoins d'obtenir de très bon taux de compression, surtout pour des images complexes. Il est à noté qu'il existe une variante du codage JPEG sans aucune perte, le lostless[16], qui est principalement utilisé dans l'imagerie médicale, et qui possède un ratio de 2 : 1, au maximum.

III. Les méthodes de compression récentes

1. La compression fractale

Dans les années 1950, un mathématicien nommé benoît Mandelbrot, découvrit ce qu'on appelle aujourd'hui, le fractales. La représentation graphique des solutions délivrées par un algorithme spécifique (cf. annexe n°2 p 34), donne une image constituée de répétitions infinies d'une même forme, d'un même motif. (cf. figure 12)

[16] Sans Perte

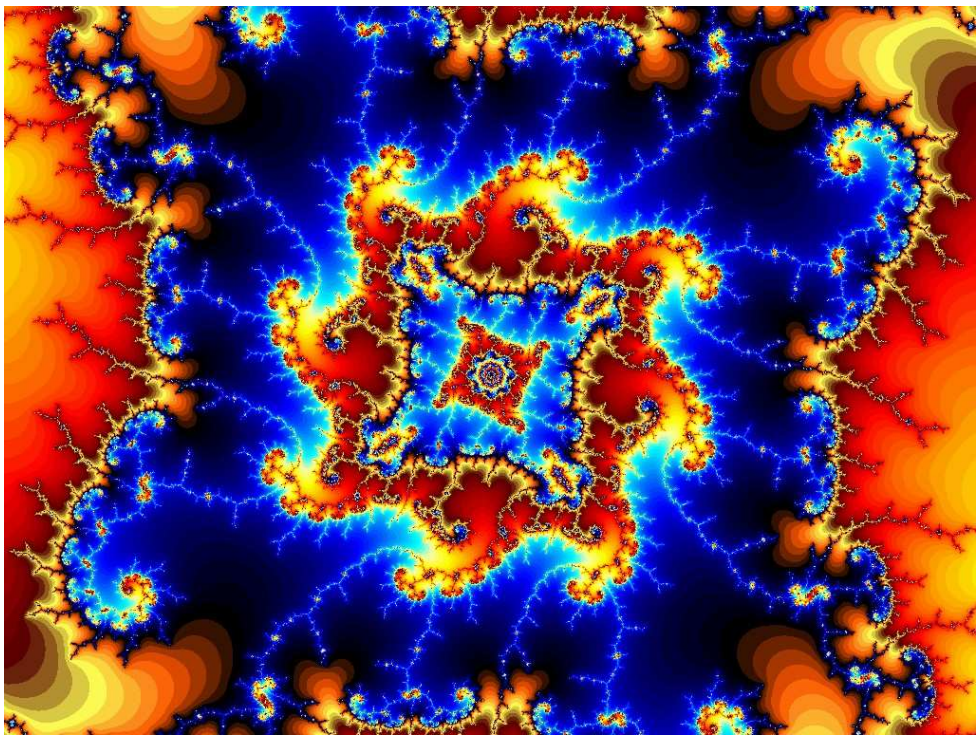


Figure 12. Fractale du type Julia

Puisqu'une image fractale est une répétition d'un motif plus ou moins rétréci, plus ou moins transformé (translation, rotation, homothéties…), Barnsley eût l'idée de rechercher dans n'importe quelle image, des formes similaires. Ainsi, au lieu de coder toute l'image, la compression fractale n'encode plus que le ou les motifs, ainsi que les transformations à réaliser.

