

Codage et compression d'images et de vidéo

Pierre Nerzic, pierre.nerzic@univ-rennes1.fr

automne/hiver 2007

Table des matières

Semaine 11 : Images non compressées	2
Présentation	2
Images	2
Images raster et pixels	4
Espaces de couleur	5
Couleurs indexées et palettes	6
Résolution et définition	8
Formats de fichier	8
Structure d'un fichier image	8
Formats PAM, PPM, PNM, PGM et PBM : librairie netpbm	9
Description des formats PNM	9
Commandes de traitement d'images PNM	10
Exemple de programme PNM	11
Description de quelques fonctions utiles	13
Format BMP	14
Contenu d'un fichier BMP	15
Exemples	15
Format TIFF non compressé	16
Semaine 12 : Compression de données (partie 1)	17
Concepts de la théorie de l'information	17
Définition d'un signal	17
Recodage de source	18
Source markovienne	18
Probabilités d'apparition et information	19
Entropie d'un signal	20
Extension d'une source	21
Codage d'une source	22
Algorithmes de compression	22
Méthode RLE	22
La méthode RLE appliquée aux images	23
Idées	23
Décomposition en plans bitmap	23
Codes de Gray	27
Semaine 13 : Codages entropiques	31
Algorithmes entropiques = basés sur les fréquences d'apparition des symboles...	31
Algorithme de Shannon-Fano	31
VLC préfixé	31
Arbre binaire de codes préfixés	32
Algorithme de Shannon-Fano	32
Exemple	33
Même exemple avec application de la note 1	34
Entropie de ce message	34
Algorithme de Huffman	34
Exemple de codage Huffman	34
Remarques finales sur ces deux algorithmes	35

Algorithme Lempel-Ziv 77	35
Principe de LZ77	36
Codage des éléments	36
Déroulement de LZ77 sur un exemple	37
Autre exemple plus intéressant	37
Un applet pour illustrer l'algorithme LZ77	38
Algorithme LZW : une amélioration de LZ77	38
Description de LZW	38
Ecriture des codes	40
Déroulement de LZW sur un exemple	40
Déroulement de LZW sur un autre exemple	41
Décodage d'un message LZW	43
Décodage d'un message avec l'exception	43
Une démonstration de l'algorithme	44
Programmation optimisée du codeur et décodeur LZW	44
Semaine 14 : Principes des formats GIF et PNGs	47
Le format GIF	47
Concepts	47
Structure des fichiers GIF	47
Exemple de fichier GIF	48
Algorithme LZW modifié pour GIF	48
Le format PNG	49
Concepts	49
Exemple de fichier png	50
Compression	51
Commentaires sur ce format	51
Semaine 15 : Etude du format JPG	52
Concepts de la compression JPEG	52
Séparation de l'image en macroblocs	53
Séparation YCrCb et sous-échantillonnage	53
Transformation DCT	58
Illustration du calcul de la DCT	59
Transformée inverse iDCT	61
Intérêt de la DCT pour la compression d'images	61
Spectre DCT	62
Bilan	64
Quantification des coefficients	65
Quantification par zonage	66
Quantification par seuil	66
Quantification par nombre	66
Quantification par matrice	66
Encodage des coefficients DCT	67
Codage des coefficients DC0 des différents blocs	67
Codage des coefficients AC0...AC63	67
Modes de transmission de l'image	68
Avenir : Jpeg2000	68
Semaine 16 : Quelques informations sur le codage de vidéos	69

Concepts de la compression vidéo	69
Réflexions sur la compression de données	69
Méthodes de compression	69
Compression MJPEG et MJPEG2000	70
Groupes d'images	70
Exemple de vidéo MPEG	71
Représentation en fichier	72
Normes	73
Fichiers AVI	74
Structure d'un fichier AVI	74
Programmation avec DirectShow	77
Lecture d'un fichier AVI	78
Création d'un fichier AVI	80
AVI sur Linux	80

Ce document a été généré par FOP 0.94 (<http://xmlgraphics.apache.org/fop/>) à partir d'un document XML créé avec Jaxe 2.4.1 (<http://jaxe.sourceforge.net/Jaxe.html>) et d'un ensemble de feuilles de style et de transformations personnelles. L'ensemble est lancé par un script Python (<http://www.python.org/>)

Codage et compression d'images et de vidéo

Prérequis

arithmétique, algorithmique.

Objectifs

Ce cours explique les concepts de la compression et du codage des images dans des fichiers.

- comprendre comment les images et les vidéos sont compressées et stockées dans les fichiers.
- savoir utiliser des bibliothèques d'accès aux données multimédia.

NB : ce cours est destiné à des étudiants en 2e année de DUT informatique, spécialisation imagerie numérique. Il manque volontairement de rigueur mathématique afin de privilégier l'aspect pédagogique et la compréhension des concepts.

Calendrier du module

6	semaines
1	h CM par semaine
1	h TD par semaine
2	h TP par semaine

Plan du cours

- a. Représentation des images
- b. Formats d'images PNM, BMP et TIFF non compressés
- c. Eléments de la théorie de l'information
- d. Algorithmes de compression de données
- e. Formats GIF et PNG
- f. Compression par corrélations spatiales
- g. Formats JPG
- h. Algorithmes de compression de la vidéo
- i. Formats MPEG et AVI

Semaine 11 (21/11/2011) : Images non compressées

Présentation

Ce cours présente quelques concepts de la représentation des images et des vidéos dans un ordinateur et leur stockage dans des fichiers. On va commencer par les modes de représentation des images (raster ou vectorielles). Puis on étudiera quelques formats de fichiers pour des images non compressées (PPM et BMP). Ensuite on étudiera la théorie mathématique de la compression de données et comment on l'applique à des images (GIF et PNG). On constatera que la compression n'est encore pas très bonne et on se dirigera vers des méthodes ayant de meilleurs résultats (JPG et JP2).

Images

On s'intéresse aux images 2D, planes et rectangulaires (cas le plus classique). Il existe principalement deux méthodes pour les définir :

image "raster" ou matricielle

Une telle image est un tableau rectangulaire de pixels qu'on appelle aussi pixmap ou bitmap. C'est le type d'image le plus répandu sur les ordinateurs actuels parce que la technologie des écrans facilite l'affichage de telles images : les écrans sont eux-mêmes des tableaux rectangulaires de pixels. Beaucoup d'imprimantes fonctionnent aussi en mode raster : la tête d'impression effectue un balayage haut-bas du papier et imprime toute une ligne de pixels à la fois.

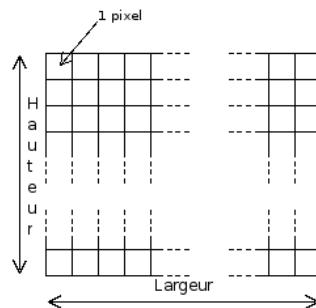
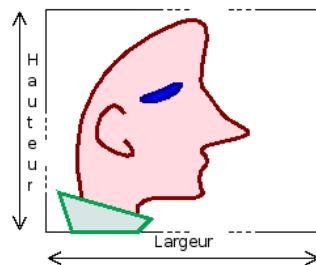


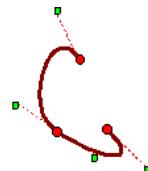
image vectorielle

Une telle image est une liste de figures 2D : segments de droite, polygones, coniques, splines, bezier... remplies ou non de couleurs unies, tramées, dégradées... L'affichage d'une image vectorielle nécessite un dispositif spécifique : écran cathodique dont le faisceau d'électron peut être dirigé à volonté en X et Y, ou table traçante. Ou alors il faut effectuer une rasterisation. Dans tous les cas, il faut réellement dessiner l'image en effectuant les tracés des figures. Voir [ce lien](http://fr.wikipedia.org/wiki/Image_vectorielle) (http://fr.wikipedia.org/wiki/Image_vectorielle) pour quelques informations de plus.



Cette image est construite à partir de 4 éléments : un polygone vert, deux courbes de Béziers fermées et une courbe de Béziers ouverte. Une courbe de báziers est une courbe continue et dérivable deux fois qui passe par des points précis appelés points de contrôle et avec une certaine "tension" = une sorte de force définie aux points de contrôle qui donne la forme de la

courbe, voir le cours de maths de S4P4 sur l'interpolation. L'image entière est spécifiée par très peu d'informations : les coordonnées des points de contrôle et la nature des tracés à faire (voir le TP3 de S3P2). Voici comment l'oreille du personnage est définie : trois points de contrôle en rouge et trois tangentes en pointillés.



On peut modifier extrêmement facilement la taille du dessin, en déplaçant les points de contrôle. D'autres opérations sont également extrêmement simples : changer les couleurs, faire des transformations géométriques : rotations, miroir, déformations linéaires. Par contre, on peut noter que les déformations non linéaires ne sont pas du tout simples à réaliser, par exemple les [anamorphoses](http://fr.wikipedia.org/wiki/Anamorphose) (<http://fr.wikipedia.org/wiki/Anamorphose>).

On peut convertir une image d'un type à l'autre : on peut rasteriser une image vectorielle et on peut vectoriser une image de type pixmap. Mais il faut noter que cela fait généralement perdre de l'information dans les deux sens. Une image vectorielle qu'on a vectorisé devient granuleuse : les contours des figures doivent se plier aux pixels et forment des marches d'escalier. Quant à la vectorisation d'une image raster, on n'obtient pas facilement un résultat correct, en particulier si la résolution (le nombre de pixel) est trop faible il devient difficile d'identifier des figures géométriques régulières.

Voici une image vectorielle qui a été agrandie peu à peu : elle reste lisse mais aucun détail supplémentaire n'apparaît (attention, comme votre écran est matriciel, il a fallu rasteriser l'image vectorielle, donc les deux images sont matricielles, simplement celle de gauche peut être agrandie ou réduite tant qu'on veut). Par contre, une image matricielle se pixelise.

image vectorielle agrandie peu à peu	image matricielle agrandie peu à peu

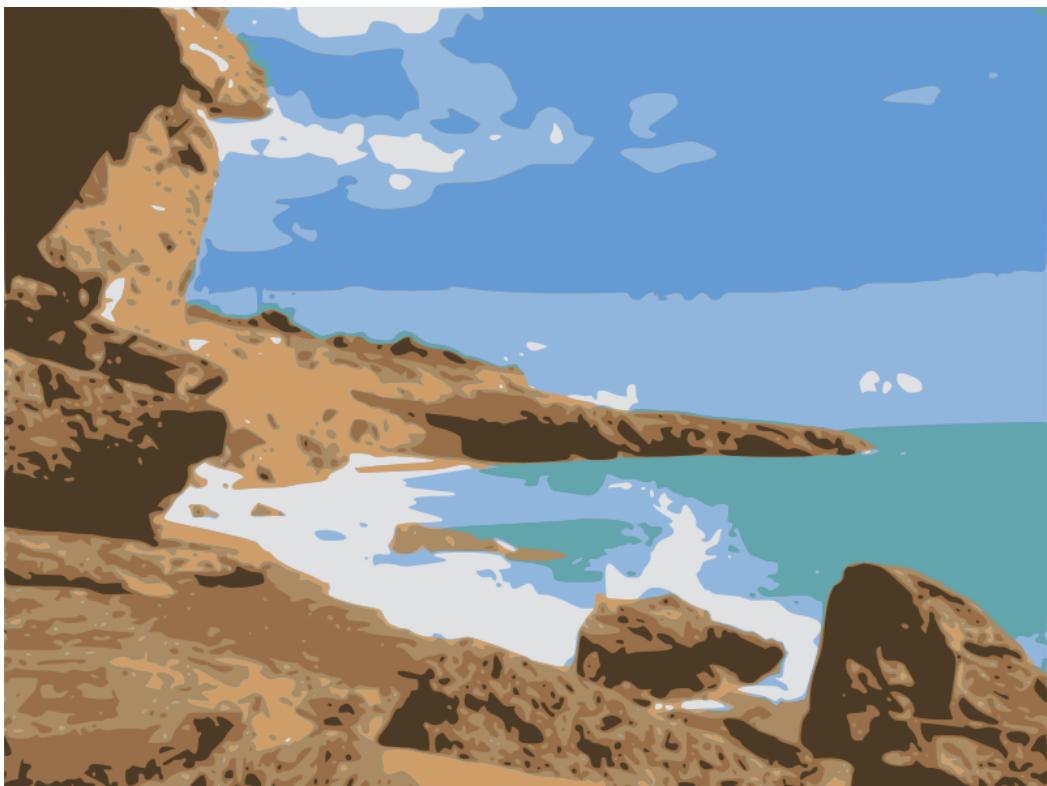
De ce comparatif, on pourrait déduire à tort que les images vectorielles sont meilleures que les images matricielles. Il n'en est rien : une image vectorielle est intéressante parce qu'elle est indépendante de la résolution d'affichage. Elle ne contient que des primitives de dessin spécifiées de manière relative à l'image entière (rectangle bleu de 10% de la largeur et 20% de la hauteur...).

Lorsqu'on a affaire à des images "réelles", on ne peut pratiquement pas vectoriser : on peut difficilement les réduire à des formes géométriques régulières (donc facile à modéliser par quelques informations). Dans ce cas, seule l'image matricielle est économique.

Par exemple, ci-dessous, cette image matricielle montre qu'on perd de la qualité avec la vectorisation. On est obligé de transformer des groupes de pixels en figures géométriques telles que des courbes de Béziers fermées. Par contre, on peut se permettre des effets impossibles avec les images bitmaps.

image matricielle	image vectorisée avec inkscape et potrace	image vectorielle transformée

On pourrait penser que l'image centrale résulte d'une simple réduction des couleurs, ce n'est pas le cas : le logiciel essaie de tracer des zones géométriques, définies par des splines dont les pixels sont à peu près tous identiques. De cette façons, l'image vectorisée est agrandissable sans changement de qualité (qui reste toutefois faible), ainsi que le montre l'image ci-dessous.



Images raster et pixels

Les pixels sont les éléments de base des images raster. Selon le type d'image, on aura des pixels :

en noir ou blanc

Les pixels sont représentés par 0 ou 1, ça dépend du codage, en général 0=noir, 1=blanc mais ça peut être l'inverse. Une telle image est appelée bitmap.

en niveaux de gris

Un nombre variant entre 0 et un maximum indique la nuance de gris à utiliser. La norme de codage indique la signification des bornes : en général, 0 = noir, $2^n - 1$ = blanc, pour n=4..16.

Un n plus grand est inutile pour l'oeil humain qui ne distingue déjà pas autant de nuances et

un n plus petit rend l'image très peu agréable à regarder. Une telle image est appelée parfois graymap.

en couleur

La couleur d'un pixel est définie dans un espace de couleur, par exemple RVB, LRVB, CMJK, HLS, YUV... Une telle image est appelée pixmap.

Espaces de couleur

Les espaces de couleurs sont des normes pour représenter les couleurs. Par exemple, on peut se baser sur la théorie des couleurs de Newton (décomposition en primaires rouge, vert, bleu) ou bien sur des techniques de compression des données.

Les espaces de couleurs sont des normes pour représenter les couleurs. Par exemple, on peut se baser sur la théorie des couleurs de Newton (décomposition en primaires rouge, vert, bleu) ou bien sur des techniques de compression des données. La volonté est de réduire toute couleur à très peu d'informations, au contraire des peintres et des artistes qui décrivent chaque couleur avec énormément de mots. Par exemple, telle nuance de jaune qu'un peintre nommera "Fleur de soufre" (15 octets) pourra être représentée par le nombre à 24 bits (3 octets) 0xFFFF6B. On étudiera plus en détail ces espaces de couleur en période S4P4.

Il existe de nombreux codages des pixels couleur pour modéliser la perception et la restitution de la couleur. Pour résumer, le problème est de parvenir à modéliser toutes les nuances que voit l'oeil. On pourra lire [ce document](http://xavier.hubaut.info/coursmath/doc/thcoul.htm) (<http://xavier.hubaut.info/coursmath/doc/thcoul.htm>) et se référer à [celui-ci](http://fr.wikipedia.org/wiki/Couleur) (<http://fr.wikipedia.org/wiki/Couleur>) ou lire le cours de S4P4. En général, on se contente de trois axes de couleurs dites élémentaires, mais ce n'est pas suffisant : certaines nuances que distingue l'oeil ne peuvent être modélisées et certaines teintes modélisables ne sont pas distinguées par l'oeil à cause des imperfections de la nature. Pour les cours de S3P3 relatifs à la compression de données, on utilisera les systèmes RGB et YUV sans se poser trop de questions.

Le système YUV consiste à séparer la notion de couleur en deux concepts : intensité lumineuse et teinte de la lumière. Par exemple, ces deux rectangles sont de même teinte (verte), mais leur intensité lumineuse est différente (sombre, clair). L'échantillon de gauche émet moins de photons que l'échantillon de droite, mais ces photons sont de même longueur d'onde.



coloration identique, intensité différente

Et ces deux rectangles sont de même intensité, mais ils n'ont pas la même coloration. Il y a la même quantité d'énergie lumineuse, mais les longueurs d'ondes (le spectre) sont légèrement différentes : un peu plus de rouge dans l'échantillon de droite.



coloration différente, intensité identique

L'intensité lumineuse est appelée luminance et représentée par la lettre Y ou L. C'est un nombre variant entre 0 et 255 dans les systèmes les plus courants (HSV, YUV, LRVB).

La coloration est plus complexe à représenter. On considère, après des expérimentations, qu'il faut au moins deux nombres. Selon les systèmes, on peut avoir : teinte et saturation, ou bien axe rouge et axe bleu ou ... Il existe de multiples possibilités qui seront examinées en S4P4. Dans le système YUV, on représente la coloration par un couple de deux octets U et V appelé chrominance. Il y a des raisons physiologiques et physiques derrière. Ce système est à la base de nombreuses méthodes de transmission d'images (PAL, Secam, JPEG, MPEG...). Ces systèmes seront présentés en détail dans le cours de [S4P4 - Architectures pour l'IN](#) ([S4P4%20-%20Outils%20IN.html#paragraphe_4](#)).

On pourra utiliser les [formules suivantes](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceddraw/html/_dxce_converting_between_yuv_and_rgb.asp) (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceddraw/html/_dxce_converting_between_yuv_and_rgb.asp) pour convertir une couleur RGB en couleur YUV :

$$\begin{cases} Y = (66R + 129G + 25B + 128) \gg 8 + 16 \\ U = (-38R - 74G + 112B + 128) \gg 8 + 128 \\ V = (112R - 94G - 18B + 128) \gg 8 + 128 \end{cases}$$

Et inversement :

$$\begin{cases} R = \text{clip}((298C + 409E + 128) \gg 8) \\ G = \text{clip}((298C - 100D - 208E + 128) \gg 8) \\ B = \text{clip}((298C + 516D + 128) \gg 8) \end{cases} \text{ avec } \begin{cases} C = Y - 16 \\ D = U - 128 \text{ et } \text{clip}(x) = (x > 255) ? 255 : (x < 0) ? 0 : x \\ E = V - 128 \end{cases}$$

Bien noter qu'il faut limiter ces nombres entre 0 et 255 (rôle de la fonction clip).

Ces formules ne font pas l'unanimité car il existe de nombreuses normes, on trouve aussi celle-ci qui correspondent au système PAL :

$$\begin{cases} Y = 0,299R + 0,587G + 0,114B \\ U = 0,492(B - Y) \\ V = 0,877(R - Y) \end{cases} \text{ et inversement } \begin{cases} R = Y + 1,140V \\ G = Y - 0,395U - 0,581V \\ B = Y + 2,032U \end{cases}$$

Pour illustrer cette théorie, voici un extrait de programme C qui calcule la luminance d'une image couleur. Il contient une astuce intéressante. On remarque que la luminance consiste à calculer $c1*r + c2*g + c3*b$ pour un pixel (r,g,b), avec c1, c2 et c3 trois constantes réelles. Au lieu de faire ces trois multiplications par des constantes à chaque pixel, et parce qu'il n'y a que 256 valeurs possibles pour chaque composante, on se sert de trois tableaux préinitialisés ; chacun contient les résultats des calculs (coefficient * composante). Par exemple LumR[128] contient 0.299 * 128 et LumG[128] = 0.587 * 128. Ensuite, il suffit d'additionner les nombres issus des tableaux pour obtenir la luminance.

```
int LumR[256], LumG[256], LumB[256];           // tableau de conversion RGB -> luminance

static void InitTablesLumRGB(void)
// initialisation des tables de coefficients
{
    int i;

    for (i=0; i<256; i++) {
        LumR[i] = (int)(i * 0.299);
        LumG[i] = (int)(i * 0.587);
        LumB[i] = (int)(i * 0.114);
    }
}

// la macro suivante calcule la luminance d'un triplet x, le calcul pour un pixel se
// limite donc à deux additions entières !
struct Pixel {
    int r,g,b;
};

#define Luminance(pixel) (LumR[ pixel.r ]+LumG[ pixel.g ]+LumB[ pixel.b ])
```

Couleurs indexées et palettes

C'est une technique pour économiser la place mémoire nécessaire pour représenter une image : indexer les couleurs, c'est à dire recenser toutes les couleurs présentes dans l'image et leur attribuer un numéro. De cette façon, l'image est représentée sous forme d'un tableau 2D de numéros accompagné d'une table (numéro de couleur, couleur).

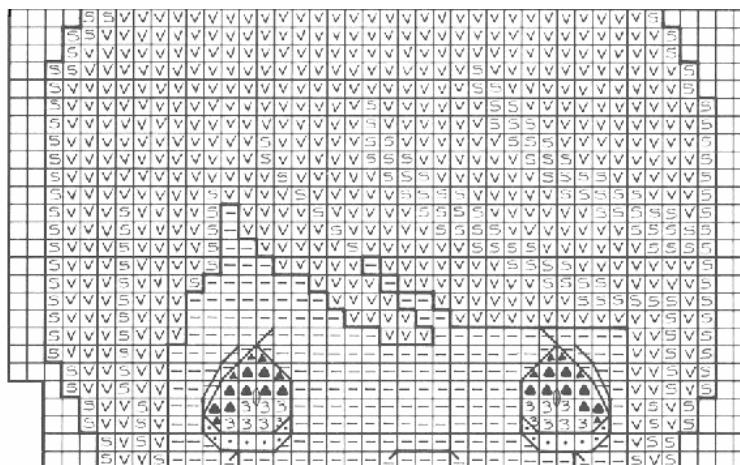
Evidemment, on ne gagne de la place que si le nombre de couleurs est faible ou alors qu'on force une réduction du nombre de couleurs en choisissant les couleurs les plus représentées. On ne gagne pas de place s'il y a un trop grand nombre de nuances différentes.

Dans une broderie simple, on utilise un petit nombre de couleurs seulement. Voici l'agrandissement d'un fragment :



Et voici la spécification des couleurs montrant la palette à gauche et les codes représentant les teintes à employer. Dans une image gif, ce ne sont pas des codes tels que les points, barres, triangles, mais des indices de 0 à 255.

DMC	COLOR
•	White White
◊ 225	Shell Pink-Vy. Lt.
• 224	Shell Pink-Lt.
◊ 677	Old Gold-Vy. Lt.
- 948	Peach Flesh-Vy. Lt.
◊ 754	Peach Flesh-Lt.
/ Ecru	Ecru
△ 822	Beige Grey-Lt.
✗ 739	Tan-Ultra Vy. Lt.
□ 738	Tan-Vy. Lt.
▼ 402	Mahogany-Vy. Lt.
§ 922	Copper-Lt.
✗ 369	Pistachio Green-Vy. Lt.
△ 368	Pistachio Green-Lt.
◊ 320	Pistachio Green-Med.
▲ 367	Pistachio Green-Dk.
□ 828	Blue-Ultra Vy. Lt.
■ 827	Blue-Vy. Lt.
■ 420	Hazel Nut Brown-Dk.



Les images GIF sont codées avec cette technique et utilisent en général 256 couleurs. Par exemple, voici une image gif 16x16, utilisant 16 couleurs :



la même image en plus gros
dump de l'image gif décompressée

```
Palette[ 0]=000 001 000
Palette[ 1]=130 000 023
Palette[ 2]=185 000 000
Palette[ 3]=220 000 000
Palette[ 4]=255 000 000
Palette[ 5]=255 000 083
Palette[ 6]=165 042 000
Palette[ 7]=219 074 001
Palette[ 8]=255 083 000
Palette[ 9]=255 106 104
Palette[10]=255 115 071
Palette[11]=255 148 115
Palette[12]=124 238 018
Palette[13]=205 207 203
Palette[14]=253 200 180
Palette[15]=255 210 212
```

13	13	13	13	13	13	13	13	0	0	13	13	13	13	13	13	13	13	13	13	13	13				
13	13	13	13	13	13	13	13	12	13	13	13	13	13	13	13	13	13	13	13	13	13	13			
13	13	13	13	13	13	13	13	0	0	13	13	13	13	13	13	13	13	13	13	13	13	13			
13	13	13	13	13	0	4	5	3	0	0	4	0	0	0	13	13	13	13	13	13	13	13	13		
13	13	0	11	11	3	2	6	12	6	3	11	10	9	13	13	13	13	13	13	13	13	13	13		
13	0	9	9	9	11	11	14	14	11	9	9	9	4	10	9	4	9	13	13	13	13	13	13		
13	10	8	9	10	9	9	9	9	9	9	9	4	4	4	8	9	8	0	0	0	0	0	0	0	
13	10	15	15	10	10	10	10	9	4	4	4	4	4	4	8	9	8	0	0	0	0	0	0	0	
13	10	15	15	10	10	8	8	10	4	4	4	4	4	4	8	8	8	0	0	0	0	0	0	0	
13	10	8	8	4	10	8	8	10	4	4	4	4	4	10	4	4	4	0	0	0	0	0	0	0	
13	0	4	4	4	10	8	4	10	4	4	4	4	4	4	4	4	4	13	13	13	13	13	13	13	
13	0	4	15	4	8	4	3	4	4	4	4	4	4	4	4	4	4	3	13	13	13	13	13	13	13
13	13	4	4	4	4	7	4	3	4	4	4	3	7	6	0	13	13	13	13	13	13	13	13	13	13
13	13	0	3	4	3	7	8	7	4	4	3	6	2	6	13	13	13	13	13	13	13	13	13	13	13
13	13	13	13	0	2	2	3	3	2	6	3	2	6	13	13	13	13	13	13	13	13	13	13	13	13
13	13	13	13	13	0	6	1	1	0	1	2	1	13	13	13	13	13	13	13	13	13	13	13	13	13

On voit que la couleur 13 est une sorte de gris moyen et la couleur 4 est un rouge vif. Le fichier GIF contient ces deux tableaux : la liste des couleurs et le tableau des numéros de couleur.

