MASTER EN SCIENCES INFORMATIQUES

Compression d'images ""Gray Scale "" par Fractales

Luppi, Marc

Award date:
1996

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Compression d'Images « Gray Scale » par Fractales

Mémoire présenté par Marc LUPPI pour l'obtention
du grade de maître en informatique

Promoteur : Prof. Jean FICHEFET

Année académique 1995-1996

Abstract

Data compression methods, considered for a long time as laboratory's curiosities, and used exclusively in research centers, far away from the public, have become, since a few years, a speaking subject of everybody concerned with computer science. This is surely due to the growing importance of multimedia and communication technologies : databases, images, movies, sounds, ... require enormous storage spaces and thus considerable transmission times. The objective followed by data compression is to write data in a different way, more compact, containing the same information, or nearly.

Among the most often mentionned image compression techniques, one find the discrete cosine transform based methods and a rather promising one : fractal compresssion, that's only ten years old. This work aims to detail the underlying mechanisms behind gray scale fractal image compression algorithms : after some generalities about images and compression techniques, iterated function systems and collages will be explained. To illustrate the theory, a little compression / decompression program will be exposed.

Abstract

Les méthodes de compression de données, longtemps considérées comme des curiosités de laboratoire et utilisées presque exclusivement dans des centres de recherches, loin du grand public, connaissent depuis quelques années un regain d'intérêt. L'avènement de l'ère multimédia et des technologies de communication n'y est évidemment pas étranger : les banques de données, les images statiques et animées, les sons, ... requièrent des espaces de stockage phénoménaux et donc des temps de transmission considérables. L'objectif visé consiste à inscrire les données sous une autre forme, plus compacte, renfermant les mêmes informations, ou presque...

Parmi les techniques le plus souvent citées pour les images, outre les méthodes à base de transformées en cosinus discrètes, on rencontre un procédé qui fête ses dix ans : la compression fractale. Ce travail se propose de lever un coin du voile sur les mécanismes mis en oeuvre lors de l'élaboration d'algorithmes de compression d'images fixes « gray scale » à base de fractales : après quelques généralités sur les images et les différentes techniques de compression, systèmes de fonctions itérées et collages y seront abordés. Pour illustrer la théorie, un petit programme de compression / décompression sera détaillé.

J'aimerais ici remercier les enseignants de tous niveaux qui ont eu à me subir comme élève, pour avoir tenté de m'inculquer un peu de leur savoir et de leur passion.

Je remercie particulièrement le professeur Jean Fichefet pour sa gentillesse et ses remarques pertinentes.

Une grosse tape amicale également sur l'épaule de Fabian Piret pour ses conseils judicieux.

Enfin, pour m'avoir permis d'entreprendre des études, je remercie ma famille, en particulier mes frères qui m'ont toujours soutenu dans le besoin.

Sommaire

La transformation fractale et la compression fractale proprement dites, le coeur du problème, sont exposées aux chapitres 6 et 7.

Ces concepts ne peuvent être introduits que si l'on maîtrise certaines notions sur les fractales et les images.

Le chapitre 5 expose la théorie des fractales IFS, il est précédé par les rappels mathématiques du chapitre 4.

Le chapitre 2 donne une formalisation mathématique des images et des termes s'y rapportant, il est introduit par le chapitre 1.

Le chapitre 3, quant à lui, apporte le vocabulaire relatif à l'univers de la compression de données et détaille quelques algorithmes de compression parmi les plus réputés.

Les annexes contiennent, d'une part, un programme de construction des IFS, pour illustrer l'obtention d'images complexes à partir de quelques coefficients réels seulement, et, d'autre part, un petit programme simple mettant en route un mécanisme de compression / décompression par fractales sur des images BMP 256 niveaux de gris.

Pour compléter ces annexes, on découvrira la description du format de fichier BMP ainsi qu'un court programme d'affichage de telles images.

Table des matières

Table des matières

Chapitre 1 : Généralités sur les images

Ce premier chapitre, bref, brosse quelques aspects généraux à connaître pour aborder les chapitres suivants, en particulier le chapitre 2.

I. Codage des images : des images aux nombres

Depuis toujours, les artistes ont su graver, peindre et dessiner directement sur une surface plane pour créer des images représentant des objets réels. L'utilisation d'un ordinateur pour la création d'images nécessite une étape supplémentaire. Avant que la machine n'affiche les idées de l'artiste à l'écran ou sur l'imprimante, ces idées doivent être converties en séquences de bits, seules traitables par la machine.

Ainsi, une **image numérique** est définie comme une étendue plane entièrement remplie par des éléments graphiques élémentaires appelés **pixels** (picture elements), et disposés suivant une trame, souvent rectangulaire. A chaque point correspond des nombres qui permettent de coder sa position et sa couleur. Une couleur quelconque à l'écran est définie par le mélange de trois composantes primaires : le rouge, le vert et le bleu. Chaque pixel comporte donc une part de chacune de ces couleurs de base. De l'intensité plus ou moins grande de chacune d'elles dépendra la couleur finale. Par exemple, si chaque composante est à son maximum d'intensité, le pixel sera blanc. A l'inverse, si chacune est à son minimum, le pixel sera noir. Sur les systèmes actuels, il est possible d'utiliser jusqu'à 8 bits pour coder chaque couleur de base, ce qui se traduit par 256 teintes par composante. Etendu aux trois composantes de base, on obtient un codage sur 24 bits, qui autorise une palette de $256^3 = 2^{24}$ teintes simultanées, soit plus ou moins 16,7 millions de couleurs. Les images 24 bits, c.-à-d. les images dont les couleurs sont stockées sur 24 bits, sont dites en « vraies couleurs » car le nombre de teintes que l'on peut obtenir est supérieur au nombre de couleurs que l'oeil humain peut distinguer, soit aux alentours de 350 000.