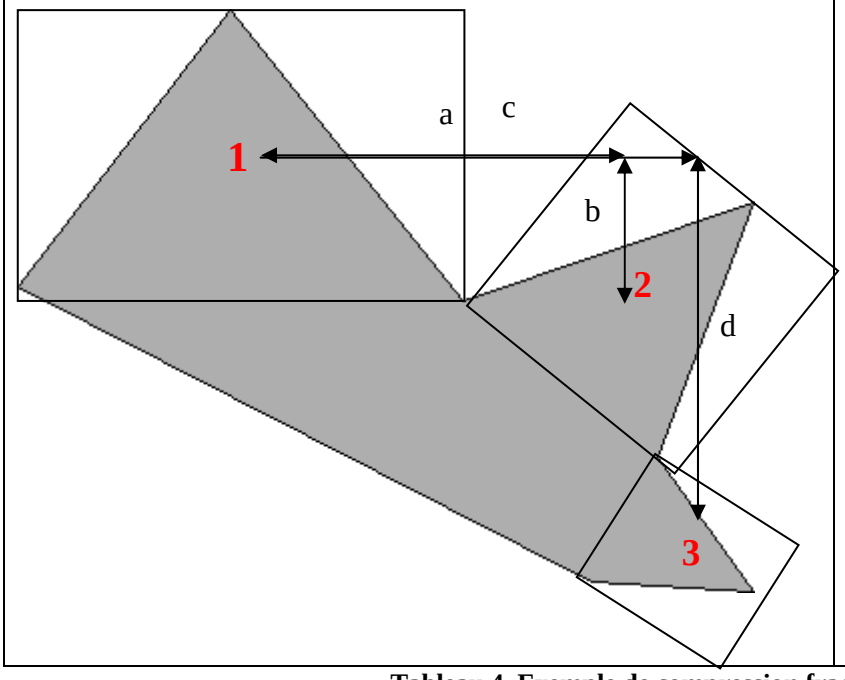
	1 : motif de base 2 : translation du motif de x=a et y=b, réduction de la taille de 1/3, et rotation du motif de 45°. 3 : translation du motif de x=c et y=d, réduction de la taille de 1/2, et rotation du motif de 100°.

Tableau 4. Exemple de compression fractale

En 1988, on annonçait des taux de compression allant jusqu'à 1/10 000. Mais ce taux de compression fabuleux a été réalisé en 100 heures de travail, avec un ordinateur biprocesseur (donc deux processeurs), et toute une équipe de chercheurs. Aujourd'hui, et malgré l'avancée considérable des microprocesseurs, la compression fractale peut compter sur des taux allant de 1/4 à 1/100, pour des temps raisonnables.

2. La compression par ondelettes

Il y a quelques mois encore, les chercheurs travaillaient à l'élaboration d'un nouveau format, le JPEG 2000, utilisant une nouvelle et prometteuse méthode de compression, appelée compression par ondelettes.

La théorie des ondelettes a été inventée par le mathématicien hongrois Alfred Haar dans les années 1910. Une ondelette est une transformation de fonction, comme Laplace ou Fourier, qui oscille principalement dans un intervalle restreint.

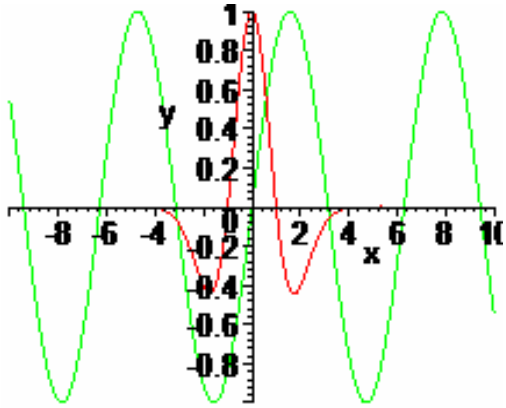


Figure 13. Représentation d'une sinusoïde et d'une ondelette

Nous pouvons voir sur la figure 13 que, contrairement à la fonction Cosinus (en vert), l'ondelette « Chapeau Mexicain » possède une amplitude variable, et est quasiment nulle en dehors de l'intervalle [-4,4]. Cette variation très locale de la fonction permet néanmoins de savoir précisément ce qui ce passe en n'importe quel endroit du signal original (non transformé).

Contrairement au format JPEG, qui décompose une image en blocs de 8x8 pixels, la compression JPEG 2000 transforme chaque ligne horizontale en un signal, qui sera ensuite transformé en somme d'ondelettes. En effet, la variation de couleur et d'intensité de chaque pixel d'une ligne peut être assimilé à la variation de deux signaux. Chaque signal sera ensuite directement transformé en une série d'ondelettes, répétées en différents endroits, et à différentes échelles, pour que la somme décrive le plus exactement le signal original (cf. figure 14). L'algorithme éliminera les variations les plus infimes pour compresser encore d'avantage l'image.