On a donc un tableau des quelques couleurs présentes dans l'image et aussi le tableau 2D des numéros de couleurs. En principe, l'ensemble des deux pèse moins que l'image non indexée, puisqu'au lieu d'avoir L^*H triplets RVB, on a seulement L^*H octets (s'il n'y a pas plus de 256 couleurs différentes dans l'image, sinon c'est davantage). En plus, le tableau 2D des indices est compressé dans le fichier. Il y a bien moins qu'un octet par pixel, puisqu'on observe de nombreuses répétitions dans les données. L'image ferait $16^3 + 16^2$ octets au moins sans compression, soit 304 octets sans compter l'entête. Or le fichier [pomme.gif](#) (figures/s3p3/pomme.gif) ne fait que 189 octets tout compris. Voir pourquoi dans 2 semaines.

Résolution et définition

Ces deux notions décrivent le nombre de pixels qu'on peut afficher avec une image. La définition d'une image raster est sa taille largeur * hauteur. La résolution caractérise le périphérique d'affichage ou de saisie de l'image. Elle définit le nombre de pixels par unité de longueur qui peuvent être produits distinctement.

Les deux sont liées quand on veut obtenir une image d'une certaine dimension spatiale : taille = définition / résolution. Par exemple, avec une résolution de 200ppi et une définition de 1280x1024, on obtient une image de 6.4x5.12 pouces, c'est à dire 16x13 cm.

Formats de fichier

Un format de fichier est une norme de codage. Il existe un très grand nombre de formats de fichiers pour les images. Par rapport au nombre de formats existants, peu de formats sont universels (lus sur toutes les plateformes et spécification domaine public). La plupart des formats sont "propriétaires" : soit secrets, soit soumis à des droits d'auteurs.

Structure d'un fichier image

Les formats de fichiers pour les images raster sont quasiment tous structurés de cette manière :

- entête de fichier qui donne les informations :
 - "nombre magique" : c'est un identifiant unique et constant du type de l'image, 1, 2 ou plusieurs octets toujours les mêmes pour le type du fichier : P6, gif89a...
 - taille de l'image : largeur, hauteur en nombre de pixels

- profondeur (ou implicite) dont on peut déduire la taille de la palette le cas échéant, ex : profondeur = 8 bits donc palette de 256 couleurs
- espace de couleur (ou implicite selon le format),
- technique de compression des données (ou implicite)
- éventuellement une palette : $2^n = N$ couleurs afin que le pixmap ne contienne que les numéros de ces couleurs
- le ou les pixmaps du fichier : certains formats (IFF/ILBM, gifs "animés",...) permettent de stocker plusieurs images à la suite dans le même fichier, il y a parfois un entête pour chacune de ces images.

Les données peuvent être structurées ensuite en octets, mots ou plus finement en bits : le nombre de bits attribués aux champs n'est pas multiple de 8 et il faut alors des fonctions de lecture spéciales.

Formats PAM, PPM, PNM, PGM et PBM : librairie netpbm

Ce format est peu utilisé pour stocker des images mais il est bien pratique pour programmer sur linux : il est très simple à lire et écrire et il est livré avec une librairie de fonctions très faciles à utiliser. Cette librairie s'appelle netpbm. Elle a été mise à jour récemment, mais on peut encore utiliser les anciennes fonctions décrites ci-dessous.

Description des formats PNM

PNM signifie "Portable aNy Map", c'est à dire format portable pour les images raster. Il recouvre les formats PBM, PGM et PPM : bitmap, graymap et pixmap. La structure d'un fichier PNM est la suivante. L'entête est écrit sous la forme d'un texte lisible codé en ascii. Ensuite on trouve le pixmap écrit en binaire.

Chaque élément de l'entête est terminé par un retour à la ligne :

1. deux caractères indiquent le type du fichier : "P6" pour PPM, "P5" pour PGM, "P4" pour PBM, "P3" pour une image en couleur mais sous forme de texte, voir l'exemple plus bas.
2. les largeur et hauteur de l'image écrites en ascii : "%d %d"
3. la valeur maximale des composantes (ex : "255" pour RVB/24 bits)

Les pixels sont représentés de haut en bas par rangées, et dans une rangée, de gauche à droite. On trouve donc hauteur * largeur pixels fournis en binaire : 3 octets RVB par pixel si maxval <= 255, 6 octets MSB-LSB sinon.

Pour les PBM, il n'y a pas la profondeur de l'image et les pixels sont codés à raison de 8 par octet. bit 1 = noir, bit 0 = blanc.

Voici un "dump" binaire d'un fichier PPM. Les valeurs FF qu'on voit sont des pixels blancs.

dump d'une image PPM

00000000	50 36 0a 31 36 20 31 36 0a 32 35 35 0a ff ff ff ff	P6.16 16.255....
00000010	fe fe fe fe fe fe fe fe ff ff ff f5 f2 f2 c2
00000020	d5 b8 ff ff ff ff ff ff ff ff fe fe
00000030	fe fe fe fe fe fe fe fe fe ff ff ff ff ff ff
00000040	ff ff ff fe fe fe fe fe fe ff ff ff e8 e0 e2 7d}.
00000050	a6 65 e7 ec e4 f8 f6 f4 f7 f4 ef ff ff ff ff	.e.....
00000060	ff fe
00000070	ff ff ff ff fb fe ed eb e8 b8 a6 92 88 b9 6e 71nq

On constate qu'un tel fichier peut être facilement créé par un petit programme, quelque soit le langage. Voici par exemple un script bash qui envoie une telle image sur stdout. L'image est au format PPM plain, c'est à dire que les composantes des pixels ne sont pas écrites en binaire, mais sous forme de chiffres

lisibles qui sont à transformer en nombres. Par exemple, on trouve "255 255 107" (11 caractères ascii) pour représenter la couleur fleur de soufre. Mettre 12 octets (11 pour les composantes + 1 espace) pour chaque pixel est évidemment un codage très peu économique.

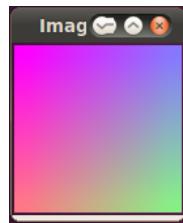
Exemple de génération d'une image PPM par un script bash

```
#!/bin/bash

# entête (image au format PPM "plain", non binaire)
echo 'P3 8 8 15'

# parcourir les 8 lignes et les 8 colonnes
for l in 0 1 2 3 4 5 6 7
do
    for c in 0 1 2 3 4 5 6 7
    do
        # produire R V B du pixel (l,c)
        R=$((15 - c))
        V=$((l + c))
        B=$((15 - l))
        echo -n "$R $V $B "
    done
    echo ''
done
```

Saisir ce script dans un fichier `gendegrades.sh`, le rendre exécutables puis faire `gendegrades.sh | display`. On obtient ceci (en agrandissant la fenêtre).



Résultat d'exécution du script

Commandes de traitement d'images PNM

On parle ici des commandes Unix à taper dans un tube (pipe) sur la ligne de commande.

Il existe un grand nombre de commandes de traitement d'images PNM. Toutes ces commandes sont des filtres Unix. On les enchaîne pour effectuer des traitements complexes. Ne pas oublier de placer une redirection de sortie à la fin du tube.

On conseille de connaître les commandes suivantes (consulter les docs en ligne) :

display

affiche une image provenant soit de stdin s'il n'y a pas de paramètre, soit du paramètre.

anytopnm

décode une image d'un format quelconque et le sort en PPM sur stdout

pnmto*

encodent une image PNM en un autre format

pnmrotate, pnmshear, pnmscale

effectuent des découpages ou redimensionnements sur une image

pnmflip, pnmrotate, pnmquant

effectuent des déformations sur une image

pnmquant, ppmdither

réduisent le nombre de couleurs de l'image

Exemple de programme PNM

On parle ici des fonctions de la librairie netpbm à utiliser dans un programme C.

Voici un exemple de programme, il charge une image au format PNM, et la transforme en Luminance et l'enregistre au format PGM. Tous les programmes PNM suivent à peu près cette trame :

1. Initialisation de la librairie PNM
2. parcours des paramètres du programme et affectation des options, ouverture des fichiers...
3. lecture du ou des images à traiter
4. traitement des images : production d'une image résultat
5. enregistrement de l'image résultat

On peut remarquer que l'absence de paramètres fichiers tant en entrée qu'en sortie implique l'utilisation de stdin et stdout : la commande se trouve dans un tube ou des redirections.

luminance.c

```
#include <math.h>
#include <pnm.h>
#include <stdio.h>
#include <stdlib.h>

/** variables globales du programme **/ 

int verbose = 0;           // les fonctions doivent être bavardes

/** Conversion en luminance **/ 

int LumR[256], LumG[256], LumB[256];

static void InitTablesRGB(void)
{
    int i;
    for (i=0; i<256; i++) {
        LumR[i] = (int)(i * 0.299);
        LumG[i] = (int)(i * 0.587);
        LumB[i] = (int)(i * 0.114);
    }
}

#define Luminance(x) (LumR[(x).r]+LumG[(x).g]+LumB[(x).b])

/** Conversion de l'image RVB en Y **/ 

static gray **pgm_Luminance(xel **imagergb, int X, int Y)
{
    gray **lumi;
    int x,y, dr,dg,db;

    // créer un tableau de X * Y pixels
    lumi = pgm_allocarray(X,Y);

    // parcourir tous les pixels
    if (verbose>0) pm_message("image %dx%d en luminance",X,Y);
    for (y=0; y<Y; y++) {
        if (verbose>1) pm_message("traitement de la ligne %d",y);
        for (x=0; x<X; x++) {
            // calculer la luminance
            lumi[y][x] = Luminance(imagergb[y][x]);
        }
    }
    return lumi;
}
```

```

}

/*** Programme principal ***/

int main(int argc, char *argv[])
{
    // variables PNM représentant les deux images
    xel **ImageRGB;      // image RGB à traiter
    gray **ImageL;       // image PGM résultante
    int Largeur,Hauteur; // dimensions
    unsigned int maxval;
    int format;

    // paramètres et options du logiciel
    char *usage = "[-algo 0|1] [-verbose] [-output pgmfile] [pnmfile]";
    int algo;             // type d'algo de traitement (démo des options)
    char *nomFichierS = NULL; // nom du fichier à créer ou NULL
    int argn;             // numéro du paramètre traité

    // fichiers à traiter
    FILE *fichierE, *fichierS; // fichiers d'entrée et de sortie

    /** initialisations : librairie PAM et algo de conversion **/
    pnm_init(&argc,argv);
    InitTablesRGB();

    /** lecture des options fournies au programme **/
    argn = 1; // on commence avec $1
    // on boucle sur tous les arguments qui commencent par un '-' suivi de quelque chose
    while (argn < argc && argv[argn][0]=='-' && argv[argn][1]!='\0') {

        // est-ce que l'argument courant est -algo (ou -a) ?
        if (pm_keymatch(argv[argn], "-algo", 1)) {
            // est-ce qu'il y a un argument de plus et est-ce que c'est un nombre
            if (argn+1 >= argc || sscanf(argv[argn+1],"%d",&algo)!=1)
                // non => erreur
                pm_usage(usage);
            argn+=1;
        } else

            // est-ce que l'argument courant est -output (ou -o, -out...) ?
            if (pm_keymatch(argv[argn], "-output", 1)) {
                // est-ce qu'il y a un argument de plus, c'est le nom du fichier
                if (argn+1 >= argc)
                    // oui
                    nomFichierS = argv[argn];
                else // non, on est à la fin des arguments => erreur
                    pm_usage(usage);
                argn+=1;
            } else

                // est-ce que l'argument courant est -verbose (ou -v...)
                if (pm_keymatch(argv[argn], "-verbose", 1)) verbose++;

        // on est sur un argument -xxx qu'on ne connaît pas
        else pm_usage(usage); // erreur fatale ->exit()

        // passer à l'argument suivant
        argn++;
    }
    /* nom du fichier à traiter ou stdin ? */
    if (argn < argc) {
        fichierE = pm_openr(argv[argn]);
        argn++;
    }
}

```

```

} else fichierE = stdin;

/* lecture de l'image à traiter */
ImageRGB = pnm_readpnm(fichierE, &Largeur, &Hauteur, &maxval, &format);
fclose(fichierE);

/* traitement : mise en luminance */
// pas fait, mais : if (algo == 1) ... else
ImageL = pgm_Luminance(ImageRGB, Largeur, Hauteur);
pnm_freearray(ImageRGB, Hauteur);

/* ouvrir le fichier de sortie */
if (nomFichiers) fichierS = pm_openw(nomFichiers);
else fichierS = stdout;

/* écrire l'image résultante dans le fichier de sortie */
pgm_writepgm(fichierS, ImageL, Largeur, Hauteur, maxval, 0);
fclose(fichierS);
pgm_freearray(ImageL, Hauteur);
return 0;
}

```

Voici le Makefile pour le compiler, rien de compliqué, il faut seulement le lier avec la librairie netpbm (qui changera peut-être de nom un jour pour s'appeler pam). On peut aussi rajouter la librairie mathématique -lm si nécessaire.

```

luminance:    luminance.c
    cc -o $@ $< -lm -lnetpbm

```

Et pour le lancer :

```

anytopnm image | luminance | display

```

Description de quelques fonctions utiles

Les fonctions de netpbm permettent d'écrire des programmes courts et lisibles. Voici quelques fonctions utiles à connaître.

Fonctions diverses

- #include <pnm.h> et autres pgm.h, pbm.h... : inclusions des types et prototypes de la librairie.
- void pnm_init(&argc, argv) : à appeler une fois pour initialiser la librairie
- bool pm_keymatch(chaine, "-option", nb) : compare la chaine aux nb premiers caractères de l'option. renvoie 1 s'il y a une correspondance. Ca permet de réduire les noms des options. ex : "-verbose", 1 permet de taper seulement -v.

Fonctions d'affichage de texte

- void pm_usage(message) : affiche le message et quitte le programme. On l'utilise pour signaler une erreur fatale sur le mode d'emploi.
- void pm_message(format [, valeurs...]) : affiche le message avec les valeurs pour des jokers %d, %s... comme le ferait printf, mais sans écrire sur stdout qui est généralement réservé pour l'image de sortie.

Lecture de fichier

- FILE *pm_openr(nomfichier) : ouvre le fichier indiqué en lecture, équivalent à fopen(nom,"r");
- xel **pnm_readpnm(FILE *, int &Largeur, int &Hauteur, unsigned int &maxval, int &format) : lit le fichier PNM ouvert en lecture, crée et remplit

un tableau 2D Hauteur*Largeur de xel = struct { unsigned int r,v,b; }; La valeur maxval indique le nombre de bits de chaque composante (maxval=255 pour RGB/24bits). Le format donne le type de l'image lue.

- `xel **ppm_readppm(FILE *, int &Largeur, int &Hauteur, unsigned int &maxval);` : lit le fichier PPM ouvert en lecture, crée et remplit un tableau 2D de xel.
- `gray **pgm_readpgm(FILE *, int &Largeur, int &Hauteur, unsigned int &maxval);` : lit le fichier PGM ouvert en lecture, crée et remplit un tableau 2D de gray = unsigned int; La valeur maxval indique le nombre de bits des pixels, maxval=255 pour 8 bits.
- `bit **pbm_readpbm(FILE *, int &Largeur, int &Hauteur, int &format);` : lit le fichier PBM ouvert en lecture, crée et remplit un tableau 2D de gray = unsigned int; La valeur maxval indique le nombre de bits des pixels, maxval=255 pour 8 bits.

Ecriture de fichier

- `FILE *pm_openw(nomfichier);` : ouvre le fichier en enregistrement
- `void ppm_writeppm(FILE *sortie, xel **image, largeur, hauteur, maxval, 0);` écrit l'image PPM dans le fichier ouvert en enregistrement.
- `void pgm_writepgm(FILE *sortie, gray **image, largeur, hauteur, maxval, 0);` écrit l'image PGM dans le fichier ouvert en enregistrement.
- `void pbm_writepbm(FILE *sortie, bit **image, largeur, hauteur, 0);` écrit l'image PBM dans le fichier ouvert en enregistrement.

Création de tableaux en mémoire

- `xel **image = ppm_allocarray(largeur, hauteur);` : crée un pixmap de cette dimension, attention on accède à ses pixels par `image[y][x]` avec $0 \leq y \leq \text{hauteur}$ et $0 \leq x \leq \text{largeur}$.
 - `void ppm_freearray(xel **image, hauteur);` : libère la mémoire occupée par le pixmap.
 - `gray **image = pgm_allocarray(largeur, hauteur);`
 - `void pgm_freearray(gray **image, hauteur);`
 - `bit **image = pbm_allocarray(largeur, hauteur);`
 - `void pbm_freearray(bit **image, hauteur);`
-

Format BMP

Le format BMP est normalisé par Microsoft. On décrira ici le format non compressé et uniquement pour information car on fera appel à des fonctions spécialisées de lecture et d'enregistrement.

Il y a quatre parties dans un fichier BMP :

entête du fichier

Il indique les caractéristiques du fichier : type, taille et emplacement de l'image dans le fichier.

entête d'image

Il définit les caractéristiques de l'image contenue dans le fichier.

palette

zone optionnelle, on donne la liste des 256 couleurs utilisées.

données de l'image

c'est la partie pixmap, compressé ou non.

Contenu d'un fichier BMP

Voici la description de ces zones pour information uniquement. Le type B = octet, W = 2 octets, DW = 4 octets, S[2] = deux octets= deux caractères.

Donnée	Type	Valeur	Explications
entête du fichier			
bfType	S[2]	"BM"	identification du format
bfSize	DW		taille exacte du fichier, on peut la déduire par la formule suivante : $14 + biSize + biClrUsed*4 + biSizeImage$
bfReserved	DW	0	pas utilisé, mettre 0
bfOffBits	DW	54 si pas de palette	position relative de l'image : $14 + biSize + biClrUsed*4;$
entête d'image			
biSize	DW	40	nombre d'octets de cet entête
biWidth	DW		largeur de l'image, attention nombre pair !
biHeight	DW		hauteur de l'image
biPlanes	W	1	nombre de plans (toujours 1)
biBitCount	W	1, 4, 8, 16, 24 ou 32	taille des pixels
biCompression	DW	0	méthode de compression, 0 = aucune
biSizeImage	DW		nombre d'octets du bitmap : $(biWidth + 7) / 8 * biHeight$
biXpelsPerMeter	DW	3780	nombre de pixels par mètre
biYpelsPerMeter	DW	3780	nombre de pixels par mètre
biClrUsed	DW	0	nombre de couleurs utilisées, mettre 0 pour toutes
biClrImportant	DW		nombre de couleurs importantes, mettre 0 pour toutes
palette (optionnelle)			seulement quand biBitCount < 24
rvb...	RGBQUAD		4 octets par couleurs (r,g,b,0), $2^{biBitCount}$ couleurs à indiquer.
image			tableau de $biWidth * biHeight$ quads ou indices dans la palette

Exemples

Chercher Storing an Image dans GDI: Platform SDK. Il y a un exemple de programme qui crée de toutes pièces un fichier .BMP.

Format TIFF non compressé

Le format TIFF a été normalisé par Adobe. Voir [ce document](http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf) (<http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf>) qui présente la spécification complète. Ce format est très polyvalent : images bitmap, niveaux de gris, palette, couleurs complètes... Il permet de compresser ou non les données. Ici, on va seulement esquisser le format dans des cas particuliers.

<PAS FINI, mais est-ce utile ? avec le format PNM et BMP on a une idée suffisante de ce qu'il y a dans un fichier non compressé : entête et pixmap>

Concepts à retenir

- Une image n'est pas nécessairement un tableau rectangulaire de pixels, il y a aussi des images vectorielles, définies par des primitives géométriques.
- Le codage de l'information consiste à représenter cette information sous forme de nombres. Par exemple représenter une couleur par 3 octets.
- La compression de l'information consiste à réduire le nombre d'octets nécessaires pour représenter l'information. Voir le cours suivant.
- Les formats BMP et PNM ne sont pas compressés : les fichiers contiennent les données des pixels successifs.

Semaine 12 (28/11/2011) : Compression de données (partie 1)

Concepts de la théorie de l'information

Compresser des données consiste essentiellement à réduire la place qu'elles occupent sans changer la *signification* des données elles-mêmes. Le mot "signification" est vague à dessein, on verra que cela autorise de perdre un peu d'information sur certaines données.

Que sont ces données ? Dans ce cours, on va principalement s'occuper de nombres codés en base deux avec un nombre de bits non nécessairement égal à 8. C'est à dire que les données sont une suite d'octets ou de mots de n bits, par exemple une image de largeur x hauteur x 3 composantes sur 8 bits ($n=24$), ou une image en niveaux de gris codés de 0 à 63 ($n=6$) ou un texte en code ascii, 1 octet par caractère ($n=8$).

Sans compression, ces données occuperaient un certain espace, largeur*hauteur*24 bits par exemple. On souhaite réduire au mieux le nombre de bits nécessaires pour stocker ces données dans un fichier. Evidemment, la relecture des données ne sera plus directe, il faudra passer par un algorithme de décompression pour retrouver les données d'origine ou des données ayant la même signification.

Sur ce concept de *signification similaire*, par exemple, si on doit compresser un texte écrit en utf-8 avec des caractères sur 8 et 16 bits et qu'on se rend compte qu'il n'y a que des caractères internationaux, aucun accent, on peut accepter qu'il soit décompressé en simple ascii 8 bits sans perdre en signification. Ce ne sont pas les mêmes données qu'on récupère, mais elles signifient la même chose. On peut même concevoir de compresser un texte en lui retirant tous ses accents, après tout, même sans accents, ça reste comprehensible, non ? La *signification* dépend donc beaucoup de l'usage qu'on compte en faire.

Beaucoup d'algorithmes de compression procèdent ainsi : ils éliminent délibérément certaines informations du signal à coder : les choses que l'humain ne perçoit pas bien, par exemple, des nuances de couleurs invisibles pour la plupart des yeux (ou impossibles à reproduire fidèlement sur la plupart des écrans, voir le cours de S4P4), des sons inaudibles pour la plupart des oreilles parce que masqués par d'autres sons plus forts (mp3, téléphonie).

Dans ce cours, on va commencer par s'intéresser à la compression de textes ou de données binaires (des octets), puis on s'occupera des images, puis des vidéos.

Les données d'entrée du compresseur sont représentées par des codes. Ex: 'A'=65 / 8bits. **La compression consiste à représenter les éléments des données par des codes plus petits que les codes d'origine.** Par exemple, si on n'a que des lettres de 'a' à 'z' et l'espace, on peut se contenter de 5 bits par caractère. Les algorithmes de compression qu'on va voir font des raisonnements bien plus complexes pour réduire le codage encore davantage.

Définition d'un signal

Quelques mots de vocabulaire :

source ou signal

Il s'agit d'une suite d'événements, ce sont les données qu'on essaie de compresser.

longueur du signal L

La longueur du signal est le nombre d'événements qu'il contient.

événement

C'est l'un des éléments du signal, par exemple l'une des lettres du texte à compresser, cet événement appartient à l'alphabet utilisé.

symbole

C'est l'unité d'information d'un signal, par exemple un nombre 8 bits ou un triplet RVB sur 24 bits. Les symboles doivent appartenir à l'alphabet.

alphabet

C'est l'ensemble des symboles possibles pour un événement, par exemple tous les codes ascii pour coder un texte international. On pourra raisonner sur la longueur de l'alphabet : le nombre de symboles connus.

Recodage de source

L'une des techniques qu'on emploie en compression est le **recodage de source**. Ça consiste à changer l'alphabet du signal : utiliser un autre alphabet pour coder les symboles du signal.

Exemple : on a une séquence de lettres majuscules, l'alphabet $A_1 = \{ A, B, C, \dots, Z \}$ donc $N_1 = 26$ qu'on veut coder en base 2, c'est à dire sur l'alphabet $A_2 = \{ 0, 1 \}$ avec $N_2 = 2$.

La théorie indique qu'il faut $\lceil \log_{N_2}(N_1) \rceil$ symboles de A_2 pour coder un symbole de A_1 . $\lceil x \rceil$ donne le plus petit nombre entier supérieur ou égal à x : $\lceil 4 \rceil = 4$, $\lceil 4.1 \rceil = 5$, $\lceil -4.1 \rceil = -4$ et on rappelle

$$\text{que } \lceil \log_N(x) \rceil = \frac{\log(x)}{\log(N)}$$

Sur l'exemple précédent, il faudra $\lceil \log_2(26) \rceil = \lceil 4.7004397 \rceil = 5$, c'est à dire qu'il faut 5 symboles de A_2 (bit 0 ou 1) pour coder un symbole de A_1 . Attention, il ne faut pas en conclure qu'on ne compresse pas sous prétexte qu'on passe, pour chaque symbole, de 1 élément ('A', 'B',...) à 5 éléments (00000, 00001, ...). Il faut seulement 1 bit pour coder un symbole de A_2 , tandis qu'il en faut 5 fois plus pour ceux de A_1 . Ce n'est pas seulement dans le nombre de symboles que réside la compression mais à la fois dans leur nombre et dans leur taille.

On verra plus bas qu'on emploie également des codages de taille variable : rien n'oblige à recoder chaque symbole de A_1 avec le même nombre de symboles dans A_2 . C'est une idée extrêmement importante en compression des données.

On peut tout de suite remarquer que si le signal est réduit à par exemple ABACABADA, on n'a pas besoin de 5 symboles de A_2 pour recoder chaque symbole, il en suffit de 2 (AB=00, AC=01, AD=10, A=11 par exemple) : 0001001011. C'est possible sur ce signal car il y a une certaine forme de redondance : on trouve des corrélations entre les événements. Dans ce cas, on dit que ce signal est une *source markovienne*. On va en parler maintenant.

Source markovienne

Les sources sont classées selon l'indépendance de leurs événements :

source constante

C'est un signal qui ne contient qu'un seul type de symbole...

source aléatoire

C'est un signal dont les événements sont complètement indépendants, sans aucune relation entre eux, que ce soit entre les voisins ou entre des événements distants. On l'appelle aussi SDSM : source discrète sans mémoire. Il y en a assez peu dans le monde réel car en général, il y a toujours des corrélations entre les événements.

source markovienne

C'est le type de signal le plus intéressant pour la compression : les événements sont susceptibles d'être liés, entre voisins ou à plus longue distance : on parle de source markovienne d'ordre n quand les corrélations entre événements s'étendent sur n événements consécutifs. Par exemple, si un B est toujours suivi d'un A, il y a une corrélation markovienne.

On comprime plus facilement les sources markoviennes si on arrive à détecter les **corrélations entre événements**. Il suffit de recoder le signal avec un autre alphabet qui regroupe les séquences répétées. Mieux on détecte les corrélations, mieux on compresse le signal. Par exemple, dans le signal ABACABADA, le A est pratiquement toujours suivi d'une autre lettre ou une autre lettre est toujours suivie d'un A, donc on pourrait économiser la représentation du A (on ne coderait le A que lorsqu'il n'est pas suivi d'autre chose).

Dans le cours, on va d'abord voir comment compresser des sources aléatoires puis on va étudier comment gagner de la place avec des sources markoviennes, d'abord avec des méthodes générales puis avec des méthodes de plus en plus spécifiques des images et vidéos.