Les pixels ne suffisent pas en eux-mêmes ; la création d'une image implique la mise en place d'une structure. Deux systèmes servent ainsi à décrire une image numérique : le mode vectoriel et le mode point (bitmap).

II. Modes de représentation des images : images vectorielles et en mode point (bitmap)

Dans le **mode vectoriel**, chaque objet graphique est créé à partir d'une définition géométrique des formes (ligne, cercle, courbe de Bézier, ...) auxquelles sont associés des attributs (couleur, épaisseur, ...). Chaque objet est stocké, dans ce cas, non sous la forme de points en mémoire, mais sous la forme de primitives géométriques dont on ne conserve que les données significatives, telles que le centre et le rayon d'un cercle, les courbures ou les longueurs des segments, etc., qui permettront à tout moment de retracer l'image sans altération.

Ce type de représentation trouve son utilisation essentiellement dans les applications de type DAO (dessin technique).

Les principales caractéristiques de ce type de représentation sont les suivantes :

- La description des objets est indépendante de la définition des périphériques graphiques. Ainsi, une figure géométrique dessinée sur un écran de faible définition peut être sortie sur une table traçante avec une haute définition.

- La description vectorielle permet de hiérarchiser les différents objets graphiques, et par conséquent de créer des images complexes à partir d'éléments plus simples.

- Les objets graphiques peuvent facilement être modifiés de façon interactive sans altérer le reste de l'image. Il est ainsi très aisé, à tout moment, de faire tourner, de déplacer, de copier, de superposer, etc., les éléments souhaités.

- En fonction du type de périphérique d'affichage, l'image vectorielle doit être transformée pour être compréhensible par ce dernier. Cette opération s'appelle la **génération du tracé**.

- Le stockage des données vectorielles est beaucoup moins gourmand en espace que dans le cas des données « bitmap ». De plus, la taille du fichier est indépendante de la résolution souhaitée.

Dans le **mode bitmap**, une image est constituée d'une mosaïque de pixels, identifiés par une position et une valeur de couleur. Ce type de représentation correspond donc au mode de codage le plus « primitif » de l'image.

Le mode bitmap est orienté « peinture » ou traitement d'images numérisées. Il convient parfaitement pour les applications issues du domaine des arts graphiques et de l'édition.

Les principales caractéristiques de ce type de représentation sont les suivantes :

- Lors de l'agrandissement ou de la réduction d'une image, chacun des points constituant celle-ci est également grossi ou diminué. Ainsi, un zoom effectué sur le bord d'un cercle

va engendrer un grossissement de celui-ci avec, en plus, un effet d'escalier ; c'est l'**effet gros pixel**.

- La manière de travailler sur une image bitmap est proche de celle utilisée traditionnellement par les graphistes : emploi de masques, de l'aérographe, de brosses, ...

- Ce type de représentation permet de modifier et de traiter facilement des images complexes ayant une définition quasi photographique.

- Ce type de représentation est dépendant de la définition des périphériques utilisés. Ainsi, une image à haute définition sur un écran n'engendre pas automatiquement une image à haute définition sur une imprimante laser.

- Les images bitmap sont gourmandes en espace de stockage sur disque : une simple image de résolution courante 1024 * 768 en vraies couleurs accapare, en format brut, près de 2,4 mégaoctets d'espace.

- Les modifications de parties de dessin sont moins évidentes qu'en mode vectoriel, car les différentes entités constituant ce dessin ne sont pas reconnues comme des objets indépendants, mais sont « fondues » l'une dans l'autre.

Chapitre 2 : Modèles mathématiques pour les images du monde réel

Ce chapitre propose une modélisation mathématique des concepts intuitifs tels que les images, la résolution, la couleur, ... Etant donné que la compression fractale repose entièrement sur des bases mathématiques, c'est sous cet aspect que doivent être présentées les notions qui nous sont familières.