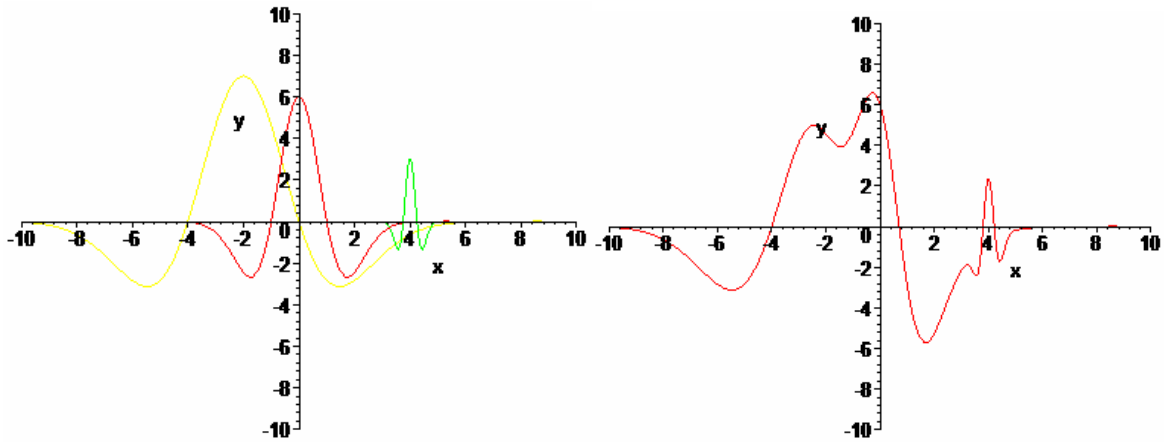


Figure 14. Exemple de fonctions ondelette et leur somme

Aujourd'hui, le format JPEG 2000 attend d'être officiellement validé par l'Organisation Internationale de Normalisation (ISO), pour se voir enfin utilisé dans tous les logiciels d'imagerie. Une variante de ce format a aussi vu le jour, le MJP2 (Motion JPEG 2000). Celui-ci utilise aussi la compression par ondelettes, mais pour des images en mouvement c'est-à-dire pour la vidéo numérique.

Conclusion

En somme, nous avons vu qu'il existait de nombreuses manières de coder, de représenter, de compresser des images. Il existe une multitude de formats pour compenser la diversité des images, différentes par leur dimension, leur nombre de couleur, leur méthode de représentation (vectorielle et bitmap). Chaque format est complémentaire des autres.

Aujourd'hui, avec la considérable avancée technologique, les algorithmes se permettent de réaliser de nombreux calculs pour compresser et décompresser une image. Les ratios devenant de plus en plus grand, sans pertes significative et réelle de la qualité de l'image.

Les dernières méthodes de compression (fractale et ondelette) sont particulièrement prometteuses. Elles montrent la nécessité d'associer à la programmation informatique, les mathématiques. L'avenir de la compression ne pourra s'opérer qu'au travers d'algorithmes mathématiques, et les progrès de la recherche mathématique entraîneront indubitablement une avancée dans la compression d'images et de fichiers.

Résumé

En informatique, une image du type bitmap est représentée par un tableau en deux dimensions, dont les valeurs contenues dans les cases (ou pixel) caractérisent la couleur du pixel. Tandis qu'une image du type vectorielle est représentée par des équations mathématiques du style z=f(x,y), avec z la couleur et x,y les composantes spatiales.

Il existe plusieurs codages pour symboliser la couleur, le plus connu étant le RGB (ou RVB : Rouge, Vert, Bleu), qui code une couleur selon ses trois composantes rouge, bleu et vert, ou le codage YUV utilisé par les formats PAL et SECAM et qui code la couleur selon la luminance et la chrominance.

On distingue deux types de compression, la compression non destructrice qui, ne modifie en rien la qualité de l'image, et la compression dite destructrice qui elle élimine les plus insignifiantes données, comme notamment un changement rapide de chrominance entre deux pixels. Certains algorithmes (comme la compression de Huffman et LZW) réalisent un arbre ou un dictionnaire afin d'y ranger les redondances les plus fréquentes dans l'image. D'autres se chargent de repérer les formes qui se reproduisent comme le codage par fractale ou RLE.