Probabilités d'apparition et information

Les sources aléatoires et markoviennes sont liées aux probabilités. Pour compresser, on s'intéresse au nombre de fois qu'un symbole est apparu dans un signal. Certains symboles sont plus représentés que d'autres. On définit deux grandeurs relatives au signal traité :

occurrence d'un symbole $o(a)$

C'est le nombre de fois que ce symbole est apparu dans le signal, par exemple dans ABACABADA, le A est apparu 5 fois et le B 2 fois : $o('A') = 5$, $o('B') = 2$.

probabilité $p(a)$

$p(a) = o(a) / L$: la probabilité d'apparition d'un symbole est le nombre de ses apparitions divisé par la longueur du signal. La somme des probabilités des symboles d'un alphabet = 1. Certains symboles peuvent être très fréquemment rencontrés et d'autres presque pas, voire pas du tout. Par exemple, dans la phrase précédente, il n'y a aucun 'z', il y a 1 'y' et par contre, il y a 12 'e'.

quantité d'information d'un événement

La quantité d'information est, pour simplifier, l'importance d'un événement en ce qui concerne son codage pour une compression optimale. En simplifiant à l'extrême, la quantité d'information est proportionnelle à la taille du codage. La quantité d'information d'un événement est inversement proportionnelle à son occurrence : un événement très probable représente peu d'information et au contraire un événement peu probable représente beaucoup d'information, au sens de cette discipline. C'est un peu paradoxal à première vue, mais il y a une raison. La compression des données consiste à remplacer des événements par des codes. On va faire en sorte de représenter les événements très fréquents (probables) avec les plus petits codes possibles : comme ce sont les plus nombreux, il faut qu'ils prennent le moins de place au total = ils représentent une petite quantité d'information. Au contraire, les événements les moins fréquents sont représentés par un code plus long car ils véhiculent plus d'information, toujours au sens de cette discipline. C'est comme au Scrabble, la valeur des lettres est proportionnelle à leur rareté.

information globale

C'est la somme des informations de chaque événement pris isolément. On fait la somme de ce que représente chaque événement du signal.

Le principe de la compression de données est de mesurer la quantité d'information véhiculée par chaque événement et la comparer à l'information globale du signal, afin de déterminer comment recoder le signal pour que, avec le nouveau codage, la somme des informations véhiculée par chaque code soit minimale.

On se place dans le cas d'une source aléatoire de longueur L finie. Elle est définie sur un alphabet α de n_α symboles notés α_i avec $0 \leq i < n_\alpha$. On veut la recoder sur un alphabet β de n_β symboles notés β_j ($0 \leq j < n_\beta$).

Tout d'abord, ainsi que vu plus haut, le nombre moyen de symboles de β qu'il faut pour coder un des symboles de α est défini par $\log_{n_\beta}(n_\alpha)$. Mais c'est un nombre moyen, valable pour un signal aléatoire, tandis qu'en réalité certains symboles α sont plus fréquents que d'autres, donc ils représentent moins

d'information que les symboles rares. On va faire en sorte qu'ils utilisent moins de symboles β pour les coder.

Pour cela, on définit la quantité d'information $I(\alpha_i)$ associée à un symbole α_i par :

$$I(\alpha_i) = -\log_{n_\beta}(p(\alpha_i)) = \log_{n_\beta}\left(\frac{1}{p(\alpha_i)}\right) \text{ pour } 0 \leq i < n_\alpha \text{ avec } p(\alpha_i) = \frac{o(\alpha_i)}{L}.$$

C'est une valeur qui est relative à l'alphabet β qu'on utilise pour le recodage. Elle représente le nombre de symboles de β qu'il faudra pour coder un symbole de α . On calcule cette valeur pour chaque symbole α_i du vocabulaire d'entrée dont les occurrences ne sont pas nulles.

Par exemple, pour le signal ABACABADA : $L = 9$ écrit avec un alphabet $\alpha = \{'A', 'B', \dots, 'Z'\}$ de $n_\alpha = 26$ symboles. On a les probabilités suivantes :

$$p('A') = 5/9 ; p('B') = 2/9 ; p('C') = 1/9 ; p('D') = 1/9$$

On veut le recoder sur un alphabet $\beta = \{'0', '1'\}$ à $n_\beta = 2$ codes. Cela donne les quantités d'information suivantes :

$$I('A') = 0.847 ; I('B') = 2.170 ; I('C') = I('D') = 3.170.$$

Ces valeurs sont les nombres moyens de symboles β qu'il faut pour coder les symboles A, B, C et D du signal d'entrée. C'est à dire qu'il faudra environ 1 bit pour coder les A, 2 pour coder les B et 3 pour coder les C et les D. Il y a 5 A, 2 B, 1 C et 1 D, donc il faut $5 + 2*2 + 3 + 3 = 15$ bits en tout.

Le signal d'origine est codé sur $n_\alpha = 26$ symboles. On a $\log_{n_\beta}(n_\alpha) = 4.700$: donc en principe, si c'était un signal aléatoire pouvant contenir d'autres lettres que A, B, C, D, il faudrait 4.7, c'est à dire 5 bits pour coder chaque lettre sans compression, donc $L*5 = 45$ bits pour l'ensemble.

Mais dans ce signal-là, certains caractères sont bien plus fréquents que d'autres et pourraient être représentés par seulement 1, 2 ou 3 bits. On peut donc certainement recoder ce signal et gagner de la place. La méthode ne dit pas comment faire mais seulement ce qu'on peut théoriquement gagner.

Entropie d'un signal

Claude Shannon a défini la notion d'**entropie** dans la théorie de l'information. Il s'agit de mesurer l'incertitude sur les événements successifs d'un signal (ou la corrélation entre événements consécutifs) : l'entropie est nulle dans un signal constant et maximale dans un signal aléatoire. L'entropie d'un événement e quelconque appartenant à un alphabet α à n_α symboles est définie par :

$$H(e) = \sum_{i=0}^{n_\alpha-1} p(\alpha_i)I(\alpha_i) = \sum_{i=0}^{n_\alpha-1} -p(\alpha_i)\log_{n_\beta}(p(\alpha_i))$$

NB: le calcul à faire est pour chaque symbole de l'alphabet : calculer $p = \text{nombre d'occurrences}/\text{longueur du message}$, calculer $-p*\log(p)/\log(2)$ et cumuler ces nombres.

Exemple, dans le signal considéré comme aléatoire ABACABADA, l'entropie d'un événement quelconque e est $H(e) = \frac{5}{9} \times 0.847 + \frac{2}{9} \times 2.170 + \frac{1}{9} \times 3.170 + \frac{1}{9} \times 3.170 = 1.657$. NB: on ne cherche pas à spécifier e , $H(e)$ est une notation pour désigner la quantité d'information moyenne des événements du signal.

C'est la moyenne pondérée par les occurrences des quantités d'informations de chaque symbole. La valeur 1.657 signifie que si le signal est purement aléatoire mais contenant une certaine proportion de A, de B, de C et de D, il faudrait en moyenne 1.657 bit pour coder chaque symbole compte tenu des occurrences observées, au lieu de 5 bits du signal d'origine, ceci parce que certains symboles sont très fréquents et d'autres très rares. C'est à dire que ABACABADA serait compressé comme AAAAABBCD, ce sont

tous les deux des signaux aléatoires avec la même entropie. On verra plus tard comment faire encore mieux avec les sources markoviennes.

On peut en déduire la longueur du signal compressé. Il faudra $L^*H(e)$ bits pour recoder le signal d'entrée avec des 0 et 1.

Voici un petit programme écrit en python2 pour calculer l'entropie d'un signal :

```
from math import log

def log2(nb): return log(nb) / log(2)

def Entropie(signal):
    # calcul des occurrences
    occurrences = {}
    for evenement in signal:
        if evenement in occurrences:
            occurrences[evenement] = occurrences[evenement] + 1
        else:
            occurrences[evenement] = 1
    # affichage des occurrences
    for symbole in occurrences:
        print symbole, '/', occurrences[symbole]
    # calcul de l'entropie
    entropie = 0.0
    for symbole in occurrences:
        p = float(occurrences[symbole]) / len(signal)
        entropie = entropie + p * log2(1.0/p)
    return entropie

# test du programme sur une chaîne de caractères
print Entropie( 'CITRONTRESCONTRIT' )

# test sur des triplets d'entiers (le même algo python tourne sur n'importe quelles
# données)
print Entropie( [ (5,0,0), (2,6,3), (5,0,0), (2,6,3), (0,3,0), (5,0,0) ] )
```

Le taper dans un fichier appelé *entropie.py* en faisant très attention aux tabulations qui spécifient l'imbrication des instructions, il n'y a pas d'accolades, seulement des tabulations. Le lancer en tapant `python entropie.py`.

Extension d'une source

Une des techniques de compression est le recodage des symboles du signal d'entrée en les groupant par nombre m fixe, ex: deux par deux, trois par trois... et à affecter un code à ces groupes. La nouvelle source est appelée extension d'ordre m . C'est intéressant quand le nombre de groupes est inférieur au nombre de symboles dans l'alphabet de la source initiale.

Exemple, avec $S = ABACABADA$, on décide de faire une extension d'ordre 3, on obtient les blocs d'événements ABA, CAB et ADA. Si on se définit un alphabet $\beta = \{ 'AAA', 'AAB', 'ABA', 'ADA', 'CAB', \dots \}$ basé sur α , on note ξ_3 l'extension d'ordre 3 de S et elle vaut " $\beta_2 \beta_4 \beta_3$ " (les indices commencent à zéro).

Evidemment, on voit que si l'alphabet ou le codage de destination n'est pas bien choisi (comme dans l'exemple précédent), on ne gagne rien, on peut même y perdre. On montre que l'entropie d'un symbole e_m d'une extension d'ordre m est $H(e_m) = m \cdot H(e)$. C'est à dire que si on groupe les symboles 3 par 3, l'entropie d'un triplet quelconque est le triple de l'entropie d'un événement isolé. On ne gagne donc rien à priori avec une extension sur un signal aléatoire. Il y a autant de désordre avec des triplets qu'avec des événements isolés.

Mais, l'idée est de trouver une bonne extension et un bon codage qui permettent de faire descendre l'entropie sur un signal markovien (= qui contient des corrélations). En particulier avec les sources markoviennes d'ordre m (rappel : un événement est lié à ses m voisins), il est intéressant de faire une extension d'ordre m car il est très probable que certaines séquences seront beaucoup plus fréquentes que d'autres, donc représenteront moins d'information.

Par exemple, la source ABACABAD, en extension $m=2$ génère AB/2, AC/1, AD/1, on obtient donc l'entropie globale : $\frac{2}{4} \times 1.0 + \frac{1}{4} \times 2.0 + \frac{1}{4} \times 2.0 = 1.5$ ce qui est inférieur à $\frac{4}{8} \times 1.0 + \frac{2}{8} \times 2.0 + \frac{1}{8} \times 3.0 + \frac{1}{8} \times 3.0 = 1.75$. Il faudra donc en moyenne 1 bit et demi pour l'extension de niveau 2 contre 1.75 pour le signal d'origine.

Codage d'une source

Le deuxième théorème de Shannon indique, pour simplifier, qu'il y a une limite minimale au nombre de bits nécessaires pour recoder un signal markovien d'ordre m et de longueur L . Dans l'exemple précédent, on a vu que $H(e_2) < H(e)$, on peut peut-être avoir $H(e_3) < H(e_2)$ et ainsi de suite mais $H(e_L)$ (c'est l'entropie d'une extension qui fait la longueur du signal) constitue la limite basse impossible à améliorer du nombre de bits nécessaires. On peut atteindre cette limite si on dispose des probabilités des événements dans le signal et des probabilités des séquences de différentes longueurs.

L'une des clés de la compression consiste à utiliser des codes à longueur variable VLC (variable length coding) : des codes courts pour les événements fréquents et des codes plus longs pour les événements rares. L'idée étant de se rapprocher de l'entropie pour la moyenne des longueurs de codes.

Algorithmes de compression

Nous allons étudier plusieurs sortes d'algorithmes de compression très connus et très intéressants. Certains (codages entropiques) ne font aucune hypothèse sur les données manipulées (source aléatoire), d'autres cherchent des corrélations dans le signal (source markovienne) et les derniers s'appuient fortement sur le type de données manipulées pour en extraire des corrélations à grande échelle et compresser les informations.

Méthode RLE

La méthode RLE s'apparente à une extension : on regroupe les événements d'une certaine manière et on code une propriété commune. RLE signifie : Run Length Encoding = codage des longueurs à suivre. On regroupe les événements identiques consécutifs (une plage uniforme) et on représente des couples <événement, nombre>.

Par exemple, soit la portion d'image en niveaux de gris suivante : 32, 32, 32, 32, 32, 34, 34, 36, 36, 36, 32, 32, 30, 30, 30, 30

Son codage en RLE donne : <32,5>, <34,2>, <36,3>, <32,2>, <30,4>

Il existe des variantes adaptées aux signaux qu'on traite. Par exemple dans le cas des sons, la variation des échantillons est en général progressive. Si on part non pas des échantillons, mais des écarts entre les échantillons consécutifs (la dérivée du signal), on peut dans certains cas faire apparaître des plages constantes.

Par exemple, soit la portion d'échantillons suivante : 30, 31, 32, 33, 34, 36, 38, 40, 37, 34, 31, 28, 28, 28 Son codage en RLE différentiel donne : <+1,4>, <+2,3>, <-3,4>, <0,2>. On doit seulement donner la valeur initiale (30 dans l'exemple).

La méthode RLE appliquée aux images

Idées

La technique consiste à coder les plages de pixels uniformes par une liste de couples <pixel, nombre d'occurrences consécutives>. Quand on a affaire à des bitmaps, on peut même se passer des pixels sachant que si on quitte une plage, on passe forcément dans la couleur opposée.

Il y a un problème de codage de ces couples : quel nombre de bits donner aux nombres d'occurrences ? Ca dépend de la longueur maximale d'une plage, mais pour le savoir, il faut connaître toutes les images qu'il y aura à coder. Il y a forcément une limite raisonnable correspondant à la majorité des plages, mais on trouvera certainement une plage plus longue qui demandera plusieurs couples. Coder une plage de $N > 2^n$ pixels consécutifs avec seulement n bits impose d'écrire une première plage de 2^n pixel puis une plage pour le reste...

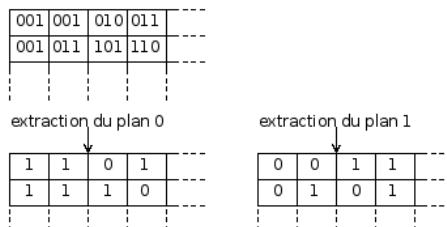
On considère une image comme un signal 1D en la balayant comme un livre : de haut en bas et de gauche à droite. On pourrait tout à fait employer un autre balayage : en colonnes ou en zig-zag.

Cette technique est intéressante pour des images présentant de grandes plages uniformes, telles que les images 2D vectorielles : figures géométriques remplies d'une couleur uniforme. Ce n'est pas le cas des images réelles qui sont bruitées. Il y a de faibles variations des pixels, mais pas de zone totalement identique ; dans ce cas, le codage sera plus coûteux que l'image d'origine.

Décomposition en plans bitmap

On s'intéresse ici aux images en niveaux de gris. On pourrait effectuer un codage RLE en cherchant des plages de nuances de gris identiques. Cependant, dans une image photographique, on a peu de chances de trouver beaucoup de pixels identiques à la suite.

Une des idées clés est de décomposer les images en bitmaps : 8 plans pour les images en niveaux de gris : on écrit chaque pixel en base 2 et on considère le bitmap qu'on obtient en extrayant le même bit sur chacun des pixels : l'ensemble des bits de poids N vont dans le bitmap n^N .



Voici l'algorithme écrit en C pour netpbm (on peut l'améliorer pour traiter les 8 plans d'un coup) :

Extraction d'un bitmap

```
bit **Plan(gray **image, int p)
// extrait le plan p de l'imageG et le place dans plan
{
    int masque = 1 << p;
    plan = pbm_allocarray(Largeur, Hauteur);
    for (int y=0; y<Hauteur; y++)
        for (int x=0; x<Largeur; x++) {
            gray point = image[y][x];           // entre 0 et 255
            if (point & masque) plan[y][x] = 0;   // blanc (convention pbm)
            else plan[y][x] = 1;                // noir
        }
    return plan;
}
```

Voici un exemple : une image en niveaux de gris.



Sa décomposition en 8 plans bitmaps donne les images suivantes, du plan 7 au plan 0 :



plan 7



plan 6



plan 5



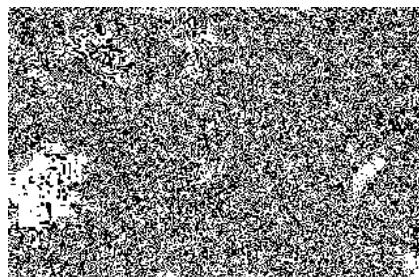
plan 4



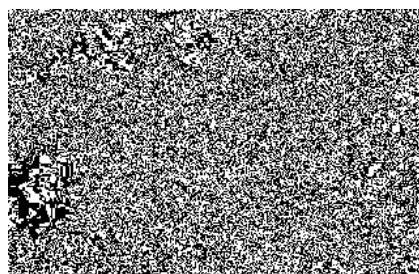
plan 3



plan 2



plan 1



plan 0

On constate que les dégradés affectent énormément les poids faible des données, mais assez peu les poids forts. Le poids faible est quasiment du bruit. Le bit de poids fort représente la luminosité de l'image. Les zones sombres ont toujours ce bit à 0, les zones claires l'ont à 1.

L'idée est d'appliquer RLE sur les 4 bitmaps de poids fort pour obtenir beaucoup de gain. On pourrait se passer du codage des 2 ou 3 plans de poids faible. Evidemment, on ne sait pas quoi faire des plans intermédiaires.

Voici le résultat de la reconstruction d'une image à partir de ses plans binaires en éliminant peu à peu les plans de poids faibles : on reconstruit l'image en prenant les plans $7 \rightarrow n$, n variant progressivement de 0 à 7 ; donc la première image ci-dessous est l'image comprenant tous les plans, la suivante ne contient pas le plan 0, la troisième ne contient pas les plans 0 et 1, et ainsi de suite jusqu'à la dernière qui ne contient que le plan de poids fort (les pixels sont soit 0 soit 128).



plan 0 à 7



plan 1 à 7



plan 2 à 7



plan 3 à 7



plan 4 à 7



plan 5 à 7



plan 6 et 7



plan 7 uniquement

Une idée pour coder quand même ces plans faibles : puisqu'on voit que ce sont des alternances de (0,1) et (1,0), on fait une extension d'ordre 2 (grouper les pixels par 2), ça donne 4 combinaisons (0,0), (0,1), (1,0) et (1,1) qu'on code par Huffman (voir plus loin) qui optimise la représentation d'une séquence aléatoire.

Codes de Gray

On peut toutefois constater que sur une image moyennement grise (niveaux aux alentours de 128), on risque de ne pas obtenir une plage uniforme même dans le bitmap de poids fort. Par exemple, si les couleurs varient entre 127 (01111111) et 128 (10000000). Autre remarque : que donne l'extraction de plans sur une image composée de dégradés noir-gris-blanc ?

L'idée est d'extraire les bitmaps autrement. Le code de Frank Gray (1953) a la propriété très intéressante que, entre deux valeurs consécutives quelconques, il n'y a qu'un seul bit qui change.

Avec le codage ordinaire, par exemple entre 6 (0...0110) et 7 (0...0111), il n'y a qu'un bit qui change. Par contre, entre 7 et 8, il y en a quatre qui changent de valeur. Avec le codage de Gray, entre n'importe quel nombre et son suivant, il n'y a qu'un seul bit qui change.

Par exemple, sur 4 bits :

Codes de Gray sur 4 bits

nbre	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	
code	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
(0)	(1)	(3)	(2)	(6)	(7)	(5)	(4)	(12)	(13)	(15)	(14)	(10)	(11)	(9)	(8)	

Pour passer d'un code à l'autre, voici des fonctions C :

Conversion Gray <--> entiers

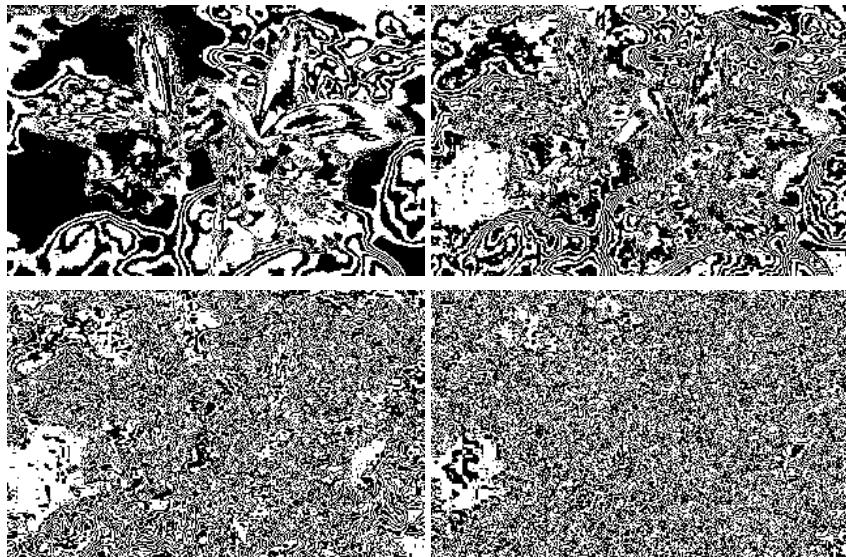
```
// grayencode(i) renvoie le code de Gray du nombre i
unsigned int grayencode(unsigned int i)
{
    return i ^ (i >> 1);
}

// graydecode(g) renvoie l'entier correspondant au code de Gray g
unsigned int graydecode(unsigned int g)
{
    unsigned int i;
    for (i = 0; g != 0; g >>= 1) {
        i ^= g;
    }
    return i;
}
```

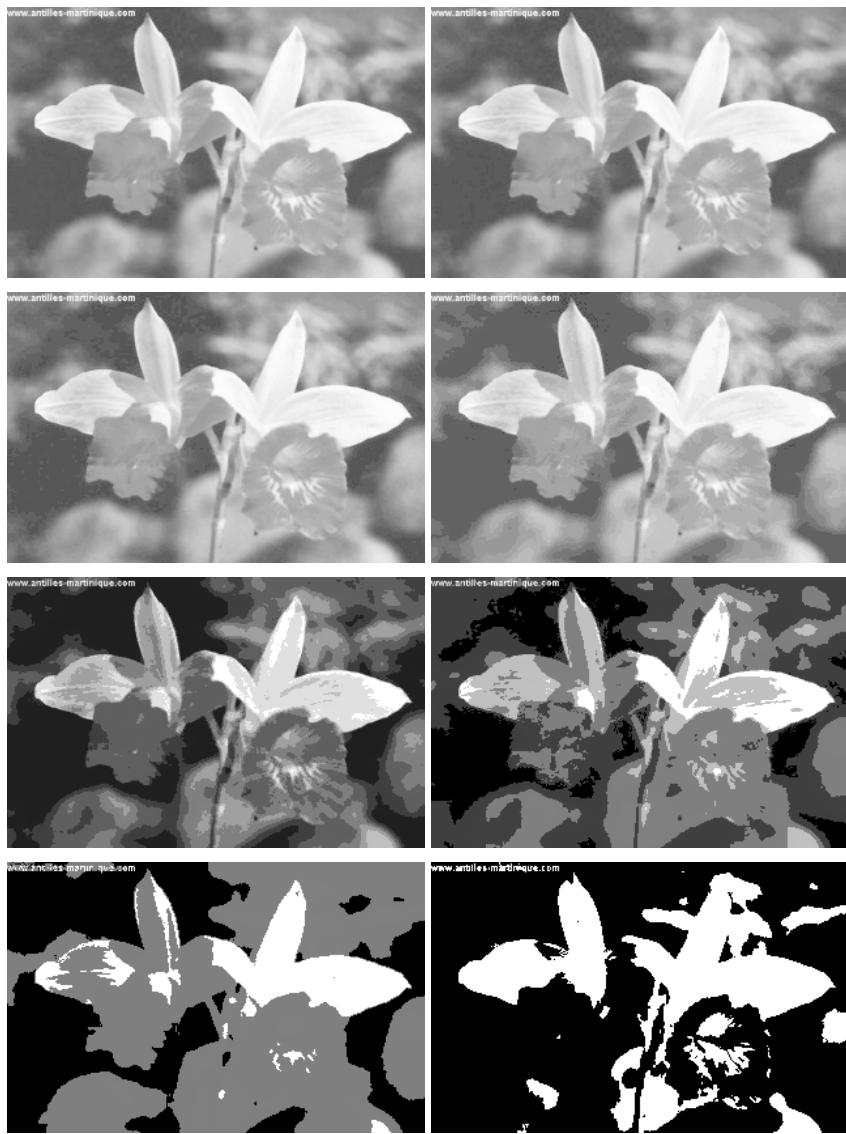
L'idée est de recoder tous les pixels en code de Gray et d'extraire les plans sur ces codes. L'intérêt est que sur des dégradés (des variations faibles entre les valeurs des pixels voisins), il n'y aura qu'un seul plan qui change et non pas tous les plans de poids faibles.

Voici ce que donne le traitement d'extraction des plans à partir des codes de Gray des pixels sur la même image que précédemment :





On voit nettement une amélioration pour RLE, il n'y a plus que les deux derniers plans qui sont réellement bruités -- donc coûteux à coder en RLE. On pourrait carrément les enlever sans qu'on voit de différence. Voici la reconstruction avec les codes de gray en éliminant progressivement les plans de poids faibles :



On voit que les trois plans de poids faibles sont peu significatifs, c'est eux qui coûtent le plus en codage RLE. Les autres plans extraits avec les codes de Gray sont intéressants à coder en RLE : la longueur des plages est d'autant plus longue que le poids est important. On peut employer un codage de longueur variable selon le plan : n bits pour le plan $n^{\circ}n$ ($n:2$ à 7), avec la convention qu'on change de valeur à chaque plage et qu'on commence avec une plage noire.

L'intérêt pédagogique de cette étude, c'est la décomposition du signal en différentes plans décorrelés : les redondances deviennent très faciles à coder. Les autres méthodes de compression étudiées dans la suite (JPEG) procèdent conceptuellement de même : passer dans un autre espace de représentation dans lequel les correlations sont plus simples à représenter.

Semaine 13 (05/12/2011) : Codages entropiques

Algorithmes entropiques = basés sur les fréquences d'apparition des symboles

Cette semaine, nous allons étudier quatre algorithmes pour compresser des données :

- deux algorithmes pour coder des sources aléatoires : Shannon-Fano et Huffman
- deux algorithmes pour coder des sources markoviennes : Lempel-Ziv et Lempel-Ziv-Welch

Les deux premiers algorithmes prennent des données aléatoires : sans corrélations entre elles. La seule information qu'on ait, c'est le nombre d'occurrences de chaque symbole. De ces informations on en déduit la probabilité, donc la quantité d'information ainsi que vu la semaine dernière. L'idée de base est d'attribuer un code binaire d'autant plus court que le symbole apparaît souvent dans le signal, et inversement. Un tel codage est appelé VLC : variable length code.