I. Propriétés des images du monde réel

Essayons d'abord de définir ce que nous entendons par « image du monde réel ». La perception intuitive des images du monde qui nous entoure est trop complexe pour en faire ici, un modèle descriptif simple à utiliser. En effet, on imagine souvent qu'une image pourrait se définir comme le résultat de sortie d'un appareil photographique (dia, positif, négatif, ...), mais si l'on y regarde bien, on se retrouve embarrassé de constater qu'un zoom important sur une région de la photographie ne nous permet pas toujours de discerner les détails que l'on observe lorsque, dans le monde réel, on se rapproche de l'endroit où a été prise la photo. On objectera, bien sûr, que cela provient du fait que la finesse du grain du papier de la photo n'est pas assez grande, et donc qu'il suffit de considérer le négatif. Le problème reste le même, on se heurte à la finesse de l'émulsion chimique. Qu'à cela ne tienne, on pourrait imaginer que la technologie moderne nous a offert un appareil à téléobjectif révolutionnaire, qui peut zoomer autant de fois que l'on désire. Là encore, on constate que les lois de l'optique anéantissent notre enthousiasme, en limitant ce facteur de zoom. Peu importe, on transforme notre appareil concret en appareil conceptuel à facteur de zoom infini, qui nous permettra de fixer sur une pellicule conceptuelle ce que l'on croit être une image du monde réel. C'est sans compter les limitations imposées par la nature elle-même, les incertitudes d'Heisenberg et le fait que, à une certaine échelle, plus aucun photon de longueur d'onde convenable ne peut venir impressionner la pellicule.

Une **image du monde réel** est donc une entité idéalisée, qui peut s'assimiler à une photographie issue d'un appareil idéal à facteur de zoom réglable indéfiniment, et qui se joue des lois imposées par la nature. Les images ainsi définies ne sont pas celles que notre système visuel peut ou pourrait recevoir, ce sont plutôt celles que notre imagination est capable de concevoir.

Nous noterons U l'ensemble de toutes les images du monde réel, et T une de ces images, c'est-à-dire un élément de U. Nous allons essayer de donner quelques propriétés que possèdent ces images du monde réel, elles nous seront bien utiles quand nous tenterons d'établir des modèles mathématiques de ces images.

Propriété 1 : Toute image $T \in U$ possède un **support** et des **dimensions physiques**. Le support est un ensemble $\mathfrak{I} \subset \mathfrak{R}^2$, où $\mathfrak{R}^2$ est le plan euclidien, défini par

$$\mathfrak{I} = \{(x, y) : a \leq x \leq b, c \leq y \leq d : a, b, c, d \in \mathfrak{R}\}.$$

Les dimensions physiques de T sont (b - a) et (d - c) unités de longueur.

Le support d'une image peut se concevoir intuitivement comme le support physique sur lequel repose l'image (figure 2.1) : transparent, écran de projection, ...

On considérera qu'un **point d'une image** est le point du support qui lui correspond, de même, une **région d'une image** est la région correspondante du support qui supporte l'image, et ainsi de suite.

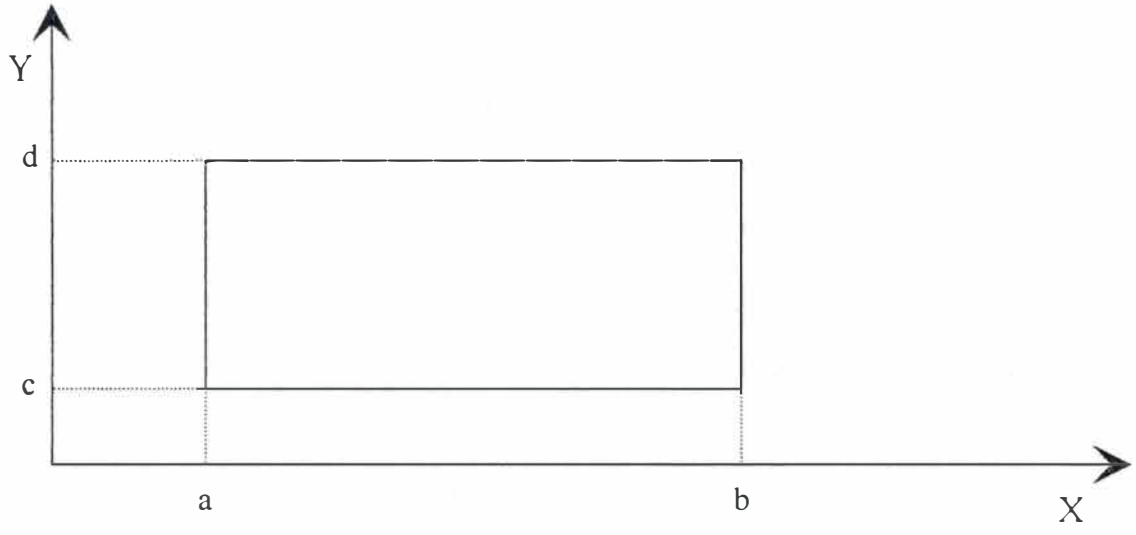


figure 2.1 : support d'une image

On peut calculer la distance d(x, y) qui sépare deux points $x = (x_1, x_2)$ et $y = (y_1, y_2)$ d'une image par la formule

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}.$$

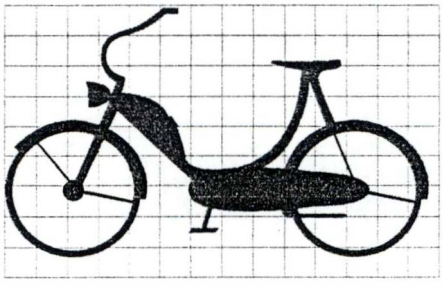
Le plan euclidien muni ainsi d'une métrique constitue un **espace métrique**. Les propriétés topologiques de cet espace et la classification des sous-ensembles du support de l'image (intérieur, ouverture, ...) sont préservées par transformation de support de métrique équivalente. Nous détaillerons cela dans le chapitre 4.