Afin, un autre format, le JPEG, découpe l'image en bloc de 8x8 pixels pour le JPEG et en lignes pour le JPEG 2000, puis réalise une transformation mathématique et change la variation de couleur en somme de fonctions sinusoïdales pour l'un et en somme d'ondelettes pour l'autre.

Environ 237 mots

Table des illustrations

Figures :

Tableaux :

Bibliographie

- œuvres et revues :

- Dictionnaire Encyclopédique Larousse, Larousse, 2002.

- Matthieu Crocq, Science et vie , page 92 à 95, N°1016 Mai 2002.

- sites Internet :

- Piscart et Olivier Van Muysewinkel, « Compression d'une image - Norme JPEG », , consulté le 20/09/2003.

- « Les compressions format par format », http://membres.lycos.fr/compressions/formats.html, consulté le 20/09/2003.

- « Ces logiciels qui transforment vos fichiers en liliputiens », http://www.fmv.ulg.ac.be/jlc/img/others/comp.html, consulté le 20/09/2003.

- « Démonstration du JPEG 2000 », http://www.cmla.ens-cachan.fr/Utilisateurs/ymeyer/jpeg2000Demo/jpeg2Dem.html, consulté le 20/09/2003.

- Yannick Pinson, « La compression par ondelette », http://www.image-etc.com/faq/wavelet/index.htm, consulté le 20/09/2003.

- Jean-François Pillou , « Compression d'image », http://www.commentcamarche.net/video/compimg.php3, consulté le 20/09/2003.

- Jacques WEISS, « Introduction au codage des images en sous-bandes » , http://www.supelec-rennes.fr/ren/perso/jweiss/wavelet/intro.htm, consulté le 20/09/2003.

- Stéphane Desplanques et Philippe MERLE, « La compression d'image : algorithme LZW », http://www-ensimag.imag.fr/cours/Exposes.Reseaux/Compression/page5.html, consulté le 20/09/2003.

- Mark Nelson, « LZW », http://dogma.net/markn/articles/lzw/lzw.htm, consulté le 20/09/2003.

- Julien Michot, « Galeries de fractales », http://www.fractals.fr.fm, consulté le 20/09/2003.