Les deux algorithmes suivants font l'hypothèse que les données sont redondantes : certaines séquences se répètent plus ou moins régulièrement, ce qui permet de leur attribuer un code spécifique bien plus court que les répétitions, ce qui réduit la taille occupée par l'ensemble.

Algorithm de Shannon-Fano

Cet algorithme ainsi que celui de Huffman transforme un signal aléatoire constitué d'événements représentés par des codes de longueur n (bits), n étant fixe, en un signal d'événements codés par un VLC (variable length coding) : un code à longueur variable. La compression réside dans la petitesse des codes de sortie par rapport aux codes d'entrée : on code les événements les plus fréquents avec un code très court et les événements les moins fréquents avec un code plus long.

Le signal de sortie est donc composé de symboles de taille variable mis bout à bout. Ca pose deux problèmes : d'abord construire ces codes et ensuite pouvoir relire le signal de sortie pour le reconstituer. On va commencer par le second problème.

VLC préfixé

Le code de sortie est composé de nombres binaires de longueur variable mis bout à bout sans aucun délimiteur. Il n'y a rien pour indiquer qu'on passe d'un code à l'autre. Comment extraire les codes sans connaître leur longueur ?

Voici un signal en exemple : 00010100101110001010010111. Voici une liste des codes qui ont été employés dans ce signal : A=01 B=010 C=101 D=0 (la convention est de s'arrêter de décoder quand on trouve un code inconnu). Quel est le message contenu dans la chaîne précédente ?

Il est impossible de proposer une solution unique, parce que le codage est mal fait : les codes sont ambigus. Quand on voit 01, on ne sait pas si c'est A seulement ou le début de B ou un D suivi d'un C.

La bonne méthode est de construire des codes non ambigus, qu'on appelle "**préfixés**" : on fait en sorte que chaque code ne puisse pas être le début d'un autre code. Dans l'exemple précédent, A est le début de B, D est le début de A et B et donc celui-là n'est pas préfixé.

Voici un code préfixé correct : A=101, L=001, S=000, T=11, U=01. On vérifie qu'aucun de ces codes n'est préfixe d'un autre. Quel est le message codé par le signal précédent ? Il faut commencer avec le premier bit, 0 c'est soit un S soit un L, ensuite un 0 il reste S ou L puis un 0, c'est donc un S. Ensuite on trouve un 1 : A ou T, puis un 0 donc c'est un A, il faut consommer le 1 suivant. Et ainsi de suite.

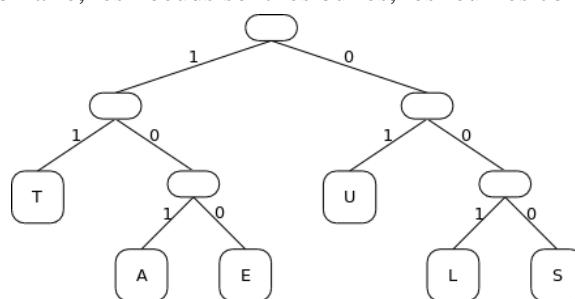
On se rend compte aussi que si le codage n'est pas bien fait, même en étant préfixé, on perd en qualité de compression. Par exemple le A contient 3 bits, le troisième bit est inutile car aucun autre code ne commence par 10.

Arbre binaire de codes préfixés

Les deux algorithmes (Shannon-Fano et Huffman) construisent chacun un codage optimal pour la compression mais de manière différente : on aboutit à des codes différents, mais au même taux de compression, très proche de ce qu'indique le théorème de l'entropie pour un signal aléatoire. Le principe est d'associer aux événements un code dont la longueur est fonction de la quantité d'information (inverse de la probabilité).

Pour construire ce code, les deux algorithmes utilisent un **arbre binaire** qui fabrique automatiquement un code préfixé. C'est tout simple : un arbre binaire est un arbre dont les noeuds sont soit réduits à des feuilles (pas de sous-branches), soit comptent exactement deux branches. Les branches sont marquées 0 et 1.

Voici un exemple d'arbre binaire, les noeuds sont les bulles, les feuilles contiennent les symboles :



Pour garantir de manière sûre que c'est bien un codage préfixé, il suffit d'interdire que les noeuds portent des symboles : seules les feuilles peuvent être des symboles codés.

Pour obtenir le code d'un symbole, il suffit de suivre les branches en partant de la racine. Par exemple, L = 001.

Pour décoder un signal, il suffit de lire événement par événement (bit par bit) et de choisir la branche qui correspond au bit lu. Quand on arrive à une feuille, on affiche cette feuille et on retourne à la racine. Evidemment, il faut que l'arbre binaire soit le même entre le codeur et le décodeur.

Les deux algorithmes présentés plus bas procèdent différemment pour créer cet arbre.

Algorithme de Shannon-Fano

Voici l'algorithme. On considère qu'on dispose de tout le signal avant de commencer la compression, ça permet de déterminer les occurrences de chaque symbole : on connaît le nombre d'apparition de chaque symbole.

Le principe est de faire deux groupes de symboles de manière à ce que les deux sommes des occurrences soient à peu près égales. L'un des groupes aura le préfixe 0, l'autre 1. Ensuite on recommence récursivement dans chacun des deux groupes : on répartit leurs symboles en deux sous-groupes de façon que les occurrences soient équilibrées...

Voici une écriture récursive simplifiée de cet algo :

```

Initialiser l'arbre binaire à vide
appeler la fonction CreerCodes("", tous les symboles)

def fonction CreerCodes(prefixe, groupe de symboles avec leurs occurrences):
    si le groupe ne contient qu'un seul symbole, alors sortir de la fonction, le
    code de ce symbole est : préfixe

```

```

couper le groupe en deux sous-groupes tels que les sommes des occurrences soit
équivalents (note n°1)
CreerCodes(prefixe+"0", premiergroupe)
CreerCodes(prefixe+"1", second groupe)

```

Note 1 : on veut que la somme des occurrences des symboles du premier sous-groupe soit à peu près égale à celle du second sous-groupe. Dans le cas général, c'est un problème NP-complet, on le retrouve dans l'algorithme du sac à dos : comment remplir un sac à doc avec des objets plus ou moins lourds sans dépasser une charge maximale.

Shannon et Fano ont proposé une solution simple basée sur un **algorithme glouton** (*greedy* en anglais). Un algorithme glouton essaie de résoudre des problèmes structurés en plusieurs étapes. Par exemple, au jeu de dames, il va devoir choisir de bouger l'un des pions... lequel ? A chaque étape, un algorithme glouton choisit ce qui paraît être la meilleure solution à ce moment-là. Par exemple, aux dames, il choisit le coup qui lui permet de manger le plus de pions adverses. Evidemment, selon le problème, un choix optimal localement peut se révéler mauvais globalement. Un exemple : pour aller au sommet du Mont-Blanc, on choisit, face à un carrefour d'aller toujours vers la route qui monte le plus. Avec cette stratégie, on peut rester tourner en rond en haut du Méné-Bré.

Ici, on a de la chance l'algorithme glouton suivant marche bien pour séparer les deux groupes de symboles :

```

classer les occurrences du groupe en nombre croissant ou décroissant
construire le premier sous-groupe en piochant les symboles dans l'ordre du classement
jusqu'à ce que la somme cumulée approche mais sans dépasser la moitié de la somme
des occurrences. Le second sous-groupe est constitué des éléments restants.

```

Ca fait un peu penser au choix des joueurs pour faire deux équipes de football au lycée : on répartit les meilleurs joueurs à tour de rôle dans les deux équipes afin d'équilibrer les forces. Seulement, ici, ce n'est pas le même objectif, on vise à donner les codes les plus courts aux "joueurs" les plus forts et à donner ce qui reste aux autres joueurs.

Exemple

Soit le signal $S = \text{"CITRONTRESCONTRIT"}$.

On commence par compter les occurrences : C/2, I/2, T/4, R/3, O/2, N/2, E/1, S/1, la somme des occurrences est bien $17=L$ longueur du signal.

Il faut faire deux groupes de symboles de manière à ce que leurs sommes soient d'environ $17/2$. Il y a généralement plusieurs possibilités. Par exemple, on peut proposer ce découpage : groupe 0 = { C/2, T/4, O/2, E/1 } et groupe 1 = { I/2, R/3, N/2, S/1 }. L'algorithme décrit dans la note 1 choisirait une autre combinaison décrite plus bas.

Ensuite dans le premier groupe, il faut parvenir à deux groupes de poids à peu près égaux : sous-groupe 00 = { C/2, O/2, E/1 } et sous-groupe 01 = { T/4 }. Dans l'autre groupe : sousgroupe 10 = { I/2, N/2 } et sousgroupe 11 = { R/3, S/1 }. Encore une fois, il y a d'autres possibilités pour découper en sous-groupes, on verra après la conséquence sur le taux de compression.

Il reste encore des groupes non réduits à des singltons, on peut les découper ainsi : sous-sous-groupe 000 = { C/2 }, sous-sous-groupe 001 = { O/2, E/1 }, sous-sous-groupe 100 = { I/2 }, sous-sous-groupe 101 = { N/2 }, sous-sous-groupe 110 = { R/3 }, sous-sous-groupe 111 = { S/1 }.

On répète encore pour obtenir : sous-sous-sous-groupe 0010 = { O/2 }, sous-sous-sous-groupe 0011 = { E/1 }.

L'algorithme s'arrête car tous les groupes sont réduits à des singltons. On obtient donc le codage :

$C = 000, I = 100, T = 01, R = 110, O = 0010, N = 101, E = 0011, S = 111$.

Le signal S est codé = "000100011100010101011100001111100000101010111010001". On vérifiera qu'on peut le décoder.

Même exemple avec application de la note 1

On lance l'algorithme avec le groupe $\{ C/2, I/2, T/4, R/3, O/2, N/2, E/1, S/1 \}$ et on veut appliquer l'algorithme glouton pour choisir les groupes. On commence donc par trier la liste dans l'ordre décroissant des occurrences : $\{ T/4, R/3, C/2, I/2, O/2, N/2, E/1, S/1 \}$. Il n'y a pas d'ordre fixe pour les symboles qui ont les mêmes occurrences.

Ensuite on pioche dans cette liste les premiers éléments jusqu'à obtenir un total de $17/2 = 8$. Ca donne les groupe 0 = $\{ T/4, R/3 \}$ et groupe 1 = $\{ C/2, I/2, O/2, N/2, E/1, S/1 \}$. Ensuite on obtient sous-groupe 00 = $\{ T/4 \}$ et sous-groupe 01 = $\{ R/3 \}$, on voit ainsi que les deux caractères les plus fréquents sont codés avec peu de bits.

Les autres groupes : sous-groupe 10 = $\{ C/2, I/2 \}$, sous-groupe 11 = $\{ O/2, N/2, E/1, S/1 \}$, sous-sous-groupe 100 = $\{ C/2 \}$, sous-sous-groupe 101 = $\{ I/2 \}$, sous-sous-groupe 110 = $\{ O/2 \}$, sous-sous-groupe 111 = $\{ N/2, E/1, S/1 \}$. Encore un niveau de récursivité pour les derniers : sous-sous-sous-groupe 1110 = $\{ N/2 \}$, sous-sous-sous-groupe 1111 = $\{ E/1, S/1 \}$ et finalement sous-sous-sous-sous-groupe 11110 = $\{ E/1 \}$, sous-sous-sous-sous-groupe 11111 = $\{ S/1 \}$.

L'algorithme s'arrête car tous les groupes sont réduits à des singletons. On obtient donc le codage :

$C = 100, I = 101, T = 00, R = 01, O = 110, N = 1110, E = 11110, S = 11111$.

Le signal S est codé = "10010100011101100001111011111001101110000110100". Il fait la même taille que le précédent. Seul le codage est différent, mais de même rendement comme on a utilisé un algorithme qui génère un code compact (optimal quelque soit le choix de représentation). Ce qu'on perd sur un code, on le regagne sur un autre. Les codes à 5 bits sont compensés par des codes à 2 bits plus nombreux.

Entropie de ce message

On rappelle que l'entropie de ce message $H(e)$ indique le nombre de bits ($n_\beta=2$) nécessaires en moyenne pour coder l'un des événements de ce signal. Elle est calculée par :

$$H(e) = \sum_{i=0}^{n_\alpha-1} p(\alpha_i)I(\alpha_i) = \sum_{i=0}^{n_\alpha-1} p(\alpha_i) \log_2 \left(\frac{1}{p(\alpha_i)} \right)$$

Le programme python du cours précédent donne la valeur 3.02832083357. NB : certains symboles sont codés avec moins, d'autres avec plus, mais en moyenne c'est ce nombre. Comme le message fait 17 événements, en principe, on ne peut pas descendre en dessous de $17*3.028 = 51.48$ bits. On est à 50, donc au mieux de la compression possible en considérant le message comme composé de caractères totalement indépendants entre eux -- ce qui n'est évidemment pas le cas...

Algorithme de Huffman

Dans l'algorithme de Huffman, on procède différemment pour un résultat comparable. On commence par choisir les deux symboles qui ont le moins d'occurrences, on leur donne les codes 0 et 1 et on les regroupe dans un arbre binaire auquel on attribue un nombre d'occurrences : la somme des deux symboles qu'il regroupe. Ensuite, on continue : on choisit deux symboles ou arbres qui ont le moins d'occurrences et on les regroupe. On regroupe ainsi deux par deux des symboles ou des arbres en fonction du nombre d'occurrences qu'ils représentent.

Exemple de codage Huffman

Soit le signal S = "CITRONTRESCONTRIT".

On commence par l'ensemble : $\{ C/2, I/2, T/4, R/3, O/2, N/2, E/1, S/1 \}$.

L'algorithme groupe les deux symboles ou arbres les moins représentés : E/1 et S/1 et crée un arbre pour eux : $\{ E, S \}/2$. Dans cette notation, on met la branche 0 à gauche, la branche 1 à droite.

On continue donc avec $\{ C/2, I/2, T/4, R/3, O/2, N/2, \{E, S\}/2 \}$. On a le choix pour grouper les moins représentés : le C et le I, le O avec le ES, etc. Le premier choix donne : $\{ \{C, I\}/4, T/4, R/3, O/2, N/2, \{E, S\}/2 \}$.

On continue et on obtient $\{ \{C, I\}/4, T/4, R/3, \{O, N\}/4, \{E, S\}/2 \}$.

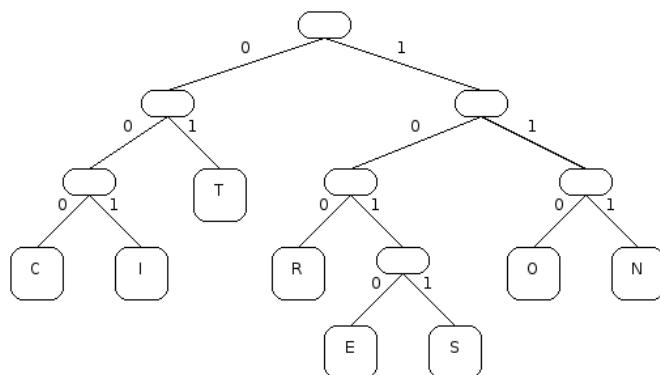
Un nouveau regroupement du R et ES donne $\{ \{C, I\}/4, T/4, \{R, \{E, S\}\}/5, \{O, N\}/4 \}$.

On continue : $\{ \{\{C, I\}, T\}/8, \{R, \{E, S\}\}/5, \{O, N\}/4 \}$

Encore : $\{ \{\{C, I\}, T\}/8, \{\{R, \{E, S\}\}, \{O, N\}\}/9 \}$.

Et la dernière étape car il ne reste plus que deux sous-groupes : $\{ \{\{C, I\}, T\}, \{\{R, \{E, S\}\}, \{O, N\}\} \}/17$.

A lire ainsi :



On obtient donc les codes suivants :

$C = 000, I = 101, T = 01, R = 100, O = 110, N = 111, E = 1010, S = 1011$.

Le signal S est codé = "0001010110011011101100101010110001101110110010101" sur 49 bits. Ce qui fait 1 bit de moins qu'avec le codage de Shannon. C'est dû aux choix qu'on a fait de regrouper tel ou tel symbole. Dans d'autres exemples, les deux procédés sont équivalents.

Remarques finales sur ces deux algorithmes

En principe, le codage de Huffman donne toujours le meilleur résultat possible en termes de compression d'un signal aléatoire. Avec ces algorithmes, les données compressées doivent être accompagnées de la table ou arbre de codage. La compression n'est rentable que si la représentation des deux : données et arbre est plus petite que les données d'origine. On peut imaginer que pour des données similaires, ex des articles de journaux en français, on puisse avoir la même table de codage basée non pas sur les statistiques d'apparition des lettres et non pas une table de codage par article. On y perdrait sur certains articles mais on éviterait de stocker une table dans chaque article.

Ces deux méthodes travaillent sur des données totalement aléatoires et produisent une compression optimale en terme d'entropie. On peut encore améliorer le codage si on parvient à détecter des corrélations dans le signal : des répétitions ou des liens entre les événements. Les algorithmes Lempel-Ziv mémorisent les séquences d'événements et codent de manière économique les répétitions exactes.

Algorithme Lempel-Ziv 77

La famille d'algorithmes Lempel-Ziv est basée sur l'étude de répétitions de symboles. On peut en voir quelques unes dans le signal S = "CITRONTRESCONTRIT" : TR, ON sont répétés 3 et 2 fois, ONTR est répété deux fois.

Admettons qu'on ait une liste de codes VLC préfixés déjà établis pour des séquences de deux ou trois caractères comme : CI=10, TR=00, ON=01, ESC=110, IT=111. Le signal se code alors : "100001001100100111" ce qui est très en dessous de l'entropie. La raison est qu'on n'a plus affaire à un signal aléatoire, il y a de fortes corrélations entre les événements.

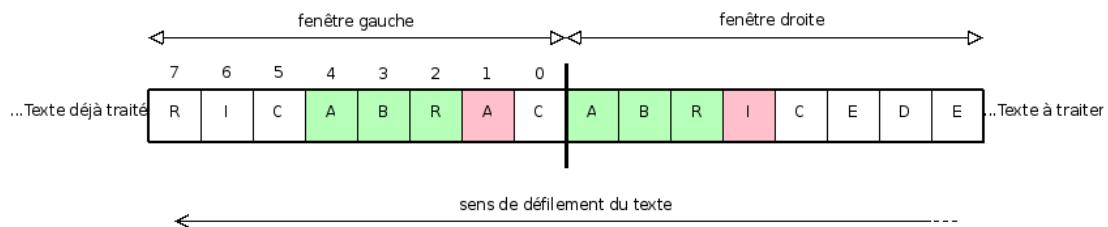
Le problème est de pouvoir construire une telle liste. Comment regrouper les lettres de manière optimale ? Les algorithmes LZ fournissent une solution intéressante.

On va commencer par présenter la version initiale de l'algorithme : LZ77 proposé par Lempel et Ziv en 1977.

Principe de LZ77

On fait passer le signal à coder dans une fenêtre de $2 \times N$ événements. Le texte défile de droite à gauche dans ce dispositif. L'idée est de regarder si le début de la partie droite se trouve dans la partie gauche. Si c'est le cas, on code la position de la sous-chaine et sa longueur.

Voici une illustration :



Le caractère courant est juste à droite du trait gras et on vient de voir tous ceux qui sont à gauche. On va comparer le **début** de la partie droite à la partie gauche et chercher des séquences identiques. C'est à dire qu'on va examiner les caractères à traiter en essayant d'en retrouver le plus possible à la suite dans la fenêtre de gauche.

Par exemple, si à gauche on a "BRICABRAC" et à droite "ABRICEDE" : on trouve les 3 premiers caractères de droite dans la fenêtre de gauche. On ne doit pas considérer la chaîne RIC située à droite du A car elle n'est pas en **début** de fenêtre.

Par contre dans un autre cas, à gauche : "CITRONTRESC" et à droite "ONTRIT", on trouve 4 caractères dans la fenêtre de gauche.

L'algorithme repose donc sur une recherche de chaînes dans une autre : on doit trouver la plus grande partie de la fenêtre de droite quelque part dans la partie gauche. Une correspondance est notée par sa position relativement au dernier caractère traité (celui qui est juste à gauche du trait gras), et sa longueur. Si on ne trouve rien, c'est noté 0,0.

Codage des éléments

A l'issue de la recherche de correspondance, on récupère un couple <position, longueur> éventuellement <0,0>.

Par exemple, si à gauche on a "RICABRAC" et à droite "ABRICEDE", on obtient <4, 3> : trois caractères à partir du numéro 4 (le C de droite dans la fenêtre de gauche est numéroté 0).

Ce signifie implicitement que le caractère longueur+1 à droite ne correspond pas. On le rajoute au codage : <4,3,T>.

L'algorithme LZ77 produit ainsi des triplets en sortie : <pos, lng, carsvt> :

pos

l'indice de 0 à Ng-1 de la partie de droite trouvée à gauche

lng

la longueur du segment de droite trouvé à gauche

carsvt

le premier caractère de droite qui n'a pas été trouvé à gauche.

Ces nombres sont codés en binaire et taille fixe, par exemple avec une fenêtre de 8 événements de type caractères, il faut 3 bits pour la position et la longueur, et 8 pour le caractère suivant.

Déroulement de LZ77 sur un exemple

On veut traiter le signal $S = "CITRONTRESCONTRIT"$ avec une fenêtre de 8 caractères. Voici les étapes de calcul.

1. gauche = "", droite = "CITRONTRESCONTRIT": on code un $<0,0,'C'>$ (un C est à mettre à l'index 0 de la partie gauche) et on avance de 1 position
2. gauche = "C", droite = "ITRONTRESCONTRIT": on code $<0,0,'T'>$ et on avance de 1 caractère
3. gauche = "CI", droite = "RONTRESCONTRIT": on code $<0,0,'T'>$
4. gauche = "CIT", droite = "ONTRESCONTRIT": on code $<0,0,'R'>$
5. gauche = "CITR", droite = "NTRESCONTRIT": on code $<0,0,'O'>$
6. gauche = "CITRO", droite = "TRESCONTRIT": on code $<0,0,'N'>$
7. gauche = "CITRON", droite = "TRESCONTRIT": on détecte 2 caractères dans la fenêtre on code $<3,2,'E'>$ et on avance de 3 positions, en ne gardant que les 8 derniers caractères à gauche
8. gauche = "ITRONTRE", droite = "SCONTRIT": on code $<0,0,'S'>$, on avance mais on ne mémorise que 8 caractères à gauche
9. gauche = "TRONTRES", droite = "CONTRIT": on code $<0,0,'C'>$ car le C a disparu de la fenêtre gauche
10. gauche = "RONTRESC", droite = "ONTRIT": on trouve une répétition en 6, on code $<6,4,'I'>$ et on avance de 5
11. gauche = "ESCONTRI", droite = "T": on code $<2,1,"">$ et c'est fini
12. gauche = "SCONTRIT", droite = "": c'est fini

Chaque code prend $3+3+8 = 14$ bits, il en faut donc 154 pour coder ce message au lieu de $17*8 = 136$ sans compression. La différence est dûe au trop peu de redondance dans ce message et au coût de représentation des triplets.

Autre exemple plus intéressant

On veut traiter le signal $S = "BRICABRACABRICEDENICE"$ avec une fenêtre de 8 caractères également. Voici les étapes de calcul.

1. gauche="", droite = "BRICABRACABRICEDENICE": code $<0,0,'B'>$
2. gauche="B", droite = "RICABRACABRICEDENICE": code $<0,0,'R'>$
3. gauche="BR", droite = "ICABRACABRICEDENICE": code $<0,0,'T'>$
4. gauche="BRI", droite = "CABRACABRICEDENICE": code $<0,0,'C'>$
5. gauche="BRIC", droite = "ABRACABRICEDENICE": code $<0,0,'A'>$
6. gauche="BRICA", droite = "BRACABRICEDENICE": code $<4,2,'A'>$
7. gauche="BRICABRA", droite = "CABRICEDENICE": code $<4,4,'I'>$
8. gauche="BRACABRI", droite = "CEDENICE": code $<4,1,'E'>$
9. gauche="ACABRICE", droite = "DENICE": code $<0,0,'D'>$
10. gauche="CABRICED", droite = "ENICE": code $<1,1,'N'>$
11. gauche="BRICEDEN", droite = "ICE": code $<5,3,"">$

Cela fait $11 * 14 = 154$ bits au lieu de $21*8 = 168$ en code ascii. On commence à voir l'intérêt de la compression. Pour l'instant, avec des sources aussi petites, la mécanique de codage est coûteuse, mais

plus la fenêtre est longue, plus on trouve de redondances. Noter que LZ77 n'est pas l'algorithme le plus performant de cette famille, mais le plus simple.

Pour comparaison, Huffman a besoin de seulement 61 bits pour coder ce même signal (hors codage du dictionnaire) mais le coût est fonction linéaire de la taille du message quelque soient ses redondances, tandis que LZ77 peut obtenir de meilleurs résultats sur des messages très redondants. D'autre part, dans Huffman, on doit rajouter le dictionnaire aux données, sauf si c'est un dictionnaire standard, tandis qu'avec LZ77, il n'y a pas de dictionnaire, tout le signal est contenu dans le recodage.

On perçoit bien que Huffman est adapté aux sources aléatoires de symboles redondants et LZ77 aux sources markoviennes d'ordre assez petit.

Un applet pour illustrer l'algorithme LZ77

[Cette page](#) (applets/lz77.htm) permet de tester l'algorithme pas à pas. L'ouvrir dans un nouvel onglet du navigateur.

Algorithme LZW : une amélioration de LZ77

L'algorithme LZ77 a été amélioré car il a plusieurs défauts qu'on voit déjà bien sur les exemples :

- trop peu de mémoire des séquences redondantes : il ne garde que les N derniers caractères.
- On ne sait coder que des séquences de la taille de la fenêtre : on ne peut pas compresser des données plus grandes que cette fenêtre.

La technique proposée par Welch s'inspire d'une amélioration proposée par Lempel et Ziv en 1978, consiste à remplacer la fenêtre de gauche par un dictionnaire : un tableau des chaînes déjà rencontrées dans le message à compresser. Ce dictionnaire est bien plus vaste que simplement les N derniers caractères rencontrés.

Description de LZW

La structure de données la plus importante est ce dictionnaire. C'est un tableau qui mémorise les séquences déjà rencontrées. On attribue une case numérotée à chaque séquence. Le tableau est préinitialisé avec tous les symboles de l'alphabet du signal à coder. Il n'y a aucune hypothèse faite sur les probabilités des symboles, donc aucune optimisation à ce sujet.