Propriété 2 : Soit $T \in U$. T possède des **attributs chromatiques**. Ces attributs chromatiques sont la couleur et l'intensité lumineuse de sous-ensembles de l'image. On pourra modéliser ces attributs à l'aide de fonctions à valeur dans $\mathfrak{R}$ ou de mesures de Borel à valeur dans $\mathfrak{R}$, sur le support de l'image.

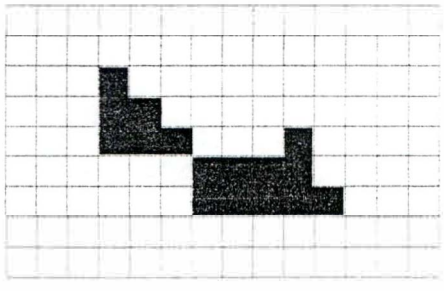
Un cas particulier est le cas où le support est divisé par une grille en sous-ensembles rectangulaires. Les images ayant pour support ces sous-ensembles sont appelées **pixels**. Ces pixels possèdent leurs propres attributs chromatiques.

Propriété 3 : Soit T ∈ U. T est indépendante de la résolution, c'est-à-dire que l'on peut considérer un maillage du support aussi fin que l'on veut (voir figure 2.2).

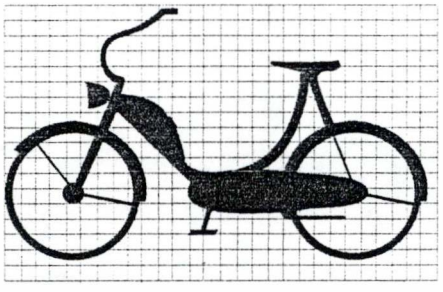
Image



Grille de digitalisation sur l'image



Approximation de résolution finie de l'image



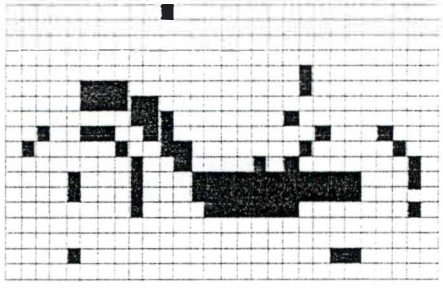


figure 2.2 : approximations de résolution finie de l'image

Propriété 4 : L'espace U des images du monde réel est fermé sous l'opération d'extraction. Soit T ∈ U, on construit T' en délimitant une zone rectangulaire de T, dont les côtés sont parallèles aux côtés de T. Alors, T' ∈ U. Donc, toute image du monde réel, aussi petite soit elle, est une image valide.

Propriété 5 : Soit T ∈ U. On construit T' comme étant le résultat d'un étirement isotrope ou d'une compression isotrope, isotrope signifiant identique dans toutes les directions. Alors T' ∈ U. Donc U est fermé sous les opérations d'étirement isotrope et de compression isotrope.

Cette propriété ne spécifie nullement comment l'étirement est effectué, ni comment sont spécifiés les nouveaux attributs chromatiques. Ces spécifications sont dépendantes du modèle mathématique utilisé pour U.

La façon dont on imagine le processus d'étirement va influer sur la manière de modéliser l'assignation de nouveaux attributs chromatiques à l'image étirée. Voyons quelques exemples.

Une façon de se représenter l'opération d'étirement est que l'image originale soit grossie à l'aide d'une loupe. Cette façon de voir réduit la luminosité de l'image. En effet, le nombre de photons émis par unité de surface de l'image d'origine est réparti sur une plus grande surface dans l'image étirée, ce qui donne le sentiment que l'image a perdu de sa coloration et de sa brillance.

Une autre façon de voir l'opération d'étirement nous est donnée par notre expérience quotidienne, lorsqu'on se rapproche d'un objet pour mieux le voir. En réduisant de moitié la distance qui nous sépare d'un objet, on multiplie par un facteur quatre la surface de l'image projetée sur la rétine ; cependant, le nombre de photons tombant sur la rétine est lui aussi quadruplé, si bien que la brillance des couleurs de l'image est conservée. Comme cette façon de voir l'opération d'étirement nous est coutumière, c'est celle que l'on aimerait adopter.

On peut envisager des situations moins simples que les deux exemples précédents. Supposons, par exemple, que la source lumineuse soit tellement éloignée qu'elle nous semble ponctuelle, cela pourrait être les feux de position d'un avion la nuit. Dans ce cas, le fait de diminuer d'un facteur deux la distance entre nous et cette source nous permet d'augmenter d'un facteur quatre la luminosité perçue, mais la surface de la source n'a pas changé, elle reste ponctuelle. Dans des cas similaires, nous utiliserons des mesures de Borel pour modéliser les attributs chromatiques de l'image.

On conclut de ces quelques considérations qu'il s'agit d'être prudent lorsqu'on définit l'opération d'étirement d'un modèle mathématique d'une image.

Propriété 6 : Soit $T \in U$. On construit T' comme le résultat de la réflexion de T par rapport à un axe parallèle à un des côtés de son support. Alors, $T' \in U$, c'est-à-dire que U est fermé sous l'opération de réflexion.