Annexe 1

Exemple d'algorithme en C de compression LZW

```
/*************************************************************
**
** Copyright (c) 1989 Mark R. Nelson
**
** LZW data compression/expansion demonstration program.
**
** April 13, 1989
**
*************************************************************************/
#include <stdio.h>
#define BITS 12                   /* Setting the number of bits to 12, 13*/
#define HASHING_SHIFT BITS-8      /* or 14 affects several constants.    */
#define MAX_VALUE (1 << BITS) - 1 /* Note that MS-DOS machines need to   */
#define MAX_CODE MAX_VALUE - 1    /* compile their code in large model if*/
                                  /* 14 bits are selected.               */
#if BITS == 14
  #define TABLE_SIZE 18041        /* The string table size needs to be a */
#endif                            /* prime number that is somewhat larger*/
#if BITS == 13                    /* than 2**BITS.                       */
  #define TABLE_SIZE 9029
#endif
#if BITS <= 12
  #define TABLE_SIZE 5021
#endif

void *malloc();

int *code_value;                  /* This is the code value array        */
unsigned int *prefix_code;        /* This array holds the prefix codes   */
unsigned char *append_character;  /* This array holds the appended chars */
unsigned char decode_stack[4000]; /* This array holds the decoded string */

/*************************************************************
**
** This program gets a file name from the command line.  It compresses the
** file, placing its output in a file named test.lzw.  It then expands
** test.lzw into test.out.  Test.out should then be an exact duplicate of
** the input file.
**
*************************************************************************/

main(int argc, char *argv[])
{
FILE *input_file;
FILE *output_file;
FILE *lzw_file;
char input_file_name[81];

/*
**  The three buffers are needed for the compression phase.
*/
  code_value=malloc(TABLE_SIZE*sizeof(unsigned int));
  prefix_code=malloc(TABLE_SIZE*sizeof(unsigned int));
  append_character=malloc(TABLE_SIZE*sizeof(unsigned char));
  if (code_value==NULL || prefix_code==NULL || append_character==NULL)
  {
    printf("Fatal error allocating table space!\n");
    exit();
  }
/*
** Get the file name, open it up, and open up the lzw output file.
*/
  if (argc>1)
    strcpy(input_file_name,argv[1]);
```

```
  else
  {
    printf("Input file name? ");
    scanf("%s",input_file_name);
  }
  input_file=fopen(input_file_name,"rb");
  lzw_file=fopen("test.lzw","wb");
  if (input_file==NULL || lzw_file==NULL)
  {
    printf("Fatal error opening files.\n");
    exit();
  };
/*
** Compress the file.
*/
  compress(input_file,lzw_file);
  fclose(input_file);
  fclose(lzw_file);
  free(code_value);
/*
** Now open the files for the expansion.
*/
  lzw_file=fopen("test.lzw","rb");
  output_file=fopen("test.out","wb");
  if (lzw_file==NULL || output_file==NULL)
  {
    printf("Fatal error opening files.\n");
    exit();
  };
/*
** Expand the file.
*/
  expand(lzw_file,output_file);
  fclose(lzw_file);
  fclose(output_file);

  free(prefix_code);
  free(append_character);
}

/*
** This is the compression routine.  The code should be a fairly close
** match to the algorithm accompanying the article.
**
*/

compress(FILE *input,FILE *output)
{
unsigned int next_code;
unsigned int character;
unsigned int string_code;
unsigned int index;
int i;

  next_code=256;              /* Next code is the next available string code*/
  for (i=0;i<TABLE_SIZE;i++)  /* Clear out the string table before starting */
    code_value[i]=-1;

  i=0;
  printf("Compressing...\n");
  string_code=getc(input);    /* Get the first code                         */
/*
** This is the main loop where it all happens.  This loop runs util all of
** the input has been exhausted.  Note that it stops adding codes to the
** table after all of the possible codes have been defined.
*/
  while ((character=getc(input)) != (unsigned)EOF)
  {
    if (++i==1000)                        /* Print a * every 1000    */
    {                                     /* input characters.  This */
      i=0;                                /* is just a pacifier.     */
```

```
      printf("*");
    }
    index=find_match(string_code,character);/* See if the string is in */
    if (code_value[index] != -1)            /* the table.  If it is,   */
      string_code=code_value[index];        /* get the code value.  If */
    else                                    /* the string is not in the*/
    {                                       /* table, try to add it.   */
      if (next_code <= MAX_CODE)
      {
        code_value[index]=next_code++;
        prefix_code[index]=string_code;
        append_character[index]=character;
      }
      output_code(output,string_code);  /* When a string is found  */