Dictionnaire utilisé par LZW pour représenter les séquences rencontrées

indice ou adresse	chaîne mémorisée	commentaire
0..N	N éléments prédefinis	la table est initialisée avec un certain nombre de symboles prédefinis, par exemple les lettres de l'alphabet ou les nuances de gris des pixels, ou les couleurs de la palette
N+1...	les séquences qu'on a rencontrées au cours de la compression	on rajoute toutes les séquences qu'on rencontre

Pour écrire l'algorithme, on se définit trois méthodes sur le dictionnaire : `contient(séquence)` renvoie vrai si le dictionnaire contient cette séquence, `position(séquence)` renvoie l'indice de cette séquence ou -1 si elle est absente du dictionnaire, `ajouter(séquence)` ajoute la séquence dans la première position libre, sauf si elle fait déjà partie du dictionnaire.

Le dictionnaire étant un sorte de tableau, on fait appel aux [] pour obtenir un élément donné par son indice. NB : il n'est pas recommandé de représenter le dictionnaire par un tableau statique, car il faut

pouvoir faire varier sa taille et il faut disposer d'une méthode de recherche très rapide car c'est sur elle que repose la rapidité du compresseur.

En résumé : l'algorithme mémorise dans le dictionnaire toutes les séquences qu'il rencontre et en sortie, il produit un code qui est leur indice dans le tableau, c'est ce qui réalise la compression. Voici maintenant l'algorithme complet :

Codeur LZW

```
initialiser le dictionnaire avec tous les symboles possibles, ex : tous les codes ascii
précédente := vide
TQ c := lire 1 caractère
    sequence := précédente + c
    si dictionnaire.contient(sequence) alors
        précédente := sequence
    sinon
        produire le code dictionnaire.position(précédente)
        dictionnaire.ajouter(sequence)
        précédente := c
    finsi
finTQ
produire le code dictionnaire.position(précédente)
```

Analyse de l'algorithme

Le codeur découpe la chaîne d'entrée en sous-chaînes $s = <g, d>$, telles que chacune soit composée d'une partie g la plus grande possible parmi celles déjà rencontrées c'est à dire que g est égale à une précédente sous-chaîne $s' = g' + d'$, et d'un caractère d suivant, ce caractère étant le premier de la sous-chaîne suivante. Au début, les seules sous-chaînes connues sont les événements isolés.

Par exemple, la chaîne ABCABCABCABCABC sera découpée en : $<A, B>$, $<B, C>$, $<C, A>$, $<AB, C>$, $<CA, B>$, $<BC, A>$, $<ABC, A>$, $<A, eof>$. C'est le seul découpage possible selon cette règle. Pour l'établir, il faut commencer par remplir le dictionnaire avec la liste des événements isolés : A tout seul, B tout seul... Ensuite, on commence à traiter la chaîne : AB... donne $<A, B>$ mais pas $<AB, C>$ car AB n'a pas encore été vu. On rajoute donc AB dans le dictionnaire des chaînes connues et on doit repartir de B et chercher une chaîne connue commençant par B, BC n'est pas connue, donc ça s'arrête là : $<B, C>$ (on rajoute BC dans le dictionnaire). On obtient $<C, A>$ par le même raisonnement. Ensuite on repart de A, comme AB est connue, on peut avoir $<AB, C>$ (ABC étant mis dans le dictionnaire). Et ainsi de suite.

Le codeur produit tout simplement les codes des parties g de ces sous-chaînes et ils sont stockés dans le dictionnaire.

Le décodeur doit reconstruire le dictionnaire en examinant les codes qu'il reçoit. Ces codes sont les indices des sous-chaînes rencontrées au cours de la lecture des données initiales. Un problème se pose avec une chaîne telle que ABBBBBBC, elle se découpe en $<A, B>$, $<B, B>$, $<BB, B>$, $<BBB, C>$, $<C, eof>$. Or il se trouve que le code de la séquence BB n'est pas encore dans le dictionnaire au moment où le décodeur le rencontre. C'est un petit problème dont on va voir la solution plus loin.

En attendant, voici l'algorithme du décodeur :

Décodeur LZW

```
initialiser le dictionnaire avec tous les symboles possibles, exactement ceux du codeur
précédente := vide
TQ code := lire 1 code
    si code hors du dictionnaire alors
        courante := précédente + premier caractère de précédente
    sinon
        courante := dictionnaire[code]
    finsi
    afficher courante
    c := premier caractère de courante (celui de gauche)
```

```

nouvelle := précédente + c
si non dictionnaire.contient(nouvelle), alors dictionnaire.ajouter(nouvelle)
pécédente := courante
finTQ

```

Ce décodeur traite les codes un par un. Chaque code a été produit par le codeur. C'est l'indice d'une sous-chaine g présente dans le dictionnaire du codeur au moment où il rencontre une séquence $\langle g, d \rangle$. Le décodeur doit donc remplacer l'indice qu'il reçoit par la séquence g présente dans le dictionnaire, puis ajouter $\langle g, d \rangle$ dans le dictionnaire. Il se trouve qu'il y a un souci : le décodeur ne connaît pas encore l'événement d . Le codeur l'a vu, mais pas le décodeur. Par contre, au code suivant, on sait que d se trouve en tête du prochain code. C'est donc seulement au prochain code que le décodeur pourra remplir son dictionnaire.

Pour bien comprendre pourquoi le décodeur utilise le premier caractère (celui de gauche) des séquences, il faut se souvenir que le codeur découpe la chaîne d'entrée en sous-chaînes $s_1 = \langle g_1, d_1 \rangle$, $s_2 = \langle g_2, d_2 \rangle$, mais avec d_1 étant le premier caractère de g_2 . C'est de ce caractère que repart le codeur quand il crée une nouvelle entrée dans le dictionnaire.

L'un des points clés concerne le test du code : s'il est hors du dictionnaire, c'est parce que le codeur avait rencontré une répétition de la même séquence précédente, comme dans ABBB : le troisième B. En effet, le codeur a toujours une entrée d'avance sur les données : il remplit son dictionnaire avec une nouvelle séquence mais n'affiche le code d'une partie précédente qu'à l'itération suivante. Dans le cas où la séquence précédente est la même que la courante, le code est celui de la dernière entrée du dictionnaire, mais le décodeur ne la connaît pas encore puisque c'est après avoir décodé qu'il remplit le dictionnaire. Par contre, en voyant un tel code, le décodeur peut deviner que c'est l'entrée g en cours de décodage qui est répétée. Donc, son dernier caractère d est égal au premier.

Ecriture des codes

Comment le codage est-il enregistré en sortie = sur combien de bits sont écrits les codes ? On pourrait décider que les codes, c'est à dire les indices dans le dictionnaire, sont enregistrés avec un nombre constant de bits, le nombre nécessaire pour coder toutes les cases du tableau. Si on fixe une borne supérieure au dictionnaire : N_{max} , il faut m bits, tels que $N_{max} \leq 2^m$.

Cependant, le dictionnaire n'a pas une taille fixe et sa taille n'est pas limitée. Au début, le dictionnaire ne contient que quelques séquences, il faut peu de bits pour coder les indices. Ensuite, le nombre de séquences augmente, il faut davantage de bits pour coder un indice. On voit bien que l'algorithme ne génère pas n'importe quels codes : à un moment donné, il ne peut générer que des codes \leq taille du tableau "actuel". On peut donc se contenter du nombre minimum de bits pour désigner une entrée du tableau. A tout moment, on n'utilise que des codes sur n bits, n étant tel que taille du tableau $\leq 2^n$. Quand le tableau s'agrandit, on augmente le nombre de bits écrits sur la sortie.

Il faut impérativement que le décodeur suive cette évolution : au début il va recevoir des codes sur 5 bits (fonction des 26 éléments présents au départ), son dictionnaire va grossir, dès qu'il arrive à 32 entrées, il est susceptible de recevoir des codes à 6 bits, méditer la ligne 9 des deux exemples suivants. Encore une fois, il y a un décalage entre le codeur et le décodeur.

Déroulement de LZW sur un exemple

On veut traiter le signal $S = "CITRONTRESCONTRIT"$. On décide que le dictionnaire est initialement rempli avec les lettres majuscules ($N=26$) : $dico[0] = A \dots dico[25] = Z$. Voici les étapes de l'algorithme.

étapes de l'algorithme LZW

étape	préc. au début du TQ	caractère lu : c	séquence	dans dico ?	décision	dico	code	préc. final

1	vide	C	C	oui				C
2	C	I	CI	non	ajout	dico[26] = CI	2/5	I
3	I	T	IT	non	ajout	dico[27] = IT	8/5	T
4	T	R	TR	non	ajout	dico[28] = TR	19/5	R
5	R	O	RO	non	ajout	dico[29] = RO	17/5	O
6	O	N	ON	non	ajout	dico[30] = ON	14/5	N
7	N	T	NT	non	ajout	dico[31] = NT	13/5	T
8	T	R	TR	oui				TR
9	TR	E	TRE	non	ajout	dico[32] = TRE	28/5 !!	E
10	E	S	ES	non	ajout	dico[33] = ES	4/6	S
11	S	C	SC	non	ajout	dico[34] = SC	18/6	C
12	C	O	CO	non	ajout	dico[35] = CO	2/6	O
13	O	N	ON	oui				ON
14	ON	T	ONT	non	ajout	dico[36] = ONT	30/6	T
15	T	R	TR	oui				TR
16	TR	I	TRI	non	ajout	dico[37] = TRI	28/6	I
17	I	T	IT	oui				IT
18	IT	<eof>	IT				27/6	

Il faudra donc $5+5+5+5+5+5+6+6+6+6+6+6 = 71$ bits pour coder ce message au lieu de $17*8 = 136$ sans compression.

C'est mieux que LZ77 mais il ne faut pas conclure sur ce seul exemple que LZW est le meilleur. LZ77 est employé dans le codage des images PNG et LZW est au coeur du codage GIF.

Déroulement de LZW sur un autre exemple

On veut traiter le signal S = "BRICABRACABRICE DENICE". Voici les étapes de calcul.

étapes de l'algorithme LZW

étape	préc. au début du TQ	caractère lu : c	séquence	dans dico ?	décision	dico	code	préc. final
1	vide	B	B	oui				B

2	B	R	BR	non	ajout	dico[26] = BR	1/5	R
3	R	I	RI	non	ajout	dico[27] = RI	17/5	I
4	I	C	IC	non	ajout	dico[28] = IC	8/5	C
5	C	A	CA	non	ajout	dico[29] = CA	2/5	A
6	A	B	AB	non	ajout	dico[30] = AB	0/5	B
7	B	R	BR	oui				BR
8	BR	A	BRA	non	ajout	dico[31] = BRA	26/5	A
9	A	C	AC	non	ajout	dico[32] = AC	0/5 !	C
10	C	A	CA	oui				CA
11	CA	B	CAB	non	ajout	dico[33] = CAB	29/6	B
12	B	R	BR	oui				BR
13	BR	I	BRI	non	ajout	dico[34] = BRI	26/6	I
14	I	C	IC	oui				IC
15	IC	E	ICE	non	ajout	dico[35] = ICE	28/6	E
16	E	D	ED	non	ajout	dico[36] = ED	4/6	D
17	D	E	DE	non	ajout	dico[37] = DE	3/6	E
18	E	N	EN	non	ajout	dico[38] = EN	4/6	N
19	N	I	NI	non	ajout	dico[39] = NI	13/6	I
20	I	C	IC	oui				IC
21	IC	E	ICE	oui				ICE
22	ICE	<eof>					35/6	

Il faut 83 bits pour coder ce message, contre 61 si on utilise Huffman, mais encore une fois, c'est parce que le message est trop court pour faire apparaître de grandes redondances.

Un autre constat est que le dictionnaire se remplit d'entrées qui paraissent peu utiles. Mais il n'y a aucun moyen de déterminer l'importance des séquences. Cela implique que les codes deviennent de plus en plus longs, inutilement donc, mais on n'y peut rien.

Décodage d'un message LZW

Soit la source suivante : "0000010100001110110" encodée en LZW avec un dictionnaire initial réduit aux 5 premières lettres : $dico[0] = 'A'$, $dico[1] = 'B'$, $dico[2] = 'C'$, $dico[3] = 'D'$, $dico[4] = 'E'$. Quel est le message initial ?

On commence par extraire le premier code. La table contient 5+1 entrées donc il suffit de 3 bits pour représenter ces entrées. On doit donc lire 3 bits qui sont 000. C'est celui du A qu'on produit en sortie (premier caractère décodé). On ne change pas le dictionnaire car A est déjà dedans. A la fin de la première boucle, précédent contient 'A'.

Le principe du remplissage du tableau : on commence par déterminer la taille du code à lire d'après le nombre d'entrées du dictionnaire, puis on lit un code qu'on doit forcément trouver dans le dictionnaire, on produit en sortie la case du dictionnaire désignée par ce code puis on doit ajouter une nouvelle entrée au dictionnaire, elle est formée du premier caractère du code courant et de la chaîne précédente. On passe à l'itération suivante en gardant le code courant.

étape	précédente	taille du dico => nbits	code	courante après test	c	nouvelle et dico
1	vide	5 entrées => code 3 bits	0/3	A	A	vide+A, $dico[0] = A$
2	A	5 => lire 3 bits	1/3	B	B	$dico[5] = AB$
3	B	6 => 3	2/3	C	C	$dico[6] = BC$
4	C	7 entrées => 3	0/3	A	A	$dico[7] = CA$
5	A	8 entrées => 4	7/4	CA	C	$dico[8] = AC$
6	CA	9 => 4	6/4	BC	B	$dico[9] = CAB$
7			eof	CA		

L'algorithme produit la séquence ABCACACBCCA.

Décodage d'un message avec l'exception

Soit la source suivante : "ABBBC" à encoder en LZW avec un dictionnaire initial réduit aux 5 premières lettres : $dico[0] = 'A'$, $dico[1] = 'B'$, $dico[2] = 'C'$, $dico[3] = 'D'$, $dico[4] = 'E'$.

Voici les étapes du codage :

étapes de l'algorithme LZW

étape	préc. au début du TQ	caractère lu : c	séquence	dans dico ?	décision	dico	code	préc. final
1	vide	A	A	oui				A
2	A	B	AB	non	ajout	$dico[5] = AB$	0/3	B
3	B	B	BB	non	ajout	$dico[6] = BB$	1/3	B
4	B	B	BB	oui				BB

5	BB	C	BBC	non	ajout	dico[7] = BBC	6/3	C
6	C	eof					2/3	

Ca donne donc la séquence "000001110010". Voici les étapes du décodage :

étape	précédente	taille du dico => nbits	code	test et courante	c	nouvelle et dico
1	vide	5 entrées => code 3 bits	0/3	ok : A	A	vide+A, dico[0] = A
2	A	5 => lire 3 bits	1/3	ok : B	B	dico[5] = AB
3	B	6 => 3 bits	6/3	hors dico : BB	B	dico[6] = BB
4	BB	7 entrées => 3 bits	2/3	C	A	dico[7] = CA
5	C		eof			

Une démonstration de l'algorithme

[Cette page](#) (applets/lzw.htm) permet de tester l'algorithme du compresseur LZW pas à pas et [celle-ci](#) (applets/ilzw.htm) permet de tester le décompresseur. Les ouvrir dans de nouveaux onglets du navigateur.

Programmation optimisée du codeur et décodeur LZW

Quand on programme l'algorithme et en particulier la gestion du dictionnaire, il y a une astuce importante qui permet d'économiser beaucoup de place en mémoire.

L'un des points importants concerne la gestion du dictionnaire. Il est hors de question de faire un tableau contenant les séquences intégrales car il y a beaucoup de redondances. En effet, on remarque que tous les ajouts dans le dictionnaire sont des séquences formées d'une partie "*précedent*" et d'un symbole. Or la partie *précedent* est toujours présente dans le dictionnaire, c'est l'indice de la précédente entrée traitée.

Pour plus de clarté, au lieu de stocker une séquence entière, comme CAB, on stocke un couple <indice de CA, B>. La partie gauche est le début de la séquence, la partie droite est la fin de séquence. La partie gauche est représentée par un indice, celui de la séquence CA dans le dictionnaire. Les entrées initiales ont une partie gauche vide, codée par -1.

Ca permet de gagner beaucoup de place en mémoire, mais on perd du temps quand il faut reconstituer la séquence entière ou quand il faut rechercher le symbole de gauche de la séquence. Il faut également modifier les algorithmes pour tenir compte de ces indices.

Par exemple, le dictionnaire de l'exemple précédent :

indice	0	1	2	3	4	5	6	7	8	9
séquence	A	B	C	D	E	AB	BC	CA	AC	CAB

sera représenté ainsi :

indice	0	1	2	3	4	5	6	7	8	9
gauche	-1	-1	-1	-1	-1	0	1	2	0	7

droite	A	B	C	D	E	B	C	A	C	B
--------	---	---	---	---	---	---	---	---	---	---

Le champ gauche qui est un indice vers une autre entrée, réalise de fait un chainage dans les entrées. La fonction SymboleGauche permet de trouver l'entrée située au début de la chaîne. Par exemple, SymboleGauche(9) renvoie C.

La fonction EcrireSequence(code) affiche toute la séquence. Par exemple, EcrireSequence(9) affiche CAB. Pour ces deux fonctions, une écriture récursive facilite les choses.

Voici une réécriture des algorithmes :

Algorithmes LZW avec tableau optimisé

```

typedef struct {
    int gauche;           // -1 si pas de précédent
    symbole droite;
} Entree;

Entree Dictionnaire[];

/* définir les opérations suivantes :
- InitialiserDictionnaire() :
efface toutes les entrées, ajoute les entrées prédéfinies,
selon l'application ça peut être tous les codes ascii ou toutes les nuances de gris.
- int ChercherEntree(gauche, droite) :
renvoie l'indice de la séquence <gauche,droite> ou -1 si elle est absente du
dictionnaire
- void AjouterEntree(gauche, droite) :
ajoute une nouvelle entrée dans le tableau
- SymboleGauche(code) :
cherche le symbole situé tout à gauche de la séquence d'indice code,
il faut remonter progressivement jusqu'à l'entrée dont gauche vaut -1.
- EcrireSequence(code) :
écrit tous les symboles de la chaîne d'indice code, dans l'ordre inverse du chainage,
c'est-à-dire en commençant par le symbole situé à l'extrémité gauche de la chaîne.
*/

```

```

procédure CodageLZW :
    InitialiserDictionnaire()
    precedent := -1
    pour chaque symbole à coder placé dans courant :
        connu := ChercherEntree(precedent, courant)
        si connu > 0 :
            precedent := connu
        sinon :
            écrire le code(precedent)
            AjouterEntree(precedent, courant)
            precedent := courant
        finsi
    finpour
    écrire le code(precedent)
fin

procédure DecodageLZW :
    InitialiserDictionnaire()
    precedent := lire un code
    EcrireSequence(precedent)
    TantQue pas fin de fichier
        code := lire un code
        si code >= NombreEntrees :
            courant := PremierCode(precedent)
        sinon :
            courant := code
        finsi

```

```
AjouterEntree(precedent, PremierCode(courant))
EcrireSequence(code)
precedent := code
finTantQue
fin
```

Semaine 14 (12/12/2011) : Principes des formats GIF et PNGs

Le format GIF

On va s'intéresser à un format d'images très connu bien que précédemment soumis à droits d'auteurs, le format GIF (Graphics Interchange Format) présenté par la société Compuserve. Son intérêt est d'employer une variante de la compression LZW sur des images indexées. Il est donc performant sur des images comprenant de nombreuses répétitions de séquences identiques, telles que les dessins 2D générés par ordinateur.

Concepts

On rappelle le principe de LZW : un dictionnaire contient des séquences de données déjà connues ou observées précédemment. L'algorithme produit des codes, ce sont les indices des séquences dans ce dictionnaire. Face à une nouvelle donnée à coder et à partir d'un code précédent, il concatène ce code avec la donnée, rajoute cet ensemble dans le dictionnaire. Ainsi le dictionnaire se remplit peu à peu de toutes les suites qui ont été observées.

Dans le cas GIF, la convention a été prise de limiter le dictionnaire à 4096 entrées. Un code spécial ("clear code") est produit par le codeur pour prévenir le décodeur que le dictionnaire doit être vidé et qu'il faut repartir avec le dictionnaire de base.

Pour signaler la fin du fichier, un code spécial a été défini (End Of Information code). C'est une des techniques envisageables. En TP, on dispose du dispositif bitstream qui compte les bits afin de détecter la fin de fichier.

Structure des fichiers GIF

Un fichier GIF contient les informations suivantes dans cet ordre :

- le nombre magique : 6 octets lisibles "GIF87a" ou "GIF89a"
- largeur et hauteur de l'écran, chacun sur 2 octets, poids faible d'abord (LSB first) ; cet écran est la zone totale dans laquelle se trouve l'image du fichier, si le fichier ne contient qu'une image, la taille de l'écran = taille de l'image
- un octet dont les bits indiquent les caractéristiques de l'image : bit 7 = palette globale ou pas, bits 2 à 0 : profondeur de la palette-1 (\Rightarrow 2 à 8 bits),
- un octet donnant la couleur de fond d'image dans la palette
- un octet nul permettant de vérifier que le fichier est correct
- la palette : $2^{\text{profondeur}}$ octets RVB.
- une ou plusieurs images à la suite, NB : le format GIF permet de stocker plusieurs images à la suite, toutes partageant la même palette globale, mais chacune pouvant avoir sa palette

Une image contient les informations suivantes :

- un octet valant 0x2C indiquant le début de l'image (nombre magique)
- LeftOffset et TopOffset : deux octets chacun LSB, indiquent l'origine de l'image dans l'écran ; NB : une image peut ne pas remplir la totalité de l'écran
- ImWidth, ImHeight : deux octets chacun LSB, donnent la dimension de l'image. C'est le nombre de pixels à suivre.

- un octet indiquant plusieurs informations booléennes : dont le bit 6 = entrelacée ou pas et bit 7 = présence d'une palette locale avec bit 0 à 2 = profondeur de cette palette.
- Les pixels encodés par LZW, voir le paragraphe suivant.

Exemple de fichier GIF

Voici l'exemple d'un fichier contenant une image gif. od -t x1 im8x8.gif :

od -t x1 im8x8.gif

```
00000000 47 49 46 38 37 61 08 00 08 00 c2 07 00 33 2f c9
00000020 ca a2 4c 42 e1 1d 3e e9 16 16 e9 c4 5a ed 27 e4
00000040 ff 00 ff ff 2c 00 00 00 00 08 00 08 00 00 03
00000060 1b 48 a0 6a b0 d0 c8 01 c6 b9 ae 82 7b ea f8 dc
0000100 27 06 81 58 14 a4 70 ae 81 a0 9e 09 00 3b
```

Voici la liste des codes intéressants (pour celui qui veut creuser les détails...) :

- 47 49 46 38 37 61 : GIF87a
- 08 00 08 00 : dimensions de l'image
- c2 := 11000010 : palette globale, 010 => 3 bits = 8 couleurs
- 07 : le fond est la couleur 7
- 00 : le fichier est correct, fin de l'entête
- 33 2f c9 : couleur 0
- ca a2 4c : couleur 1
- 42 e1 1d : couleur 2
- 3e e9 16 : couleur 3
- 16 e9 c4 : couleur 4
- 5a ed 27 : couleur 5
- e4 ff 00 : couleur 6
- ff ff ff : couleur 7
- 2c : début de l'image
- 00 00 00 00 : offset de l'image dans l'image complète
- 08 00 08 00 : taille de l'image
- 00 : pas de palette spécifique
- 03 1b... : indices des pixels encodés par LZW : 03=> codes sur 3 bits, 1b : 27 codes à suivre
- 00 3b : fin des codes

Algorithme LZW modifié pour GIF

On rappelle que cet algorithme travaille sur des couleurs indexées, c'est-à-dire qu'il reçoit en entrée des octets indiquant des numéros de couleurs dans une palette. Le dictionnaire est initialement rempli avec tous les indices possibles, dico[i] = i pour tout i : 0..255 si l'image est en N=256 couleurs. Le code N+1 est appelé clear code, il a un rôle particulier, et le code N+2 indique la fin des données. En 256 couleurs, les premiers codes à lire font donc 9 bits. La taille du code est indiquée dans le premier octet des données.

Ensuite, le codeur stocke les codes dans des blocs de N: 1..255 octets successifs, la valeur N étant indiqué au début du bloc. La fin des blocs est indiqué par N=0.

A l'intérieur de ces blocs, les codes LZW sont empaquetés dans des octets de la manière suivante : on commence par remplir le poids faible du premier octet, puis son poids fort, puis le poids faible de

l'octet suivant, etc. La documentation donne un exemple avec des codes sur 5 bits : bbbaaaaa, dccccbbb, eeeedddd, ggfffffe, hhhhgggg, ...

Le format PNG

Le format PNG (Portable Network Graphics) est plus moderne que GIF, il offre de nombreuses possibilités : images de toutes sortes (indexées ou non, profondeurs très variées), il compresse sans perte et de manière très efficace.

Concepts

Un fichier PNG commence par une signature (nombre magique) puis contient une série de paquets appelés chunks.

Le nombre magique contient 8 octets, leurs valeurs hexadécimales : 89 50 4e 47 0d 0a 1a 0a. Ces codes ont été choisis pour minimiser les problèmes de mauvaise copie d'un système à l'autre (ex : un transfert ftp en mode texte par erreur dégraderait entre autres les retour-chariot présents dans la signature, donc on pourrait se rendre compte du problème).

Un paquet est constitué de 4 parties :

Longueur/4

On trouve 4 octets MSB-LSB pour donner la taille des données du paquet sans compter l'entête du paquet ni le CRC.

Identifiant/4

4 octets indiquent le type de paquet. Ce sont toujours des octets lisibles en ascii. Les lettres majuscules ou minuscules signalent des propriétés particulières.

Données/L

L = longueur du paquet octets de données

CRC/4

Un code de redondance cyclique pour vérifier l'intégrité du paquet.

Certains paquets sont obligatoires, on doit les trouver dans toute image png :

IHDR

Ce doit être le premier paquet d'un fichier, ses données contiennent les informations sur l'image : Largeur/4, Hauteur/4, profondeur par composante/1, type de couleur/1 (1:palette, 2:couleur, 4: canal alpha), compression/1, filtrage/1, entrelacement/1

PLTE

Ce paquet contient 1 à 256 triplets RGB/3 définissant les couleurs d'une image couleurs indexées (type de couleur = 3)

IDAT

Ces octets contiennent l'image compressée. L'image peut être codée sur plusieurs paquets IDAT.

IEND

Ce paquet indique la fin du fichier, il doit être le dernier. Il ne contient pas de données.

Certains paquets sont optionnels :

bKGD

Ce paquet spécifie la couleur de fond.

tRNS

Indique la ou les couleurs transparentes de l'image

gAMA

Donne la correction Gamma de l'image (voir S4P4). Elle est codée sous forme d'un produit par 100000.

tEXt

Description de l'image ou de l'auteur de l'image, par mot-clés et lignes de texte.