Propriété 7 : Soit $T \in U$. Construisons T' de la façon suivante : on délimite d'abord une zone rectangulaire dans T, on fait ensuite subir à ce rectangle une rotation pour amener ses cotés parallèles aux côtés du support de T. Alors, $T' \in U$, c'est-à-dire que U est fermé sous de telles opérations de rotation.

L'étirement et la compression isotrope d'images sont des exemples de **similitudes**. Les similitudes, les réflexions et les rotations sont des exemples de **transformations affines**. Nous en reparlerons au chapitre 4. Les propriétés 5, 6 et 7 peuvent se généraliser par la propriété suivante.

Propriété 8 : U est fermé sous l'application d'une transformation affine arbitraire, appliquée à un parallélogramme quelconque extrait de l'image, et telle qu'elle donne comme résultat une image rectangulaire.

Les propriétés que l'on vient de voir nous montrent qu'une image du monde réel donne lieu à toute une collection d'images qui en dérivent. Ceci nous amène à une définition d'une relation d'ordre sur U. Les définitions qui vont suivre ne prendront en considération que les propriétés 4 et 5, mais elles pourront évidemment être adaptées pour tenir compte des propriétés 5 et 6, ou 6 et 8, ou 8, ...

Définition : Soient S et T deux éléments de U. On dira que T est **dérivée** de S si et seulement si T peut être obtenue à partir de S par les opérations d'extraction et d'étirement (compression) isotrope, en utilisant les propriétés 4 et 5. T est dérivée de S se note $T < S$.

Définition : Soit $E \subset U$ un ensemble d'images du monde réel. On définit $W(E) \subset U$ par

$$W(E) = \{S \in U : S < T \text{ pour un } T \in E\}.$$

On dit que W(E) est l'ensemble des images du monde réel **générées** par E, en utilisant les propriétés 4 et 5.

Définition : Deux images S et T sont **équivalentes**, sous les propriétés 4 et 5, si et seulement si $S < T$ et $T < S$. On note cette équivalence $S \sim T$.

II. Modèles mathématiques

Nous allons, ici, donner des exemples de modèles mathématiques de U, l'ensemble des images du monde réel. Ils seront utiles pour travailler en pratique avec des images. Bien que ces modèles fassent appel à des notions détaillées plus loin (chapitre 4), comme les mesures ou les transformations affines, nous pensons qu'ils sont suffisamment intuitifs pour que l'on puisse se les représenter.

A. Premier modèle

Le premier modèle de U, noté U_1, ne va s'occuper que des images en deux niveaux de couleurs. Soit T un élément de U_1, représenté par son support S et une fonction caractéristique (ou d'appartenance) $\chi_A : S \rightarrow \{0, 1\}$ d'un sous ensemble mesurable de Borel $A \subset S$. A « plaqué » sur S donne une figure noire sur un fond blanc. Au noir est associée la valeur un, tandis qu'au blanc est associée la valeur zéro. χ_A va représenter les attributs chromatiques de l'image, elle est définie par

$$\chi_A(x) = \begin{cases} 1 \text{ si } x \in A \\ 0 \text{ si } x \notin A \end{cases}.$$

On impose à A et χ_A d'être des mesures de Borel pour pouvoir mesurer des surfaces sur des sous-ensembles mesurables de l'image, afin de permettre de définir une approximation de résolution finie de l'image, comme le mentionne la propriété 3. Par exemple, on réalise une approximation digitale de résolution arbitraire aussi grande que voulue de l'image T en affectant la valeur 0 ou 1 au pixel $P \subset S$ selon que le nombre

$\frac{\int_P \chi_A(x)\,dx}{\int_P dx}$ est inférieur à 0,5 ou plus grand ou égal à 0,5. On note l'approximation de résolution finie de A, $\tilde{A}$. On écrit aussi $\tilde{A} = P_{m\times n}(A)$, si m et n sont respectivement le nombre de pixels horizontaux et verticaux de S. $P_{m\times n}$ est une projection de U_1 dans lui-même, car $P_{m\times n}(P_{m\times n}(A)) = P_{m\times n}(A)$ pour tout sous-ensemble $A \subset S$. Les modèles de résolution finie $\{ P_{m\times n}(A) : m, n = 1, 2, \ldots \}$ sont **consistants**. C'est-à-dire que, si l'on forme une approximation digitale suffisamment élevée de A, $P_{m'\times n'}(A)$, où m' est nettement supérieur à m et n' nettement supérieur à n, et si l'on réalise la même approximation pour $P_{m\times n}(A)$, on obtient quasiment le même résultat. Autrement dit, $d(P_{m'\times n'}(A), P_{m'\times n'}(P_{m\times n}(A)))$ tend vers 0 quand m' et n' tendent ensemble vers l'infini, où d est une métrique sur U_1, comme par exemple

$$d(A, B) = \int_{\Im} | \chi_A(x) - \chi_B(x) |\, dx,$$

pour comparer les images de même support.

On va considérer l'étirement (la compression) de T, d'attributs chromatiques $\chi_A(x)$, donnant une image $T' \in U_1$, comme une transformation de $\Im$, support de T, en $\Im'$, support de T', dont les dimensions physiques sont multipliées par un facteur α, supérieur à un pour un étirement, inférieur à un pour une compression. Alors, si l'on considère comme origine des coordonnées le coin inférieur gauche des supports, les attributs chromatiques de T' sont $\chi(x) = \chi_A(x/\alpha)$.