      string_code=character;            /* that is not in the table*/
    }                                   /* I output the last string*/
  }                                     /* after adding the new one*/
/*
** End of the main loop.
*/
  output_code(output,string_code); /* Output the last code               */
  output_code(output,MAX_VALUE);   /* Output the end of buffer code      */
  output_code(output,0);           /* This code flushes the output buffer*/
  printf("\n");
}

/*
** This is the hashing routine.  It tries to find a match for the prefix+char
** string in the string table.  If it finds it, the index is returned.  If
** the string is not found, the first available index in the string table is
** returned instead.
*/

find_match(int hash_prefix,unsigned int hash_character)
{
int index;
int offset;

  index = (hash_character << HASHING_SHIFT) ^ hash_prefix;
  if (index == 0)
    offset = 1;
  else
    offset = TABLE_SIZE - index;
  while (1)
  {
    if (code_value[index] == -1)
      return(index);
    if (prefix_code[index] == hash_prefix &&
        append_character[index] == hash_character)
      return(index);
    index -= offset;
    if (index < 0)
      index += TABLE_SIZE;
  }
}

/*
**  This is the expansion routine.  It takes an LZW format file, and expands
**  it to an output file.  The code here should be a fairly close match to
**  the algorithm in the accompanying article.
*/

expand(FILE *input,FILE *output)
{
unsigned int next_code;
unsigned int new_code;
unsigned int old_code;
int character;
int counter;
unsigned char *string;
char *decode_string(unsigned char *buffer,unsigned int code);
```

```
  next_code=256;           /* This is the next available code to define */
  counter=0;               /* Counter is used as a pacifier.            */
  printf("Expanding...\n");

  old_code=input_code(input);  /* Read in the first code, initialize the */
  character=old_code;          /* character variable, and send the first */
  putc(old_code,output);       /* code to the output file                */
/*
**  This is the main expansion loop.  It reads in characters from the LZW file
**  until it sees the special code used to inidicate the end of the data.
*/
  while ((new_code=input_code(input)) != (MAX_VALUE))
  {
    if (++counter==1000)   /* This section of code prints out     */
    {                      /* an asterisk every 1000 characters   */
      counter=0;           /* It is just a pacifier.              */
      printf("*");
    }
/*
** This code checks for the special STRING+CHARACTER+STRING+CHARACTER+STRING
** case which generates an undefined code.  It handles it by decoding
** the last code, and adding a single character to the end of the decode string.
*/
    if (new_code>=next_code)
    {
      *decode_stack=character;
      string=decode_string(decode_stack+1,old_code);
    }
/*
** Otherwise we do a straight decode of the new code.
*/
    else
      string=decode_string(decode_stack,new_code);
/*
** Now we output the decoded string in reverse order.
*/
    character=*string;
    while (string >= decode_stack)
      putc(*string--,output);
/*
** Finally, if possible, add a new code to the string table.
*/
    if (next_code <= MAX_CODE)
    {
      prefix_code[next_code]=old_code;
      append_character[next_code]=character;
      next_code++;
    }
    old_code=new_code;
  }
  printf("\n");
}

/*
** This routine simply decodes a string from the string table, storing
** it in a buffer.  The buffer can then be output in reverse order by
** the expansion program.
*/

char *decode_string(unsigned char *buffer,unsigned int code)
{
int i;

  i=0;
  while (code > 255)
  {
    *buffer++ = append_character[code];
    code=prefix_code[code];
    if (i++>=4094)
    {
```

```
      printf("Fatal error during code expansion.\n");
      exit();
    }
  }
  *buffer=code;
  return(buffer);
}

/*
** The following two routines are used to output variable length
** codes.  They are written strictly for clarity, and are not
** particularyl efficient.
*/

input_code(FILE *input)
{
unsigned int return_value;
static int input_bit_count=0;
static unsigned long input_bit_buffer=0L;

  while (input_bit_count <= 24)
  {
    input_bit_buffer |=
        (unsigned long) getc(input) << (24-input_bit_count);
    input_bit_count += 8;
  }
  return_value=input_bit_buffer >> (32-BITS);
  input_bit_buffer <<= BITS;
  input_bit_count -= BITS;
  return(return_value);
}

output_code(FILE *output,unsigned int code)
{
static int output_bit_count=0;
static unsigned long output_bit_buffer=0L;

  output_bit_buffer |= (unsigned long) code << (32-BITS-output_bit_count);
  output_bit_count += BITS;
  while (output_bit_count >= 8)
  {
    putc(output_bit_buffer >> 24,output);
    output_bit_buffer <<= 8;
    output_bit_count -= 8;
  }
}
```