Exemple de fichier png

Voici un viddage d'un fichier png :

od -t ax1 im8x8.png

```
00000000 ht P N G cr nl sub nl nul nul nul cr I H D R
  89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52
00000020 nul nul nul bs nul nul nul bs eot etx nul nul nul 6 ! #
  00 00 00 08 00 00 00 08 04 03 00 00 00 00 36 21 a3
00000040 8 nul nul nul soh s R G B nul . N fs i nul nul
  b8 00 00 00 01 73 52 47 42 00 ae ce 1c e9 00 00
00000060 nul can P L T E 3 / I J " L B a gs >
  00 18 50 4c 54 45 33 2f c9 ca a2 4c 42 e1 1d 3e
00000100 i syn syn i D Z m ' d del nul del del del 7 _
  e9 16 16 e9 c4 5a ed 27 e4 ff 00 ff ff ff 37 5f
00000120 Q , nul nul nul 0 I D A T bs W c p p p
  d1 2c 00 00 00 30 49 44 41 54 08 d7 63 70 70 70
00000140 H ` a a K c 0 0 / O ` f / g
  48 60 60 61 61 4b 63 30 30 2f 4f 60 60 66 2f 67
00000160 f 0 6 / 7 f 0 6 dc4 6 f bs nak T cr e
  66 30 36 2f 37 66 30 36 14 36 66 08 15 54 0d 65
00000200 bs nak R cr enq nul ht : ack V etb 7 p u nul nul
  08 15 52 0d 05 00 89 3a 06 d6 17 b7 d0 f5 00 00
00000220 nul nul I E N D . B ` stx
  00 00 49 45 4e 44 ae 42 60 82
```

L'affichage des caractères ascii en face des codes permet de décoder plus facilement. On peut ainsi reconnaître les éléments suivants :

- 89 50 4e 47 0d 0a 1a 0a : l'entête PNG
- 00 00 00 0d = 13 : taille du premier chunk
- 49 48 44 52 : IHDR : type du chunk
- 00 00 00 08 00 00 00 08 : dimensions de l'image
- 04 : profondeur de l'image
- 03 : type d'image : en couleur avec une palette
- 00 : type de compression : deflate voir plus bas
- 00 : filtrage
- 00 : pas d'entrelacement
- 36 21 a3 b8 : CRC de ce chunk. On a bien 13 octets de données après le nom du chunk et avant le CRC.
- 00 00 00 01, 73 52 47 42 : 1 octet de données pour un chunk sRGB
- 00 : donnée du chunk sRGB
- ae ce 1c e9 : CRC du chunk sRGB
- 00 00 00 18, 50 4c 54 45 : 24 octets pour le chunk PLTE : 8 triplets (R,G,B)
- 33 2f c9 : couleur 0
- ca a2 4c : couleur 1

- 42 e1 1d : couleur 2
- 3e e9 16 : couleur 3
- 16 e9 c4 : couleur 4
- 5a ed 27 : couleur 5
- e4 ff 00 : couleur 6
- ff ff ff : couleur 7
- 37 5f d1 2c : CRC de la palette
- 00 00 00 30, 49 44 41 54 = 48 octets de données dans un chunk IDAT
- 08 d7... : données compressées
- le fichier se termine par un chunk iEND vide

Compression

PNG utilise l'algorithme Deflate proposé par Phil Katz en 1987, utilisé pour pkzip et gzip. Il est basé sur LZ77 et Huffman. Il code les données par ensembles de 64Ko indépendants. L'algorithme LZ77 travaille avec une fenêtre de 32Ko. Les couples <longueur, position> sont encodés à l'aide de Huffman.

Commentaires sur ce format

Moderne, polyvalent, amélioration de GIF.

Semaine 15 (02/01/2012) : Etude du format JPG

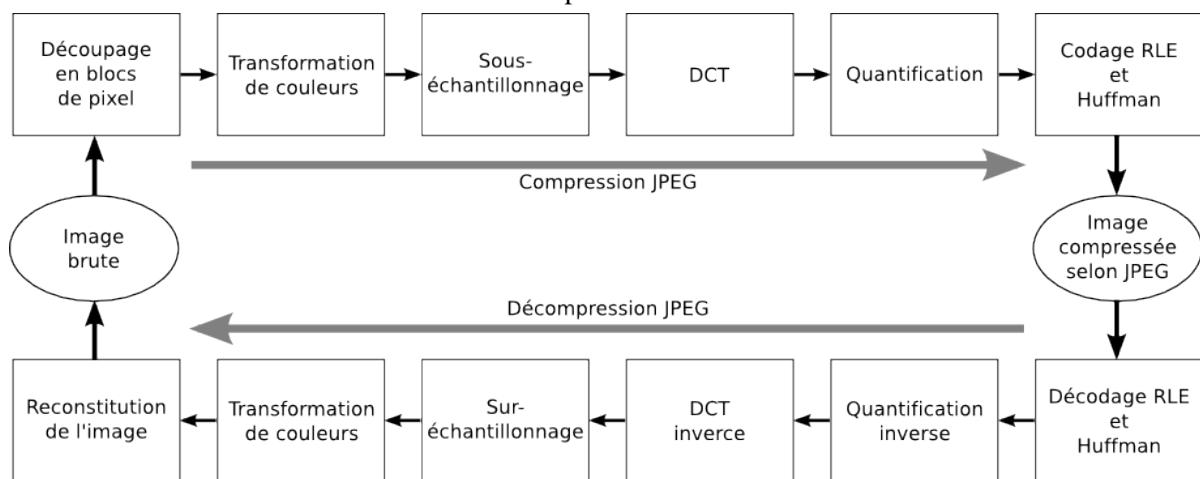
Concepts de la compression JPEG

Cette norme a été mise au point à la fin des années 1980 par un ensemble d'organismes nationaux de normalisation et des industriels : le Joint Photographic Expert Group (JPEG). Le format est donc relativement indépendant des éditeurs de logiciels.

La compression JPG est appropriée pour les images photographiques. Elle n'est pas aussi performante que gif ou png sur des images bitmap. Là où un aplat est compressé en quelques octets en png, il en faut toujours des dizaines avec jpeg. Par contre, là où les teintes varient en permanence, jpeg compresse toujours avec le même ratio tandis que png et gif sont à la rue.

La compression s'effectue en plusieurs traitements séquentiels. Le coeur est une transformée en cosinus (DCT) qui permet d'extraire des corrélations spatiales sur l'image, c'est à dire que certains motifs et formes géométriques caractéristiques sont codées de manière économique. Par exemple, un bord vertical = transition clair/sombre pour un grand nombre de pixels est codé avec un seul nombre.

Voici une représentation des traitements. On commence avec une image à gauche. La compression se déroule avec les traitements du haut et la décompression se fait avec les blocs du bas.



1. Division de l'image en blocs de 8x8 pixels appelés macroblocs
2. Séparation de chaque bloc en plans Y (luminance), Cr et Cb (chrominance : rouge et bleu), ces deux plans étant sous-échantillonnés d'un rapport 2
3. Transformation discrète en cosinus (DCT) sur chaque bloc : on obtient 8x8 coefficients de Fourier
4. Quantification des coefficients = mise à zéro des plus petits en valeur absolue et/ou des moins significatifs
5. Compression des coefficients restants : parcours en zigzag, rle et codage huffman

La décompression fait les mêmes étapes mais dans l'ordre inverse.

Le gain de place, c'est à dire la compression résulte des étapes 2 et 4 : c'est le fait de ne pas coder tous les coefficients de Fourier et de ne pas représenter toutes les informations de couleur qui permet d'obtenir une image compressée. Evidemment, la décompression ne peut pas restaurer les coefficients et couleurs perdues, donc l'image décompressée n'est pas identique à l'image source.

Nous allons étudier les points intéressants de chacune de ces étapes.

Séparation de l'image en macroblocs

L'image d'origine est découpée en blocs de 8x8 pixels. Chacun est traité indépendamment. La raison de ce découpage est la difficulté de mener les traitements suivants sur des blocs plus grands : le temps de calcul et le nombre de variables augmentent considérablement. La valeur 8x8 a été choisie car les temps de calculs sont raisonnables et les algorithmes peuvent être implantés sur des circuits intégrés sans recourir à trop de transistors.

Le découpage en macroblocs indépendants cause souvent des artefacts aux bords de ces blocs car la compression varie d'un bloc à son voisin. En effet, la compression consiste à supprimer certaines informations, mais pas les mêmes d'un macrobloc à l'autre. Donc, la décompression peut montrer des petites différences sur les frontières des macroblocs.

Voici une illustration du phénomène :



Cette image, issue de la wikipédia sur les fleurs, a été réenregistrée avec un très fort taux de compression. La figure montre un agrandissement de l'image obtenue. On peut noter plusieurs phénomènes :

- On voit nettement les macroblocs 8x8
 - On voit les effets de la transformation en cosinus : les trames verticales ou horizontales résultent des coefficients les plus grands (voir plus bas) et des choix du compresseur de ne pas garder les petits coefficients.
 - Les blocs ne sont pas compressés de la même manière : ce ne sont pas les mêmes coefficients qui sont gardés d'un bloc à l'autre.
-

Séparation YCrCb et sous-échantillonnage

Le cours [S4P4 - Outils et matériels pour l'IN](#) (S4P4 - Outils IN - plan.html) montrera les limitations de la vision humaine, en particulier son manque de sensibilité couleurs. La représentation RVB n'est pas la plus adaptée pour exploiter ces "défaux". On lui préfère le mode YUV ou YCrCb. Y est nommé *luminance*, c'est pour simplifier la luminosité du pixel, et les deux valeurs Cr et Cb sont nommées *chrominance*, ils spécifient la coloration du pixel.

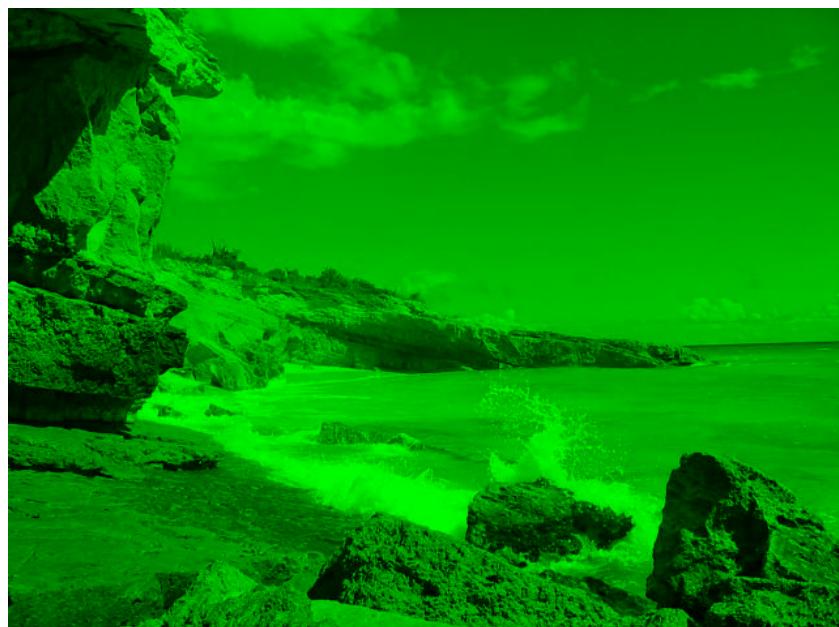
Voici les formules de conversion entre les systèmes RVB et YCrCb. A noter qu'il existe une multitude de normes (YUV, YPrPb...) correspondant à des standards du marché et aux possibilités techniques des dispositifs.

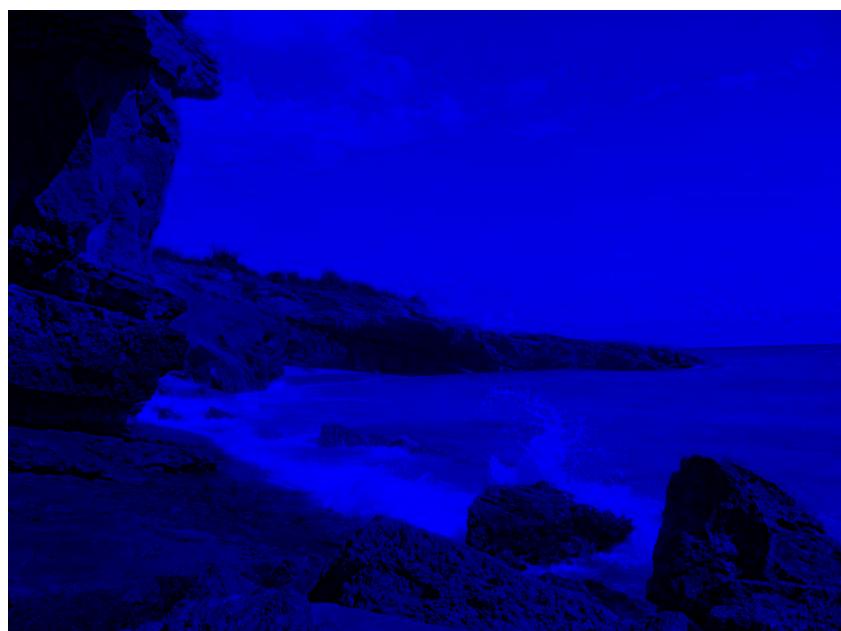
$$\begin{cases} Y = 0.257R + 0.504G + 0.098B + 16 \\ Cr = 0.439R - 0.368G - 0.071B + 128 \\ Cb = -0.148R - 0.291G + 0.439B + 128 \end{cases} \text{ et } \begin{cases} R = 1.164(Y-16) + 1.596(Cr-128) \\ G = 1.164(Y-16) - 0.813(Cr-128) - 0.391(Cb-128) \\ B = 1.164(Y-16) + 2.018(Cb-128) \end{cases}$$

Avec ce système de représentation des couleurs, on s'aperçoit qu'il est inutile de coder la chrominance pour chaque pixel car l'oeil ne la distingue pas bien : on code la luminance de chaque pixel mais seulement la chrominance moyenne de carrés de 2x2 pixels. Ainsi, il y a 4 fois moins d'informations à compresser pour la couleur.

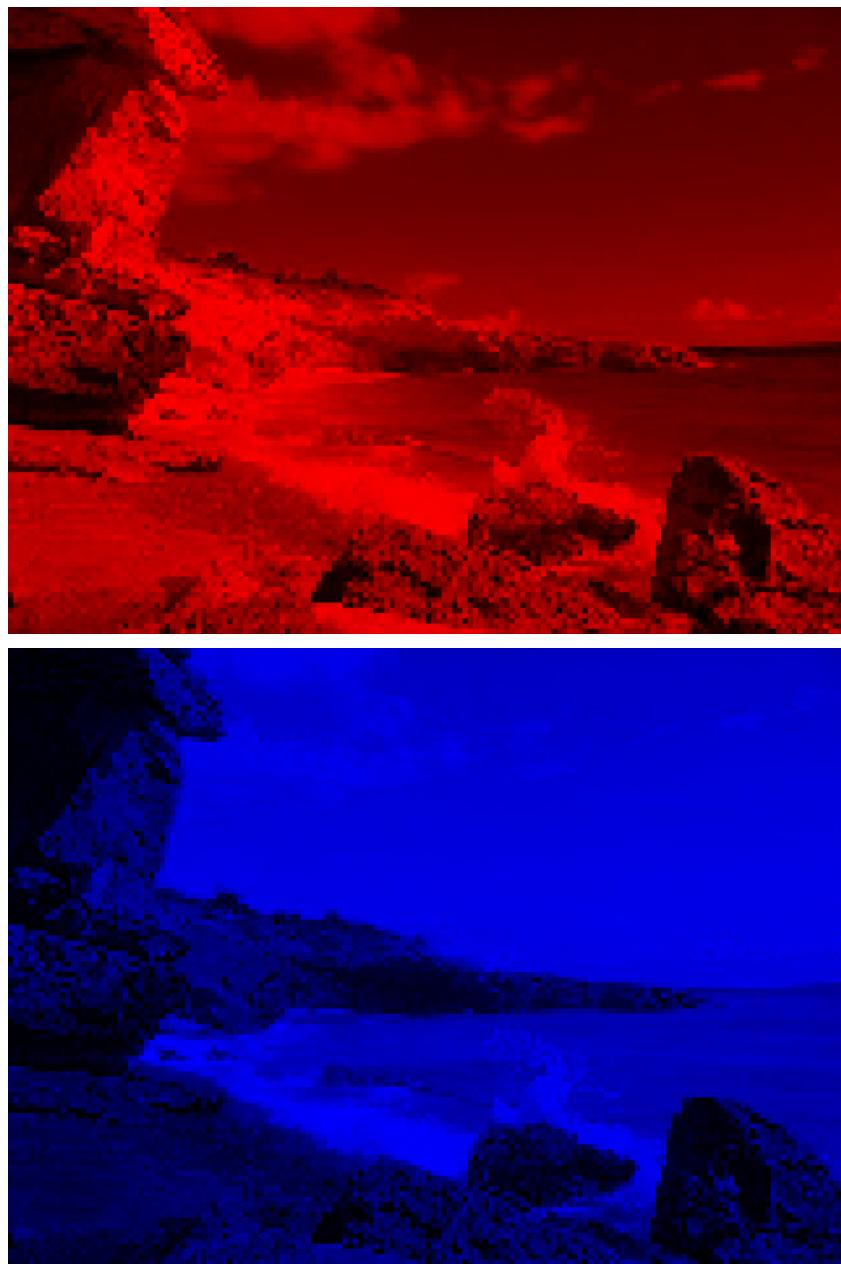
Démonstration sur le [tutoriel n°1](#) (documents/tutoriel1 - linux.zip). Ouvrir une image quelconque et regarder ses plans couleur. On s'aperçoit que les plans rouge et bleu portent assez peu d'information : il y a peu de nuances et s'il y en avait, l'oeil ne les distinguerait pas. On constate aussi, avec les images jpeg, que les pixels rouges et bleus sont empatés : ils ont été groupés 4 par 4 et codés ensemble. C'est le mélange des couleurs avec le vert qui trompe l'oeil.

Voici une illustration de ce phénomène. On part d'une image couleur qu'on sépare en plans R, V, B (et non pas Y, Cr et Cb), pour simplifier.





Ensuite, on fait descendre la résolution des couches R et B d'un facteur 4 (au lieu de 2 pour le jpeg) :



Puis on réunit à nouveau les couches (mode écran dans TheGimp avec un léger décalage des couches vers le haut et la gauche pour centrer les pixels). Voici le résultat à gauche, à côté de l'image d'origine à droite ou dessous.



On voit nettement quelques artefacts colorés, mais finalement, pas tant que ça : l'image garde ses couleurs et à peu près sa forme, pourtant on travaille déjà avec 3/4 des informations de couleurs en moins !

Dans la compression jpeg, on utilise le modèle YCrCb dans lequel les informations de couleur collent le mieux avec les capacités de la vision humaine. Voici une image résultant d'un sous-échantillonnage de rapport **x16** (!) des plans Cr et Cb. On voit clairement des artefacts, mais le résultat est très surprenant de qualité malgré ce traitement violent ; on s'attendrait à une image affreuse. C'est la vision humaine qui distingue extrêmement mal les nuances de couleurs.



Transformation DCT

L'étape suivante consiste à effectuer une transformée DCT sur les 8x8 pixels du bloc. La transformation fait passer d'une image NxN pixels à un tableau NxN de nombres réels qui ont des propriétés très intéressantes. Cette transformation est définie par l'équation suivante :

$$DCT(i; j) = \frac{2}{N} C(i) C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \cos\left(\frac{(2x+1)\pi i}{2N}\right) \cos\left(\frac{(2y+1)\pi j}{2N}\right) \text{pixel}(x; y) \quad \text{avec } C(0) = \frac{1}{\sqrt{2}} \text{ et } C(>0) = 1$$

i et j sont les indices dans la transformée, x et y sont les indices dans l'image de départ (attention, le i n'est pas le nombre complexe). Pour calculer la transformée d'une image, on fait varier i et j entre 0 et N-1 et on calcule la formule qui elle-même fait varier x et y pour parcourir l'image.

Cette équation d'apparence compliquée peut se comprendre ainsi :

$$DCT(i; j) = \frac{2}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} k(x; i) k(y; j) \text{pixel}(x; y) \quad \text{avec } k(xy; ij) = C(ij) \cos\left(\frac{(2xy+1)\pi ij}{2N}\right),$$

rappel : $C(0) = \frac{1}{\sqrt{2}}$ et $C(>0) = 1$

Avec cette écriture, le calcul consiste simplement en une combinaison linéaire des pixels avec des coefficients : pour un coefficient DCT(i,j) à calculer, on multiplie chaque pixel (x,y) du bloc par une valeur $k(x)*k(y)$ dépendant de la position du pixel et de i et j, et on fait la somme du tout ; on répète cela pour chaque position (i,j) de la transformée à calculer.

Par exemple, soit une DCT 4x4 à faire. On doit calculer DCT(0,0), DCT(1,0)... DCT(2,3), DCT(3,3). Par exemple,

$$DCT(2; 1) = \frac{2}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} k(x; 2) k(y; 1) \text{pixel}(x; y)$$

$$\text{avec } k(x; 2) = C(2) \cos\left(\frac{(2x+1)\pi 2}{2N}\right) = \cos\left(\frac{(2x+1)\pi}{4}\right) \text{ et } k(y; 1) = C(1) \cos\left(\frac{(2y+1)\pi 1}{2N}\right) = \cos\left(\frac{(2y+1)\pi}{8}\right)$$

Ca donne les calculs suivants, partiellement factorisés pour simplifier :

$$ligne_0 = \cos\left(\frac{\pi}{8}\right)\left(\cos\left(\frac{\pi}{4}\right)\text{pixel}(0;0) + \cos\left(\frac{3\pi}{4}\right)\text{pixel}(1;0) + \cos\left(\frac{5\pi}{4}\right)\text{pixel}(2;0) + \cos\left(\frac{7\pi}{4}\right)\text{pixel}(3;0)\right)$$

$$ligne_1 = \cos\left(\frac{3\pi}{8}\right)\left(\cos\left(\frac{\pi}{4}\right)\text{pixel}(0;1) + \cos\left(\frac{3\pi}{4}\right)\text{pixel}(1;1) + \cos\left(\frac{5\pi}{4}\right)\text{pixel}(2;1) + \cos\left(\frac{7\pi}{4}\right)\text{pixel}(3;1)\right)$$

$$ligne_2 = \cos\left(\frac{5\pi}{8}\right)\left(\cos\left(\frac{\pi}{4}\right)\text{pixel}(0;2) + \cos\left(\frac{3\pi}{4}\right)\text{pixel}(1;2) + \cos\left(\frac{5\pi}{4}\right)\text{pixel}(2;2) + \cos\left(\frac{7\pi}{4}\right)\text{pixel}(3;2)\right)$$

$$ligne_3 = \cos\left(\frac{7\pi}{8}\right)\left(\cos\left(\frac{\pi}{4}\right)\text{pixel}(0;3) + \cos\left(\frac{3\pi}{4}\right)\text{pixel}(1;3) + \cos\left(\frac{5\pi}{4}\right)\text{pixel}(2;3) + \cos\left(\frac{7\pi}{4}\right)\text{pixel}(3;3)\right)$$

$$\text{Pour finir, } \text{DCT}(2;1) = \frac{2}{N}(ligne_0 + ligne_1 + ligne_2 + ligne_3)$$

Les calculs peuvent paraître impressionnantes mais il existe une simplification algorithmique tout aussi impressionnante : le calcul sur ordinateur n'est pas en $O(n^2)$ mais en $O(n \cdot \log_2(n))$. C'est à dire qu'il suffit de faire 8x8 calculs de cosinus par exemple, on en fait seulement $8 \cdot 3 \dots$ Ceci grâce aux propriétés de la fonction cosinus qui permettent de trouver des factorisations. Pour autant, il ne faut pas en profiter pour faire des DCT à tout bout de champ, ça reste un calcul lourd.

Les coefficients k correspondent à un ensemble de fonctions orthogonales (c'est à dire : indépendantes entre elles mais "couvrant" tout l'espace) qui donnent le nom à la transformée (cosinus). On peut noter que la fonction k s'écrit aussi $k(x,i) = C(i) \cos\left(\left(x + \frac{1}{2}\right)i \frac{\pi}{N}\right)$ qui fait apparaître plus clairement le cosinus sur plusieurs (i) fractions de tour (π/N) avec une sorte de déphasage d'une fraction de tour en x .

Illustration du calcul de la DCT

Le calcul de la DCT consiste comme on vient de le voir à multiplier les pixels $\text{pixel}(x,y)$ par des nombres $k_{ij}(x,y)$ valant $k(x,i)k(y,j)$ et à faire la somme de tous ces produits. Ces nombres $k_{ij}(x,y)$ varient selon le coefficient $\text{DCT}(i,j)$ qu'on veut calculer. Par exemple, quand on veut calculer $\text{DCT}(0,0)$, on fait la somme

des produits : $\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} k_{0;0}(x,y) \text{pixel}(x,y)$, quand on veut calculer $\text{DCT}(2,3)$, on calcule cette somme :

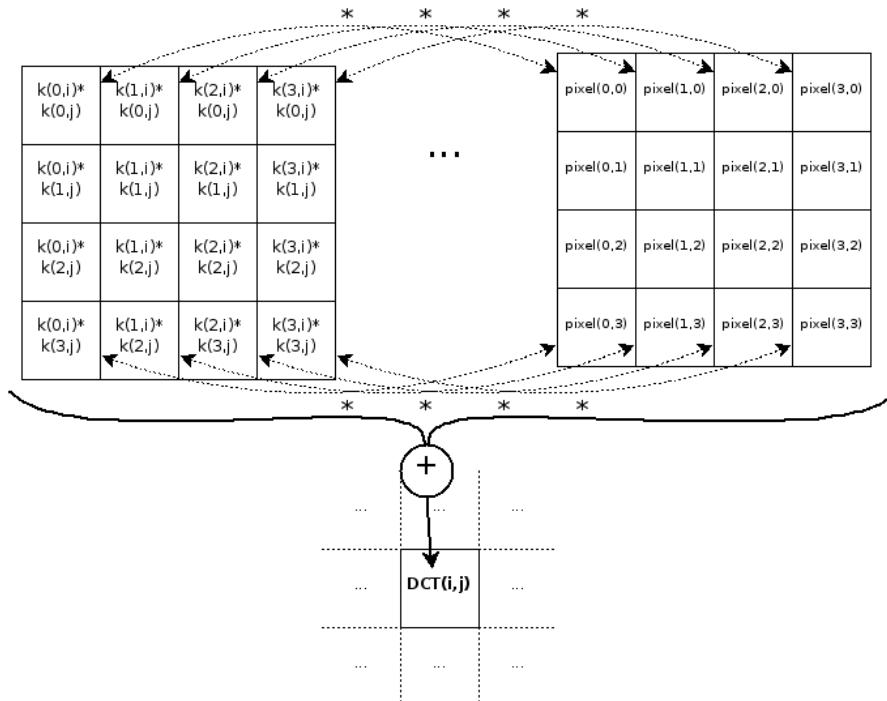
$$\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} k_{2;3}(x,y) \text{pixel}(x,y)$$

et ainsi de suite.