B. Deuxième modèle

Dans ce deuxième modèle, U est modélisé par U_2. Soit T un élément de U_2, représenté par son support $\Im$ et ses attributs chromatiques. Ceux-ci sont modélisés par un intervalle I de nombres réels, par exemple [0, 255], spécifiant les intensités possibles en niveaux de gris, et une fonction $f : S \rightarrow I$, telle que f(x, y) est la brillance du point situé en (x, y). On peut réaliser une approximation de résolution arbitraire aussi grande que l'on veut de l'image $T \in U_2$ en affectant la valeur $f(P) = \frac{\int_P f(x,y)\,dx\,dy}{\int_P dx\,dy}$ au pixel P de $\Im$.

Si on considère l'étirement ou la compression de T, d'attributs chromatiques f(x, y), donnant une image $T' \in U_2$, comme une transformation de $\Im \subset \Re^2$, support de T, en $\Im'$, support de T', dont les dimensions physiques sont multipliées par un facteur α, supérieur ou inférieur à un selon que l'opération est un étirement ou une compression, et si l'on considère comme origine des coordonnées le coin inférieur gauche des supports, les attributs chromatiques de T' sont $g(x, y) = f(x/\alpha, y/\alpha)$, pour tous les (x, y) de $\Im'$.

Dans U_2, la distance entre deux images I et H de même support peut être mesurée à l'aide d'une fonction distance correspondant à l'espace choisi, par exemple

$$d(I, H) = \int_{\Im} | f(z) - g(z) |\, dz,\ z = (x, y) \in \Im.$$

C. Troisième modèle

U, l'ensemble des images du monde réel, est modélisé par U_3. Une image est représentée par une mesure de Borel μ normalisée à valeurs réelles et de support $\Im$. La quantité totale de lumière émise par un sous-ensemble mesurable de Borel $A \subset \Im$ est $\int_A d\mu$. Ce modèle ne permet pas de décrire l'intensité lumineuse en un seul point du support. On peut réaliser une approximation de résolution arbitraire aussi grande que l'on veut de l'image $T \in U_3$ en affectant la valeur $f(P) = \int_P d\mu$ au pixel P de $\Im$.

Si on considère l'étirement ou la compression de T, d'attributs chromatiques donnés par $\mu(B)$, fournissant une image $T' \in U_3$, comme une transformation A de $\Im \subset \Re^2$, support de T, en $\Im'$, support de T', dont les dimensions physiques sont multipliées par un facteur constant, supérieur ou inférieur à un selon que l'opération est un étirement ou une compression, et si l'on considère comme origine des coordonnées le coin inférieur gauche des supports, les attributs chromatiques de T' sont fournis par $\nu(B) = \mu(A^{-1} B)$, où A^{-1} désigne la transformation affine de $\Re^2$ qui transforme $\Im'$ en $\Im$.

La distance entre deux images, I et H, peut se mesurer à l'aide d'une fonction distance telle que

$$d(I, H) = \int_{\Im} | d\mu(x) - d\nu(x) |.$$

D. Quatrième modèle

Dans ce quatrième modèle, U est modélisé par U_4. Soit T un élément de U_4, représenté par son support $\Im$ et ses attributs chromatiques. Ceux-ci sont modélisés par un intervalle de nombres réels, par exemple [0, 255], spécifiant les intensités possibles, et trois fonctions $f_r : S \rightarrow I$, $f_v : S \rightarrow I$, $f_b : S \rightarrow I$, où $f_r(x, y)$, $f_v(x, y)$ et $f_b(x, y)$ fournissent respectivement l'intensité de la composante rouge, verte et bleue au point (x, y) du support de l'image.

Dans U_4, de même que pour U_2, la distance entre deux images I et H de même support peut être mesurée à l'aide d'une fonction distance correspondant à l'espace choisi, par exemple

$$d(I, H) = \int_{\Im} | f_r(z) - g_r(z) | dz + \int_{\Im} | f_v(z) - g_v(z) | dz + \int_{\Im} | f_b(z) - g_b(z) | dz,$$

où $z = (x, y) \in \Im$.

Le reste des idées est essentiellement le même que pour le modèle U_2.

Il existe, bien sûr, d'autres modèles, mais tous ont ceci en commun que l'entité mathématique sous-jacente pour modéliser le support est d'un caractère infiniment détaillé : toute région, aussi minuscule soit elle, est associée à des nombres qui décrivent ses caractéristiques optiques. L'étape suivante devrait permettre de transformer notre image en chaîne de zéro et de un.

III. Digitalisation, résolution et quantification

Il est souvent pratique de considérer les données comme des fonctions. Par exemple, le son est représenté comme une amplitude qui évolue en fonction de la variable temps, une image statique a comme variable l'espace, tandis que les images animées (vidéos) ont comme variables libres l'espace et le temps. On sépare généralement ces données en deux catégories : les **données analogiques** sont telles que leur domaine et leur ensemble d'arrivée sont des intervalles de nombres réels, les **données digitales**, par contre, possèdent un domaine et un ensemble d'arrivée qui sont des ensembles finis de valeurs possibles, généralement des entiers (figure 2.3).