Annexe 1 – ***(Propriété de Mark R. Nelson)***

Annexe 2

Exemple de programme en C de réalisation de fracales

```
/*****************************************************************************
                                    FRACTALES
Auteur: Michot Julien
Créé le 06-04-03
Disponible sur www.fractals.fr.fm ou sur www.cppfrance.com

Julia :
      z=z*z+complex(0.3,-0.3)
Mandelbrot :
      z=z*z+pixel

*****************************************************************************/

#include <iostream.h>
#include <complex.h>
#include <graphics.h>
#include <conio.h>
#include <dos.h>

#define MANDELBROT        z=z*z+pixel
#define JULIA             z=z*z+complex(0.3,-0.3)     //ou (-.1,-.7)
#define ITERATION  50
#define MAX_VALUE         128
#define ADRESSE  "C:\\TCLITE\\BGI"     // metre l'adresse du dossier BGI de votre compilateur



void Initialise(void);
void Couleur(complex z,long int x, long int y,int o);
void Parametres(float Choix[]);

void main(void)
{

float Choix[6];
double i,j;
float Xmin,Xmax,Ymin,Ymax,Type,colorr;
long int x,y;
```

```
int stop=1,o,teste;
long int val;
int iter=1;
int iter_max=ITERATION;
int cblanc=0;
float bailout=MAX_VALUE;
complex z=complex(0,0);
complex pixel=complex(0,0);

Initialise();
restorecrtmode();

do
{
z=complex(0,0);
Parametres(Choix);

Xmin=Choix[0];
Xmax=Choix[1];
Ymin=Choix[2];
Ymax=Choix[3];
Type=Choix[4];
colorr=Choix[5];

setgraphmode(getgraphmode());

if ( Choix[4]==1) o=100;
else { o=1; bailout=1;}

stop=1;
outtextxy(0,0,"Please press a key...");
getch();

for(i=Xmin;i<=Xmax&&stop;i+=(Xmax-Xmin)/800)
  for(j=Ymin;j<=Ymax&&stop;j+=(Ymax-Ymin)/640)
  {
  x=(i-Xmin)*800/(Xmax-Xmin);
  y=(j-Ymin)*640/(Ymax-Ymin);
  z= complex(i,j);
  for(iter=1;iter<=iter_max&&stop;iter++)
  {
  pixel=complex(i,j);
```

```
  if( Choix[4] ==2 ) JULIA;
  else MANDELBROT;

  if( norm(z)>bailout || kbhit() ){ stop=0; cblanc=1;}
  if( norm(z)<=0.0001) { stop=0; }
    }

  val=norm(z)*1000;

 if(cblanc)  {
 if (!colorr) putpixel(x,y,WHITE);
 else Couleur(z,x,y,o);
 }
 else              putpixel(x,y,BLACK);
 cblanc=0;
  if(kbhit()) stop=0; else stop=1;
  }
  getch();
  setcolor(BLACK);
  cleardevice();
  restorecrtmode();
  clrscr();
  cout << "Voulez-vous continer ?\n 1->OUI\n 0->NON\n";
  teste=getch();
} while ( teste!='0');
closegraph();
}

void Initialise(void)
{
int GraphPilote= DETECT;
int GraphMode ;
initgraph(&GraphPilote,&GraphMode, ADRESSE);
}

void Couleur(complex z,long int x,long int y,int o)
{
 double val=norm(z);
 double i;
 int Nb=0;
```

```
  int
Tab[]={WHITE,YELLOW,LIGHTCYAN,CYAN,LIGHTBLUE,BLUE,LIGHTMAGENTA,MAGENTA,LIGH
TRED,RED,LIGHTGREEN,GREEN,BROWN,LIGHTGRAY,DARKGRAY};

  for (i=0;i<=val;i+=o*(Nb+1)) Nb++;
  putpixel(x,y,Tab[Nb]);
}

void Parametres(float Choix[])
{
int a=0;
do{
clrscr();
cout << "\t\t\t    ---- FRACTALES ----" <<endl;
cout << "\t\t\t\t\t\t\t\t    par  MiTcH"<<endl;
cout << "Quelle fractale voulez-vous ?\n 1 - Mandelbrot\n 2 - Julia"<<endl<<" ";
cin >> Choix[4];
cout << "Voulez-vous de la couleur ?\n 1-Oui\n 0-Non"<<endl<<" ";
cin >> Choix[5];
cout << "Voulez-vous changer la fenetre d'affichage ?\n 1-Oui\n 0-Non"<<endl<<" ";
cin >> a;
if (a) {
cout << endl<< "Fenetre :\nXmin=";
cin >> Choix[0];
cout << "Xmax=";
cin >> Choix[1];
cout << "Ymin=";
cin >> Choix[2];
cout << "Ymax=";
cin >> Choix[3];
} else {
Choix[0]=-1.5;
Choix[1]=1.5;
Choix[2]=-1.256;
Choix[3]=1.25;
}
} while ( !Choix[4] );
}
```

Annexe 2 – *(Propriété de Julien Michot)*