Et on a vu que $k_{ij}(x,y) = C(i) \cos\left(\frac{(2y+1)\pi i}{2N}\right) C(j) \cos\left(\frac{(2x+1)\pi j}{2N}\right)$, les nombres $C(i)$ et $C(j)$ valent 1 sauf en 0

où ils valent $\frac{1}{\sqrt{2}}$.

C'est comme une sorte de produit entre deux images : on multiplie les pixels de l'une par ceux de l'autre et on fait la somme des valeurs obtenues. Pour obtenir tous les coefficients $\text{DCT}(i,j)$, on doit refaire tous ces calculs pour chaque couple (i,j) , parce que les valeurs $k(x,i)*k(y,j)$ changent selon i et j .



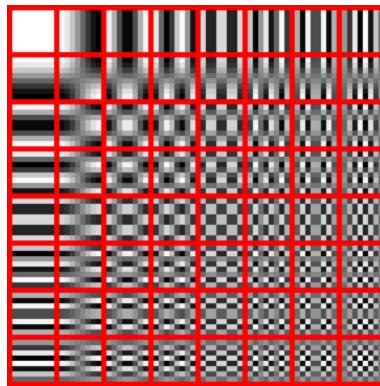
Par exemple, voici pour $i=1, j=2$ en dimension 4×4 les valeurs $k(x,1)*k(y,2)$ (les colonnes sont les x , les lignes sont les y) :

	0	1	2	3
0	0,65	0,27	-0,27	-0,65
1	-0,65	-0,27	0,27	0,65
2	-0,65	-0,27	0,27	0,65
3	0,65	0,27	-0,27	-0,65

Voici les valeurs pour $i=3, j=1$:

	0	1	2	3
0	0,35	-0,85	0,85	-0,35
1	0,15	-0,35	0,35	-0,15
2	-0,15	0,35	-0,35	0,15
3	-0,35	0,85	-0,85	0,35

Voici une image qui représente les coefficients $k(.,.,i,j)$ pour chaque i,j entre 0 et 7 (dimension $N=8$). Chaque carré rouge est l'un des $k(.,.,i,j)$ et à l'intérieur d'un carré, on fait varier x et y pour dessiner les nuances de gris. Ainsi, $k(0,0)$ est constant (on voit bien que si $i=j=0 \Rightarrow k(0,0,x,y) = C(0)*C(0) = 1/2$. et $k(7,7)$ est un damier.



Ce [fichier tableur OpenOffice](#) (documents/CalculsDCT.ods) permet de calculer les valeurs K pour N=4 et 8. Il faut changer les cellules D2 et F2 qui contiennent les valeurs i et j. Le diagramme de droite est une tentative pour afficher les valeurs K, mais il n'est pas toujours très lisible. NB : il faut activer les macros, il y en a pour calculer les valeurs K.

Transformée inverse iDCT

La transformation inverse notée IDCT est le miroir de la DCT :

$$\text{pixel}(x; y) = \frac{2}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i) C(j) \cos\left(\frac{(2x+1)\pi i}{2N}\right) \cos\left(\frac{(2y+1)\pi j}{2N}\right) DCT(i; j)$$

On peut aussi la rendre plus claire :

$$\text{pixel}(x; y) = \frac{2}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} k(x; i) k(y; j) DCT(i; j)$$

C'est i et j qu'on fait varier dans la somme pour calculer le pixel (x,y). Mais attention à ne pas se laisser tromper par la très grande ressemblance des formules. Rappel :

$$DCT(i; j) = \frac{2}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} k(x; i) k(y; j) \text{pixel}(x; y)$$

Bien que les formules soient syntaxiquement très proches, il faut bien comprendre que les sommes font varier i et j dans la transformée inverse et x et y dans la transformée directe, or les termes k(x,i) et k(y,j) ne sont pas du tout symétriques en x et en i, ou en y et en j. Donc ce n'est pas du tout le même calcul qui est mené, on ne peut pas utiliser l'algorithme de la transformée directe pour calculer la transformée inverse en lui fournissant un spectre à la place d'une image, ça ne marcherait pas.

Ce [fichier tableur OpenOffice](#) (documents/DCT4.ods) détaille les calculs de la transformée directe et inverse. Il faut juste placer des valeurs dans les zones saumon, le résultat apparaît dans les zones vertes (faire un collage spécial des nombres uniquement si on veut faire la transformée inverse de la transformée directe).

Intérêt de la DCT pour la compression d'images

L'intérêt de la DCT est au moins triple pour la compression d'images :

- pas de nombres complexes, uniquement des valeurs réelles.
- un seul ensemble de valeurs (pas de symétrie autour d'un point central) et autant de coefficients que de pixels transformés.
- de très bonnes propriétés subjectives pour la représentation des fréquences spatiales.

Elle s'exécute très rapidement sur les ordinateurs, par un algorithme en O(n.log2(n)) : DCT(8x8) en 64*6 ensembles d'opérations. En fait, on ne fait pas les quatre boucles imbriquées dont la double somme car il y a de nombreuses symétries périodiques grâce au cosinus, on peut factoriser des calculs.

La transformée en nombre réels est parfaitement inversible, par contre si on travaille avec des nombres entiers (réels arrondis), l'inversion ne restitue pas tout à fait l'original.

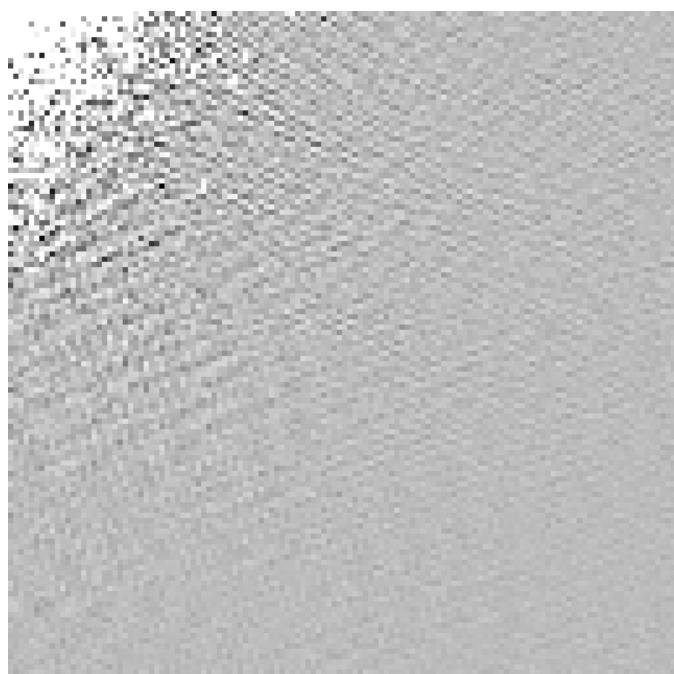
Spectre DCT

La DCT extrait le *spectre d'amplitude* d'un signal 2D : c'est à dire qu'elle représente une image comme une superposition de motifs de fréquences différentes. Les valeurs du spectre sont appelées *coefficients dct*. Bien se souvenir qu'on travaille avec les plans Y, Cr et Cb séparément, donc ce sont finalement des images en niveaux de gris qu'on traite.

Exemple de spectre (le gris moyen représente un coefficient nul, le gris sombre est un coefficient négatif et le gris clair un coefficient positif) :



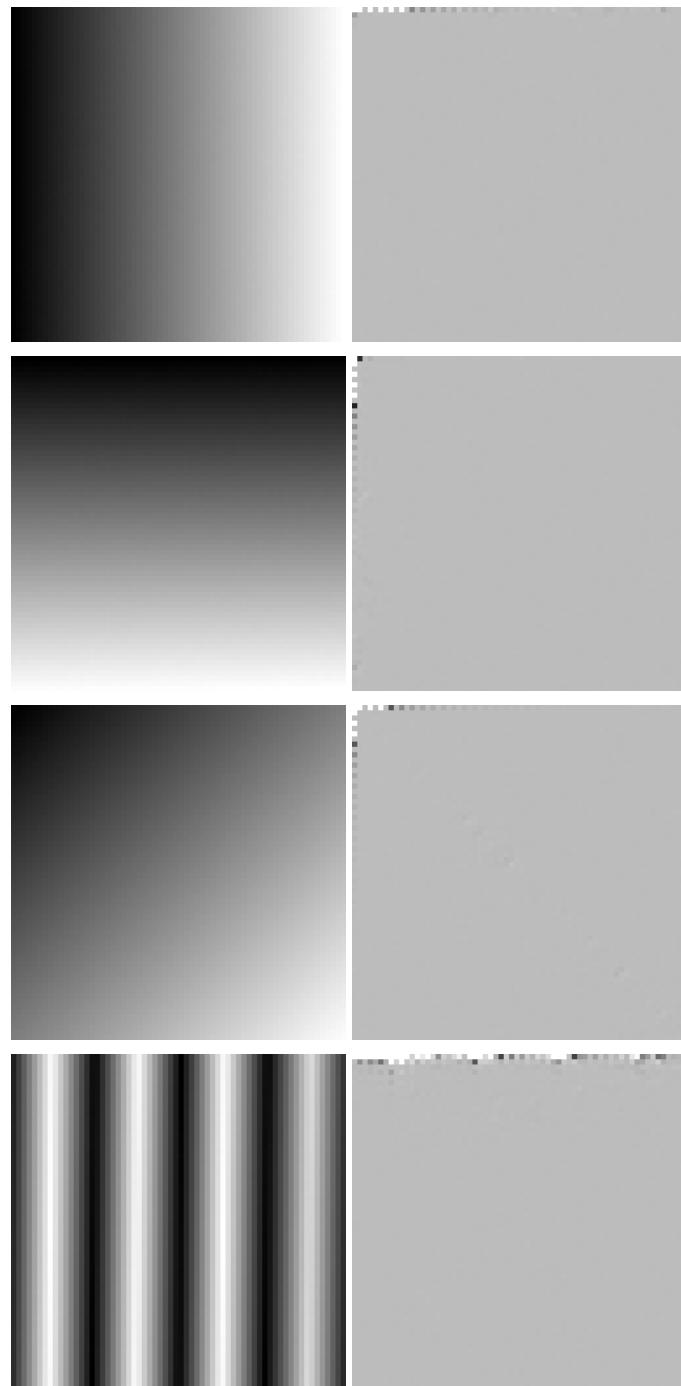
image d'origine

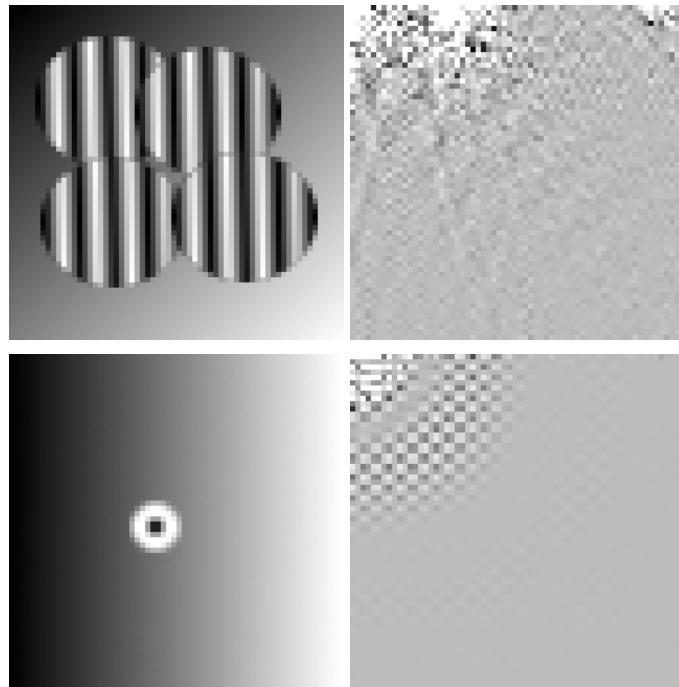


Spectre de l'image précédente

Les spectres DCT sont toujours similaires à celui-ci : le coefficient situé en haut et à gauche représente la valeur moyenne des pixels de l'image initiale. Ensuite, autour de ce coin, on trouve les coefficients de basse fréquence spatiale, c'est à dire l'importance des ondulations de grande taille dans l'image initiale. Quand l'image contient de nombreux petits détails très marqués, on trouve plutôt des coefficients élevés dans les zones droite et basses, selon que c'est en x ou en y ou les deux à la fois.

Pour bien comprendre, voici une série d'images avec leur spectre (calculé pour N=64).





On voit donc que les détails sont représentés par des valeurs non nulles situées vers le bas et la droite, tandis que les grandes ondulations sont représentées en haut et à gauche. Bien noter qu'il n'y a jamais un seul coefficient non nul isolé, on trouve toujours des sortes d'échos autour, voir le cours de maths pour comprendre exactement pourquoi.

On peut aussi apercevoir des petites variations dans les coefficients nuls. Elles sont dûes à une imprécision de représentation des pixels : les dégradés sont faits avec des niveaux de gris pas suffisamment précis, donc les coefficients ne sont pas exactement nuls.

Le résultat de la transformation DCT d'une image NxN est un tableau [ligne:0..N-1, colonne:0..N-1] de réels. Ce tableau contient des valeurs plus ou moins grandes ; en général, la case [0,0] contient la valeur la plus grande en absolu qu'on peut assimiler à l'intensité moyenne des pixels du bloc, et les cases [i,j] contiennent des valeurs d'autant plus petites que i et j sont grands. On se base sur cette propriété pour la compression : tous ces coefficients ne sont pas codés.

Bilan

Voici un exemple extrait du [Wikipedia JPEG](http://en.wikipedia.org/wiki/JPEG) (<http://en.wikipedia.org/wiki/JPEG>).

Soit le macrobloc suivant :



Voici les valeurs des pixels :

52	55	61	66	70	61	64	73
63	59	55	90	109	85	69	72
62	59	68	113	144	104	66	73
63	58	71	122	154	106	70	69
67	61	68	104	126	88	68	70
79	65	60	70	77	68	58	75
85	71	64	59	55	61	65	83
87	79	69	68	65	76	78	94

On leur soustrait 128 pour les placer entre -128 et +127 :

-76	-73	-67	-62	-58	-67	-64	-55
-65	-69	-73	-38	-19	-43	-59	-56
-66	-69	-60	-15	16	-24	-62	-55
-65	-70	-57	-6	26	-22	-58	-59
-61	-67	-60	-24	-2	-40	-60	-58
-49	-63	-68	-58	-51	-60	-70	-53
-43	-57	-64	-69	-73	-67	-63	-45
-41	-49	-59	-60	-63	-52	-50	-34

On effectue la transformation DCT et on obtient les coefficients entiers suivants :

-415	-30	-61	27	56	-20	-2	0
4	-22	-61	10	13	-7	-9	5
-47	7	77	-25	-29	10	5	-6
-49	12	34	-15	-10	6	2	2
12	-7	-13	-4	-2	2	-3	3
-8	3	2	-6	-2	1	4	2
-1	0	0	-2	-1	-3	4	-1
0	0	-1	-4	-1	0	1	2

Par exemple, le -415 est la valeur moyenne des pixels, pondérée par les coefficients de normalisation.

Quantification des coefficients

Le codage du tableau des coefficients (NxN réels) occupe plus de place que les pixels d'origine (NxN octets), donc si on tentait de le coder tel quel, on perdrait de la place par rapport à l'image d'origine.

On réduit la taille du fichier résultant en "supprimant" des coefficients, c'est à dire en mettant des valeurs nulles à la place. Evidemment, dans ce cas, la compression est destructrice d'information : on ne pourra pas récupérer l'image d'origine.

Le but est de ne garder que les coefficients qui contribuent réellement à la qualité d'image : ce sont les plus grandes valeurs absolues dans le spectre. On doit éliminer (mettre à zéro) les coefficients parmi les plus faibles. On peut également choisir d'accentuer certains coefficients afin de réhausser le contraste.

Démonstration sur [ce tutoriel à télécharger et à extraire](#) (documents/tutoriel2.zip). Un applet java aurait été trop lent.

Il existe plusieurs méthodes pour choisir les coefficients à garder :

- par zone : on garde seulement les coefficients situés en haut et à gauche.
- par seuil : on garde uniquement les coefficients plus grands qu'une valeurs fixée à priori.
- par nombre : on garde un certain nombre de coefficients parmi les plus grands.
- par matrice : on multiplie les coefficients par une matrice qui affaiblit certains et annule les plus faibles.

Quelle que soit la méthode, le principe est de mettre plusieurs des cases du tableau des coefficients à zéro afin de pouvoir les coder efficacement (matrice creuse).

Quantification par zonage

Ca consiste à choisir arbitrairement la partie de la matrice qu'on garde : par exemple $n \times n$ ($n < N$) coefficients situés à gauche et en haut ; les autres ($i > n$ ou $j > n$) sont mis à zéro. L'avantage est de pouvoir prévoir le taux de compression à l'avance : $n \times n$ réels au lieu de $N \times N$ octets qui sont faciles à coder car on sait où sont les zéros. L'inconvénient est de supprimer arbitrairement des coefficients qui pourraient être significatifs dans certains cas.

Quantification par seuil

Ca consiste à mettre à zéro tous les coefficients qui dépassent en valeur absolue un seuil fixé pour chaque bloc. L'inconvénient est de ne pouvoir prévoir la compression, on ne sait pas combien de valeurs resteront dans chaque bloc transformé.

Quantification par nombre

Une variante de la précédente consiste à mettre à zéro m coefficients, $0 < m \leq N \times N$ parmi les plus petits (tri des valeurs dans l'ordre décroissant). C'est ce qui est fait dans le tutoriel 2. On peut prévoir le taux de compression, mais pas l'impact sur la qualité : en effet, même si le spectre contient des coefficients élevés, s'il y en a beaucoup, certains seront éliminés.

Quantification par matrice

La technique employée dans la compression jpg consiste à diviser en nombres entiers les coefficients DCT par ceux d'une matrice prédéfinie. En général, cette matrice a de petites valeurs dans les cases de bas indices (haut-gauche). La division est entière, elle entraîne des valeurs nulles dans le résultat.

Chaque compresseur jpeg possède une ou plusieurs matrices de quantification prédéfinies. Il les insère au début du fichier. Par exemple, une matrice peut être comme celle-ci :

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

Le résultat de la division entière (arrondi au plus proche entier) case à case de cette matrice par les coefficients DCT obtenus précédemment :

$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix}$$

donne ceci :

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Certains dispositifs ont des matrices figées (ex: appareil photo en mode super, fine, normale...). Les coefficients de ces matrices prédéfinies effectuent une coupe plus ou moins sévère sur certaines parties du spectre.

Il existe d'autres techniques permettant de faire varier la qualité de l'image compressée, par exemple, de définir les coefficients de quantification par cette formule $Q(i,j) = 1 + (1 + i + j) * F_q$

Le décompresseur essaie de restituer les coefficients initiaux en effectuant la multiplication des valeurs par la matrice, mais le résultat n'est pas exact. Par exemple $-26 * 16 = -416$ (au lieu de -415), $-3 * 11 = -33$ (au lieu de -30)... C'est une autre source d'imprécision de l'image décompressée.

Encodage des coefficients DCT

A ce stade, on doit enregistrer différents tableaux $N \times N$ de réels dont beaucoup sont nuls. On appelle matrices creuses ce genre de tableaux. La méthode RLE est indiquée pour coder les nombreux zéros, mais le sens de parcours des cases a son importance afin de maximiser les séquences identiques.

Par exemple, soit à encoder ce spectre dans le fichier .jpg :

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Ces nombres ont des valeurs absolues importantes dans le coin haut-gauche, en particulier, le nombre [0,0] est la plus grande valeur, il représente grossièrement la moyenne des pixels du bloc d'origine. Ensuite les valeurs des cases [i,j] décroissent à mesure que la somme $i+j$ augmente.

C'est pour cette raison qu'on effectue un parcours en zigzag (voir TP5). De cette manière, passés les premiers coefficients, on trouve de plus en plus de zéros

Le codeur RLE de Jpeg représente une séquence finale composée uniquement de zéros par un code spécial nommé EOB : #26, #3, 0, #3, #2, #6, 2, #4, 1, #4, 1, 1, 5, 1, 2, #1, 1, #1, 2, 0, 0, 0, 0, 0, #1, #1, EOB.

Codage des coefficients DC0 des différents blocs

En fait, le premier coefficient de la série n'est pas représenté avec les autres en RLE. Il fait l'objet d'un traitement spécial. Ce coefficient est appelé DC0 (composante continue, comme dans le sigle AC/DC) et les suivants sont appelés AC0..AC63. DC0 est la valeur moyenne des pixels du macrobloc.

- Si on compose une image uniquement avec ces coefficients, on obtient une sorte d'imagette de l'image complète. Ces coefficients représentent l'image entière sans ses détails. Il est donc intéressant de les stocker à part de manière à pouvoir y accéder très vite. Ca permet d'afficher une vignette sans devoir faire les transformées inverses de chaque bloc.
 - Ces DC varient peu d'un macrobloc à l'autre : dans une image normalement contrastée, les variations sont faibles. Au lieu de coder chaque DC0 de chaque bloc, on code uniquement les variations de ce coefficient d'un bloc à l'autre. On peut même employer RLE pour coder cette séquence.

Codage des coefficients AC0...AC63

Les autres coefficients représentent la composante alternative du macrobloc. Ceux-là sont codés en RLE appliquée à un parcours zigzag. On crée des triplets (longueur/4, nbits/? , valeur/nbites).

Pour gagner encore de la place, on utilise un codage entropique : Huffman qui représente les valeurs par des codes à longueur variable dépendant de la fréquence d'apparition.

Modes de transmission de l'image

Il existe différents modes de transmission des images compressées :

mode séquentiel

C'est le mode normal : on transmet d'abord l'ensemble des coefficients DC, puis bloc par bloc tous les coefficients AC.

mode progressif

Ce mode est employé lorsque les images sont transmises à faible vitesse mais qu'il faut afficher très vite une esquisse. On transmet donc en priorité les coefficients de basse fréquence de chaque bloc, puis on envoie les coefficients de haute fréquence.

mode hiérarchique

Il s'agit d'empiler plusieurs qualités d'images afin de pouvoir afficher une qualité donnée le plus vite possible. Chaque niveau utilise les données des niveaux précédents. Par exemple, on transmet d'abord une image ne contenant qu'1 pixel sur 16, puis 1/8 puis 1/4...

Avenir : Jpeg2000

La méthode Jpeg2000 fait appel à une autre transformée : les ondelettes. Il s'agit d'une transformée hiérarchique concernant la totalité de l'image à plusieurs niveaux de résolution imbriqués. Contrairement à la DCT, on peut travailler sur la totalité de l'image ; on n'est pas contraint de couper en macroblocs pour limiter le temps de calcul -- cela ne crée donc pas d'artefact sur les frontières des blocs. D'autre part, la transformée en ondelettes présente des propriétés intéressantes pour des éléments localisés dans une image.

Nous ne pourrons pas étudier cette transformée faute de temps.

Semaine 16 (09/01/2012) : Quelques informations sur le codage de vidéos

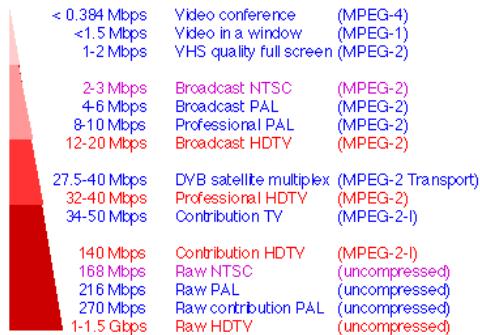
Concepts de la compression vidéo

La compression d'une séquence d'images fait appel aux meilleures techniques de compression pour une image isolée ainsi qu'à des techniques très élaborées pour l'aspect temporel : détection de mouvement et prédition.

Réflexions sur la compression de données

Tout d'abord, quel est le volume de données d'un film couleur durant 1h30 avec 25 images 720*560 par seconde ?

Chaque seconde il faut transférer 28.8 Mo vers l'écran. Un tel débit impose une compression pour le stockage et la transmission par les réseaux. Les normes, en particulier MPEG-n définissent différents débits selon les supports envisagés (image tirée de ce [lien](http://www.erg.abdn.ac.uk/research/future-net/digital-video/mpeg2.html) (<http://www.erg.abdn.ac.uk/research/future-net/digital-video/mpeg2.html>)) :

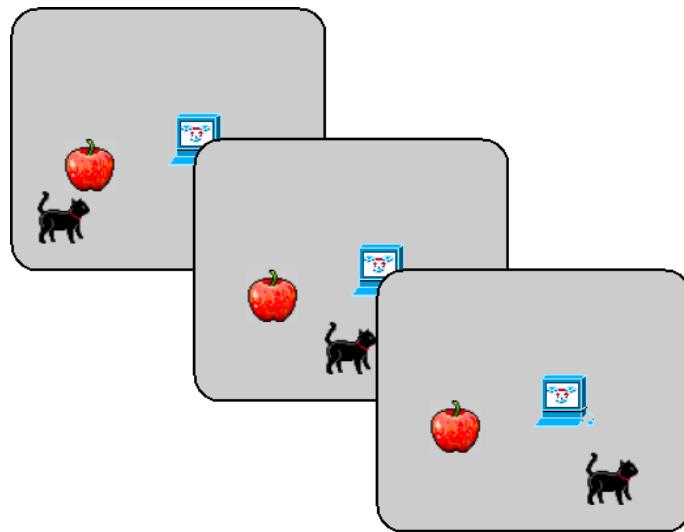


La compression ne se réduit pas seulement à l'aspect place occupée (sur disque ou bande passante). Il faut aussi prendre en compte le temps de calcul pour la dé/compression (temps réel possible ou pas, symétrique ou pas), savoir si la compression est sans perte ou pas.

Méthodes de compression

Il existe de nombreuses méthodes de compression de la vidéo. Le principe est toujours de détecter des corrélations dans les données. Ces corrélations sont en quelque sorte des informations redondantes, qui peuvent être représentées plus économiquement. Dans les images, on cherche des corrélations spatiales : des formes qui se répètent, des motifs qui peuvent être décrits par quelques coefficients d'une transformée...

Dans les vidéos, on cherche aussi des corrélations temporelles : des éléments qu'on retrouve d'une image à l'autre.



L'idée est de ne coder que ce qui change. Mais comme pour la compression jpeg, la découverte de corrélations intéressantes n'est pas triviale.

réduction de la redondance spatiale (intra-image)

Il s'agit de décorréler les pixels à l'intérieur d'une même image. La méthode JPEG utilise une DCT pour extraire les fréquences spatiales utiles. On peut aussi utiliser une transformée en ondelettes (Intel Indeo 5). On peut aussi faire appel à des techniques de prédiction (DPCM = Differential Pulse Code Modulation) pour réduire la quantité d'information concernant des pixels adjacents : dans une image "normale", les pixels adjacents sont souvent proches de la teinte moyenne -- il suffit donc de coder les différences entre cette prédiction et la réalité -- évidemment c'est faux près des contours, mais on y gagne sur l'image globale.

réduction de la redondance temporelle (inter-images)

Il s'agit de comparer les images successives et détecter le mieux possible toutes les ressemblances, en particulier les mouvements de zones. Les algorithmes utilisés sont des plus sophistiqués pour être rapides et précis.