Un **scanner (digitaliseur)** est un dispositif qui permet de transformer des images analogiques, ou supposées comme telles, en images digitales. L'image est échantillonnée par une grille de pixels, ces pixels sont détectés par des cellules CCD (Charge Coupled Device) qui transforment les attributs chromatiques des régions de la taille d'un pixel en charges électriques, qui, après traitement dans un convertisseur analogique-digital, sont transformées en nombres réels tronqués.

Les données analogiques renferment des informations en nombre potentiellement infini : le nombre de couleurs existant dans la nature est fixé par le nombre de longueurs d'ondes situées dans l'intervalle de longueurs d'ondes décelables par l'œil humain, entre plus ou moins 0,4 et 0,8 microns, c'est-à-dire un nombre infini, alors que les graphiques sur ordinateurs ne supportent au maximum que 16.777.216 couleurs différentes ; les pixels apparaissant lors d'une digitalisation ont des dimensions finies, alors que les images du monde réel offrent une continuité spatiale. Les données digitales introduisent donc des erreurs.

Prenons l'exemple d'un support CD audio. Le son n'est pas représenté de façon continue, seule une fraction du son original y est inscrit, le signal sonore est échantillonné à 44,1 KHz, ce que l'on appelle le **taux d'échantillonnage** : cela signifie que l'on n'inscrit sur le CD que l'amplitude du son mesurée 44100 fois par seconde. Mais, l'amplitude aussi est discrétisée, elle ne peut prendre que certaines valeurs. Pour un CD, ce que l'on nomme la **résolution d'échantillonnage** est de 16 bits : le son peut prendre 65536 amplitudes possibles.

Dans le domaine du traitement d'images, on parle de **résolution**, exprimée en bits par pixel ou en DPI (Dots Per Inch), et de **quantification**, exprimée en nombre de couleurs ou en nombre de bits. Ainsi, on rencontre des images 640 x 480 x 256 (résolution horizontale x résolution verticale x nombre de couleurs), 1280 x 1024 en vraies couleurs (24 bits), 320 x 200 en un bit (noir et blanc), ... Les erreurs introduites, dues au quadrillage en pixels et à la discrétisation des couleurs, s'appellent respectivement les **erreurs d'échantillonnage** et les **erreurs de quantification**.