Pour réduire la redondance temporelle, les méthodes mpeg font appel à des détecteurs de mouvement entre les images. Le compresseur économise le codage des pixels d'une image quand il constate qu'ils sont les mêmes que ceux d'une autre image, éventuellement déplacés d'une certaine distance.

Compression MJPEG et MJPEG2000

Le format MJPEG consiste à stocker une succession d'images encodées en JPEG. Les images sont mises l'une après l'autre dans le fichier. Comme la compression JPEG n'examine pas les corrélations temporelles, on n'arrive pas à un bon taux de compression global. Les fichiers MJPEG sont en général de très grande taille. La norme MJPEG2000 fait appel à la compression JPEG2000 qui est meilleure.

L'intérêt du format MJPEG n'est pas dans le taux de compression pour le stockage, mais dans la rapidité d'accès aux images. Le flux MJPEG autorise un accès immédiat à n'importe quelle image. Cela permet la recherche d'information et le montage vidéo. Au contraire, les formats MPEG et autres stockent les images dans un ordre spécifique : celui du décodage et non pas celui de la lecture, comme on va le voir ci-dessous.

Groupes d'images

Plus précisément, un compresseur MPEG emploie trois types de compression d'images qu'il alterne dans la vidéo : les images fournies en entrée sont compressées soit avec la méthode I, la méthode P ou la méthode B. Cela produit des données plus ou moins importantes dans le fichier de sortie. Par abus de langage, on parle d'images I, P ou B.

Image I

Il s'agit d'une image qui a été compressée avec une variante de JPEG : couleurs en YCrCb avec sous-échantillonnage des plans Cr et Cb, DCT, quantification un peu plus violente que pour une image fixe, encodage zigzag classique, avec les coefficients DC0 des macroblocs mis à part et encodés en relatif les uns par rapport aux autres.

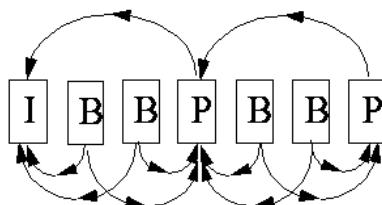
Image P

Une image P est compressée en comparant l'image fournie en entrée avec la précédente image de type P ou I du fichier. Le compresseur travaille avec des macroblocs 16x16 et indique seulement à quel autre bloc il ressemble dans l'image précédente et s'il y a un mouvement ou une erreur de prédiction (changements dans les coefficients DCT). Le compresseur peut également ignorer le macrobloc s'il ne change pas assez et au pire, s'il change trop, il le code comme dans une image I. Plus on accumule des images de type P, plus le résultat de la décompression sera imparfait. Il faut donc régulièrement insérer une image I.

Image B

Une image devant être compressée avec la méthode B est comparée à toutes les images de type I ou P précédente ou suivante, à la recherche de la plus ressemblante. Le principe est le même que pour les images P : vecteurs de déplacement et informations sur les changements dans la DCT du macrobloc. Les images B sont encore moins précises que les images P car elles s'appuient sur des images elle-mêmes imprécises.

Ce dessin représente les dépendances entre les images. Il faut comprendre : les pixels de telle image sont élaborés à partir des pixels de l'autre image ainsi que des informations de mouvement et d'évolution de coefficients DCT. Seules les images I ont une certaine qualité. Les erreurs sur les autres images s'accumulent.



type d'image	image initiale	image décodée
I		
B		
B		
B		
P		
B		
B		
B		
I		

Représentation en fichier

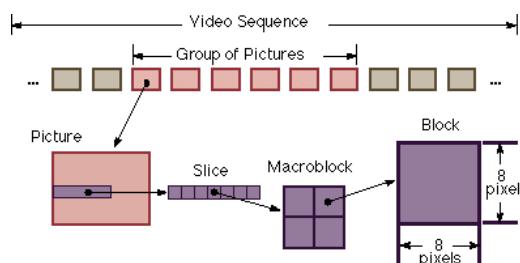
Les données des GOP sont enregistrées en binaire dans le fichier résultat. Un point très important de la représentation en fichiers est de pouvoir récupérer les erreurs de transmission. Il est donc important de placer des marqueurs caractéristiques à intervalles réguliers dans le fichier.

A noter que dans le fichier, les images d'un GOP ne se suivent pas dans l'ordre temporel, mais dans l'ordre nécessaire pour le décodage : l'image I suivie des images P suivies des images B. Cela ralentit fortement l'accès à une image aléatoire puisque par exemple pour décoder une image B, il faut avoir décodé l'image I et/ou l'image P dont dépend celle qu'on veut.

Dans le système MPEG, la structuration est faite ainsi :

- fichier mpeg = suite de GOP
- 1 GOP = suite d'images (frame)
- 1 image = suite de tranches (slice)
- 1 tranche = suite de macroblocs
- 1 macrobloc = 6 blocs 8x8 (4 blocs Y, 1 bloc Cr et 1 bloc Cb) + infos de mouvements

Voici une illustration tirée de [cette page](http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/sab/report.html) (http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/sab/report.html) :



Ensuite, au niveau du stockage dans un fichier ou de la transmission par un réseau, toutes ces informations sont encapsulées dans ce qu'on appelle un *format de transport* (transport stream). Le format de transport rassemble les informations provenant des compressseurs vidéo (1 piste au moins) et audio (1 ou + pistes) et éventuellement d'autres informations selon la source : timecode... On appelle *multiplexage* cet assemblage des pistes et *démultiplexage* l'opération inverse qui sépare le fichier ou flux réseau en image et son. Ainsi, dans le fichier résultant, les données image et audio sont entrelacées, ce qui permet de les jouer en parallèle.

Il y a deux formats de transport pour la compression MPEG : le format TS (transport stream) qui ne contient pas d'informations de type timecode (horloge pour dater les images comme sur un caméscope),

utilisé en transmission satellite, et le format PS (program stream) qui est utilisé pour l'enregistrement sur DVD.

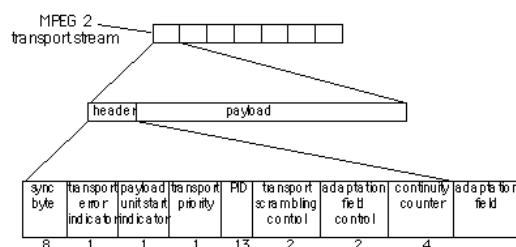
Pour faciliter la transmission sur réseau et le stockage dans un fichier, les données sont découpées en paquet de 188 octets, taille correspondant à différents types de réseaux (ATM) et optimisée par rapport au taux d'erreur moyen. Certains paquets transportent la vidéo, d'autres transportent l'audio. Chaque paquet est identifié par un nombre (PID paquet identifier).

Voici un exemple tiré de cette [page](http://www.vbrick.net/Topics/transport_stream.asp) (http://www.vbrick.net/Topics/transport_stream.asp), on y voit deux types de paquets : 51 (vidéo) et 64 (audio).



L'alternance des paquets vidéo/audio n'est pas fixée dans le temps par le multiplexeur, mais seulement selon l'avancement du travail des compresseurs. C'est aux décodeurs de rester synchronisés.

Plus en détail encore, pour les curieux, voici la structure d'un paquet Mpeg-2 :



On y voit les 4 premiers octets pour décrire le paquet, dont le PID sur 13 bits. Ensuite, la charge utile (payload). Dans le cas des transmissions par satellite (DVB-S), on trouve en plus 16 octets pour le code de cryptage (AES ou autre).

Normes

Pour revenir sur les normes développées pour la compression vidéo :

MPEG-1 (1988)

Cette norme est la plus ancienne, elle a été développée pour des vidéos 352x240, 30 im/s ou 352x288 à 25im/s. La norme MPEG-1 audio est le très connu mp3. Cette norme est destinée aux vidéo-cd (l'ancêtre du DVD).

MPEG-2 (1994)

Cette norme complète la précédente avec de nombreuses résolutions. Elle est destinée à la télévision numérique par cable, satellite, TNT et ADSL, aux DVD.

MPEG-4 (1998)

Elle complète la norme précédente pour les bas débits, le contenu 3D et les droits numériques (DRM).

H.263 (1996)

Pour des lignes à très bas débit et basse résolution : téléphones portables, pour la visioconférence, elle apporte des améliorations à MPEG-2 : davantage de flexibilité dans les choix du compresseur (vecteurs de mouvement et codage des informations RLE) et la correction des erreurs.

H.264 ou MPEG-4/AVC (2005)

Elle affine très nettement les algorithmes de détection de mouvement : nombreuses possibilités de comparaison entre les images, autres tailles de macroblocs, précision supérieure pour la modélisation des mouvements, filtrage pour réduire les artefacts des frontières de bloc, autres transformées, meilleure synchronisation du son et des images... Toutes ces améliorations significatives lui donnent une nette supériorité sur MPEG-2.

Les normes évoluent extrêmement vite. De nouveaux formats sont proposés chaque année pour améliorer la qualité de la compression.

Fichiers AVI

Structure d'un fichier AVI

Le format AVI (audio video interleave) est un format de transport. Il permet d'entrelacer les pistes vidéo et les pistes audio. Ce n'est pas un format de compression. La compression est effectuée par des codecs indépendants et au choix de l'utilisateur ou du logiciel de compression, par exemple, indeo, xvid, divx, mp4...

Un fichier AVI suit la norme RIFF (celle des fichiers PNG et Wave) qui consiste à structurer les données en *chunks*. Un chunk regroupe des données (pixels, codes compressés,...), des descriptions (entête) et éventuellement d'autres chunks. Un fichier AVI est lui-même un chunk. La doc se trouve [ici](http://msdn2.microsoft.com/en-us/library/ms779636.aspx) (<http://msdn2.microsoft.com/en-us/library/ms779636.aspx>).

Un chunk commence par 4 octets qui sont à interpréter comme 4 codes ascii de caractères (FOURCC), par exemple 'LIST' ou 'RIFF'. Le système windows est basé sur une machine poids fort devant (little-endian ou lsb first), 'abcd' est donc représenté par le nombre 32 bits : 0x64636261.

Après ces 4 octets (ckID), on trouve la taille du chunk (ckSize) sur 4 octets puis ses données (qui doivent faire exactement ckSize octets). Comme ckSize est arrondi au multiple de 4 supérieur ou égal, les données sont complétées avec 0 à 3 octets nuls.

Windows définit trois types de chunks :

le fichier avi

Un fichier AVI est un chunk à lui seul : sa structure est 'RIFF' taille_fichier_entier/4 type/4 données. Le type est un code FOURCC qui vaut 'AVI' pour les fichiers AVI (l'espace est significatif).

une liste

Certains chunks contiennent des énumérations, leur structure est 'LIST' taille_de_la_liste/4 type/4 données. Le type est un code FOURCC qui donne le type de la liste, par exemple : 'hdrl'.

un chunk ordinaire

comme dit précédemment, la structure est 'XYZT' taille contenu.

Voici la structure plus fine d'un fichier AVI :

```
RIFF ('AVI'
  LIST ('hdrl'
    'avih'(<Main AVI Header>)
    LIST ('strl'
      'strh'(<Stream header>)           // type de la piste (vidéo, audio...)
      'strf'(<Stream format>)           // caractéristiques de la piste
      [ 'strd'(<Additional header data>) ] // informations pour le codec
      [ 'strn'(<Stream name>) ]          // texte d'information sur la piste
      ...
    )
    ...
  )
)
```

```

        )
LIST ('movi'
    {SubChunk | LIST ('rec '
        SubChunk1
        SubChunk2
        ...
    )
    ...
}
...
)
['idx1' (<AVI Index>) ]
)

```

Chunk hdrl

donne des informations sur le fichier vidéo : nombre de pistes (stream) vidéo et audio, dimensions de la séquence.

Chunk strl

il en faut un par piste de données, il contient deux types de sous-chunk : strlh entête de piste et strf format de piste.

Voici le contenu exact du chunk hdrl :

```

typedef struct _avimainheader {
    FOURCC fcc;
    DWORD cb;
    DWORD dwMicroSecPerFrame;
    DWORD dwMaxBytesPerSec;
    DWORD dwPaddingGranularity;
    DWORD dwFlags;
    DWORD dwTotalFrames;
    DWORD dwInitialFrames;
    DWORD dwStreams;           // nombre de flux, par exemple 2 s'il y a de la
    // vidéo et une piste audio
    DWORD dwSuggestedBufferSize;
    DWORD dwWidth;            // taille de l'image en nombre de pixels
    DWORD dwHeight;
    DWORD dwReserved[4];
} AVIMAINHEADER;

```

Voici le contenu exact d'un chunk strh :

```

typedef struct _avistreamheader {
    FOURCC fcc;
    DWORD cb;
    FOURCC fccType;          // 'auds'=audio stream, 'mids'=MIDI stream, 'txts'=text
    // stream, 'vids'=video stream
    FOURCC fccHandler;       // nom du codec préféré pour le décodage, ex : DX50
    DWORD dwFlags;
    WORD wPriority;
    WORD wLanguage;
    DWORD dwInitialFrames;
    DWORD dwScale;
    DWORD dwRate;            // images par seconde
    DWORD dwStart;
    DWORD dwLength;
    DWORD dwSuggestedBufferSize;
    DWORD dwQuality;
    DWORD dwSampleSize;
    struct {
        short int left;
        short int top;
        short int right;
    }
}

```

```

        short int bottom;
    } rcfFrame;
} AVISTREAMHEADER;

```

Voici les détails d'un chunk strf pour une piste vidéo, il s'agit simplement d'un chunk d'information sur une image, comme dans les fichiers bmp vus dans la première semaine de ce cours :

```

typedef struct tagBITMAPINFO {
    DWORD biSize;
    DWORD biWidth;
    DWORD biHeight;
    WORD biPlanes;
    WORD biBitCount
    DWORD biCompression;
    DWORD biSizeImage;
    DWORD biXPelsPerMeter;
    DWORD biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
    RGBQUAD bmiColors[1];
} BITMAPINFO;

```

Pour les pistes audio, il y a une structure aussi, qui est celle des fichiers wave :

```

typedef struct {
    WORD wFormatTag;
    WORD nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD nBlockAlign;
    WORD wBitsPerSample;
    WORD cbSize;
} WAVEFORMATEX;

```

On arrive au chunk 'movi' qui contient les données vidéo compressées par le codec. Le chunk movi contient une multitude de sous-chunks nommés d'après la piste qu'ils contiennent : 'NNcc', NN est le numéro sur deux chiffres de la piste (00 pour la première), et cc est un code qui indique ce que contient le chunk : 'db' pour des données non compressées, 'dc' pour des données compressées, 'wb' pour des données audio.

Voici par exemple, le début d'un fichier avi :

```

000000 R I F F           A V I sp L I S T
      52 49 46 46 04 73 37 00 41 56 49 20 4c 49 53 54
000010           h d r l a v i h
      c0 00 00 00 68 64 72 6c 61 76 69 68 38 00 00 00
000020           6a 04 01 00 00 28 23 00 00 00 00 00 10 06 00 00
000030           6b 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00
000040           40 01 00 00 f0 00 00 00 01 00 00 00 0f 00 00 00
000050           L I S T
      00 00 00 00 0f 00 00 00 4c 49 53 54 74 00 00 00
000060 s t r l s t r h           v i d s
      73 74 72 6c 73 74 72 68 38 00 00 00 76 69 64 73
000070 m s v c
      6d 73 76 63 00 00 00 00 00 00 00 00 00 00 00 00
000080           01 00 00 00 0f 00 00 00 00 00 00 00 6b 00 00 00
000090           00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00 00
0000a0           s t r f
      40 01 f0 00 73 74 72 66 28 00 00 00 28 00 00 00

```

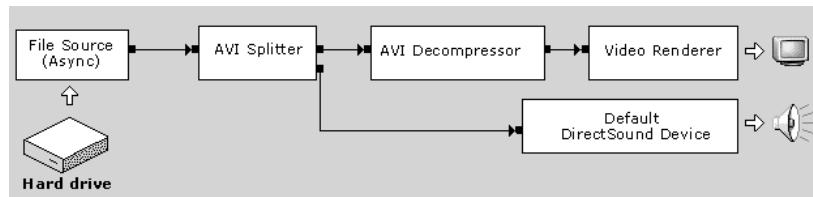
En conclusion, on peut remarquer que les fichiers AVI sont plus descriptifs que les fichiers Mpeg, ils contiennent beaucoup d'informations sur les données vidéo et audio.

Programmation avec DirectShow

DirectShow est la librairie Windows pour travailler avec tout ce qui est vidéo et audio : capture, compression, lecture... Elle est capable de traiter toutes sortes de formats de fichiers et de périphériques.

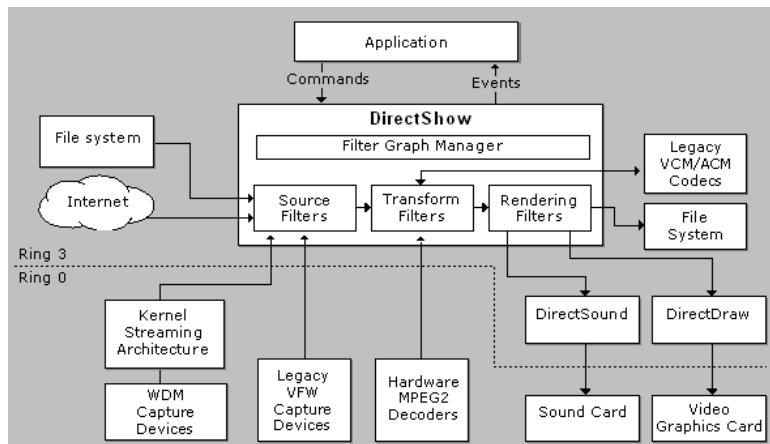
Le problème majeur de DirectShow est l'architecture logicielle orientée objet sur laquelle elle repose : Component Object Model. Sa mise en oeuvre est... euh on va dire... sans doute mal expliquée :-).

Le principe général de DirectShow est de travailler avec des filtres et de les assembler avec des sortes de tubes comme des commandes sur Unix. Un filtre est une sorte de processus qui effectue un traitement sur un flux de donnée : fichier, piste vidéo, piste audio... Il existe plusieurs types de filtres prédéfinis, voici un exemple d'assemblage extrait de la [documentation MSDN](http://msdn2.microsoft.com/en-us/library/ms786509.aspx) (<http://msdn2.microsoft.com/en-us/library/ms786509.aspx>) :

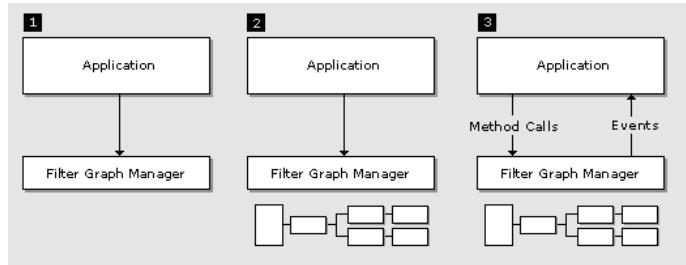


Sur Unix, on enchaîne les commandes d'un tube avec le signe |. Sur Windows, c'est nécessairement plus compliqué car les filtres ne sont pas séquentiels (branches, dérivations...). L'ensemble constitue un graphe de filtres et doit être géré par un gestionnaire de graphe de filtres (Filter Graph Manager). Il y a un grand nombre de filtres et ils travaillent avec le matériel. L'un des points importants est la gestion du temps. Un autre est l'envoi synchronisé de commandes à l'ensemble des filtres.

Voici l'architecture globale de DirectShow :



Une application DirectShow destinée à lire ou créer des fichiers AVI doit d'abord créer un Gestionnaire de Graphe de Filtre puis lui faire créer l'architecture qu'elle veut. Ensuite, elle pilote les filtres via le gestionnaire et peut recevoir des notifications.



Le gestionnaire dispose de quelques méthodes pour construire et commander le graphe : établir une connexion entre deux pattes (pins), créer un graphe prédefini pour jouer un fichier... Les filtres ont une ou plusieurs pattes (comme les circuits intégrés en électronique), certaines sont des entrées, d'autres des sorties. D'autre part, on peut configurer ce gestionnaire pour qu'il devienne plus complet : qu'il sache faire d'autres choses comme se rajouter des contrôles pour démarrer ou stopper la lecture du fichier... on appelle cela une interface.

On reviendra sur la programmation DirectShow en [S4P4, semaine 22](#) (S4P4%20-%20Outils%20IN.html#semaine22).

Lecture d'un fichier AVI

Voici un court exemple de programme tiré de la [documentation](#) (<http://msdn2.microsoft.com/en-us/library/ms783787.aspx>) qui joue un fichier (ne pas trop chercher à comprendre les appels système) :

```
#include <dshow.h>           // fichier à inclure dans toute application DirectShow
void main(void)
{
    IGraphBuilder *pGraph = NULL;
    IMediaControl *pControl = NULL;
    IMediaEvent   *pEvent = NULL;

    // Initialisation de la librairie COM
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr)) {
        printf("ERROR - Could not initialize COM library");
        exit(1);
    }

    // Création du gestionnaire de filtres et récupération d'éléments de commande
    // (interfaces)
    hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
    IID_IGraphBuilder, (void **)&pGraph);
    if (FAILED(hr)) {
        printf("ERROR - Could not create the Filter Graph Manager.");
        exit(1);
    }

    // rajouter une interface permettant de démarrer (run) ou stopper (pause, stop)
    // la lecture du fichier
    hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);

    // rajouter une interface permettant de savoir ce qui se passe, savoir si c'est
    // fini (voir ci-dessous une amélioration)
    hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);

    // Créer le graphe prédefini pour la lecture d'un fichier
    hr = pGraph->RenderFile(L"C:\\bimbamboum.avi", NULL);
    if (SUCCEEDED(hr)) {

    }
}
```

```

    // utiliser le contrôle run() de l'interface MediaControl pour lancer le film,
    // c'est géré dans un thread indépendant, donc on a encore la main
    hr = pControl->Run();
    if (SUCCEEDED(hr)) {
        // attendre que la vidéo se termine : méthode de l'interface MediaEvent
        // (on peut/doit mettre un timeout en ms à la place de INFINITE)
        long evCode;
        pEvent->WaitForCompletion(INFINITE, &evCode);
    }
}

// libération des ressources
pControl->Release();
pEvent->Release();
pGraph->Release();
CoUninitialize();
}

```

Dans cet exemple, on boucle indéfiniment jusqu'à ce que le film soit fini. C'est pas très intéressant dans une application windows. On peut améliorer la gestion des événements : être prévenu à la manière windows quand le film est fini ou qu'il y a quelque chose.

```

IMediaEventEx *g_pEvent = NULL;
....
g_pGraph->QueryInterface(IID_IMediaEventEx, (void **)&g_pEvent);
g_pEvent->SetNotifyWindow((OAHWND)hwndFenetrePrincipale, WM_GRAPHNOTIFY, 0);
...

....WindowProc()
{
    ...
    case WM_GRAPHNOTIFY:
        if (g_pEvent != NULL) HandleGraphEvent();
        break;
    ...
}
void HandleGraphEvent()
{
    // récupérer le code de l'événement
    long evCode;
    LONG_PTR param1, param2;
    HRESULT hr;
    while (SUCCEEDED(g_pEvent->GetEvent(&evCode, &param1, &param2, 0))) {
        g_pEvent->FreeEventParams(evCode, param1, param2);
        // traiter l'événement selon son code
        switch (evCode) {
            case EC_COMPLETE:
                ...
                break;
            case EC_USERABORT:
                ...
                break;
            case EC_ERRORABORT:
                CleanUp();
                PostQuitMessage(0);
                return;
        }
    }
}

```

Création d'un fichier AVI

La création d'un fichier AVI par une application, par exemple de synthèse d'images, qui crée différentes images est relativement compliquée. Curieusement, le cas n'est pas prévu d'office dans DirectShow. Il faut écrire un filtre qui fait ce travail. Ce [lien](http://whorld.org/temp/BmpToAvi.html) (<http://whorld.org/temp/BmpToAvi.html>) fournit tout le code nécessaire.

AVI sur Linux

Sur Linux, on fait appel à la librairie [avifile](http://avifile.sourceforge.net/) (<http://avifile.sourceforge.net/>) (libaviplay). C'est extrêmement simple, voici un petit exemple, trois fonctions pour enregistrer des images successives dans un fichier avi, sans compression. Le principe est de construire un bitmap avec l'image et de rajouter ce bitmap dans le fichier : comment faire plus simple ?

```
#include <avifile-0.7/avifile.h>
#include <avifile-0.7/videoencoder.h>
#include <avifile-0.7/image.h>
#include <avifile-0.7/avm_fourcc.h>
#include <avifile-0.7/avm_cpuinfo.h>
#include <avifile-0.7/utils.h>
#include <avifile-0.7/avm_creators.h>
#include <avifile-0.7/avm_except.h>

// descripteur du fichier AVI
mutable avm::IWriteFile *aviFile;
mutable avm::IVideoWriteStream *aviStream;

void OpenAVIFile(char *nom, int width, int height)
{
    // création du fichier de sortie
    aviFile = avm::CreateWriteFile(nom);

    // codec choisi pour la compression : aucun
    fourcc_t codec = RIFFINFO_I420;
    //fourcc_t codec = fccHFYU;
    //fourcc_t codec = RIFFINFO_DX50;
    //fourcc_t codec = RIFFINFO_MP2;

    // description des images qu'on va y mettre (taille, résolution...)
    BITMAPINFOHEADER bi;
    memset(&bi, 0, sizeof(bi));
    bi.biSize = sizeof(bi);
    bi.biWidth = width;
    bi.biHeight = height;
    bi.biSizeImage = (bi.biWidth * bi.biHeight * 3) / 2;
    bi.biPlanes = 3;
    bi.biBitCount = 12;
    bi.biCompression = codec;

    int frameRate = 10;
    if (frameRate<1) frameRate=1;
    aviStream = aviFile->AddVideoStream(codec, &bi, 1000*1000/frameRate);

    SaviStream->SetQuality(10000);
    aviStream->Start();
}

void CloseAVIFile()
{
    if (aviStream) {
        aviStream->Stop();
        delete aviFile;
        aviFile = NULL;
    }
}
```

```
    aviStream = NULL;
}

void SaveImageAVI(unsigned char *yuv, int width, int height)
{
    BITMAPINFOHEADER bi;

    memset(&bi, 0, sizeof(bi));
    bi.biSize = sizeof(bi);
    bi.biWidth = width;
    bi.biHeight = height;
    bi.biSizeImage = (bi.biWidth * bi.biHeight * 3) / 2;
    bi.biPlanes = 3;
    bi.biBitCount = 12;
    bi.biCompression = fccI420;

    bi.biHeight = - bi.biHeight;
    avm::BitmapInfo info(bi);
    avm::CImage img(&info, yuv, false);
    aviStream->AddFrame(&img);
}
```