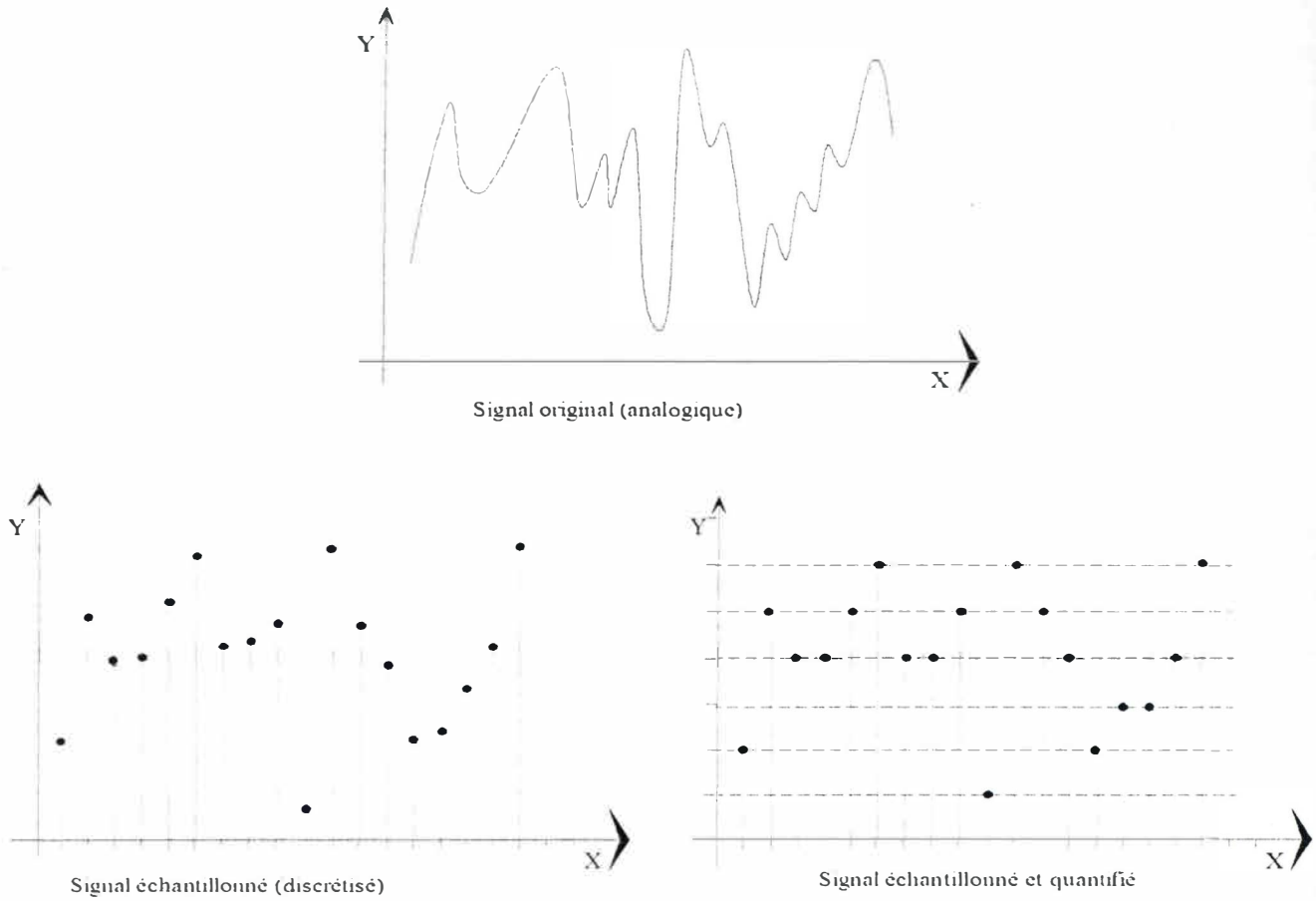


figure 2.3 : discrétisation et quantification

Si les données, images, sons, vidéos, ainsi stockées contiennent tant d'erreurs, pourquoi les utilise-t-on si intensément, alors que la technologie nous permet aujourd'hui des enregistrements analogiques quasi parfaits (les « masters » des enregistrements musicaux sont analogiques) ? La réponse est que le digital est plus facile à stocker, à transmettre et à manipuler. Les télécommunications prenant de plus en plus d'importance dans notre quotidien, il est logique d'utiliser des données digitales, et non pas analogiques, pour véhiculer sur les canaux de communication : d'une part, la transmission de données digitales s'effectue avec une très haute fidélité, grâce aux codes autocorrecteurs d'erreurs, d'autre part, la copie de données digitales se déroule sans altération, contrairement aux données analogiques.

Il est possible de se donner une idée de l'impact des erreurs d'échantillonnage. Une façon de procéder est de se demander quel est le degré de difficulté à interpoler les points manquants. Supposons, par exemple, que la valeur que prend une fonction inconnue f(x) soit donnée pour une suite de points séparés par une égale distance : $y_n = f(n.\Delta x)$. On désire interpoler la fonction f par une autre fonction : $f(x) = \cos(\omega.x)$, avec ω à déterminer. Un problème surgit alors, du fait que le ω solution n'est pas unique. En effet, on peut montrer que si ω est solution, alors il en est de même pour $\omega + 2\pi m / \Delta x$, pour tout m entier. Si on définit $f_e = 2\pi / \Delta x$, **appelée fréquence d'échantillonnage**, on voit de suite que plus cette fréquence d'échantillonnage est élevée, autrement dit plus l'**intervalle d'échantillonnage** Δx est petit, meilleure sera la reconstruction de la fonction f. Puisque la solution ω est périodique de période f_e, on peut restreindre ω sur l'intervalle compris entre $-f_e/2$ et $f_e/2$, pour ne conserver qu'une solution ω et une fonction d'interpolation. A l'inverse, si l'on sait que |

ω | est inférieur à un valeur f_{max}, on doit prendre une fréquence d'échantillonnage au moins égale à $2.f_{max}$; cette valeur de $2.f_{max}$ est appelée la **fréquence de Nyquist**.

De façon plus générale, d'après les travaux du mathématicien français Joseph Fourier, toute fonction f(x) peut se représenter par sa transformée de Fourier f'(ω) :

$$f(x) = \int_{-\infty}^{+\infty} f'(\omega) \cos(\omega . x)\, dx$$

Si on trace le graphe du carré de l'amplitude de f'(ω) en fonction de ω, on obtient le **spectre de puissance** P(ω) de la fonction f. Si il existe une fréquence ω_M, appelée **fréquence de coupure**, telle que P(ω) = 0 pour toute fréquence ω telle que | ω | > ω_M, on dit que le signal possède une **limite de bande**. Le théorème de Nyquist montre alors que si un signal possédant une limite de bande est échantillonné à une fréquence supérieure ou égale à $2.\omega_M$, le signal peut être reconstitué de façon unique à partir des valeurs échantillonnées. Si le signal ne possède pas de limite de bande, on s'en tire en filtrant le signal : on en supprime les composants de fréquences supérieures à la moitié de la fréquence d'échantillonnage, avant d'échantillonner les données.

IV. Intensités

Pour discrétiser l'intensité lumineuse de chacun des pixels de l'image, on pourrait « saucissonner » l'intensité lumineuse en intervalles égaux entre l'intensité minimale et l'intensité maximale de l'image. Cette pratique est hélas trop simple, car, en pratique, à deux couples de niveaux d'intensités lumineuses également séparées ne correspondent pas deux couples de niveaux de brillance perçue à égale distance ; la réponse de l'œil humain à la lumière ne se fait pas selon une échelle linéaire, mais se comporte plus ou moins de manière logarithmique. On doit effectuer la quantification de telle façon que tout niveau d'intensité maintienne un taux de croissance constant avec le niveau précédent. Par exemple, si on décide de 256 niveaux pour la quantification, I_0 représentant la plus petite intensité de l'image, on prend les 255 autres intensités à partir de I_0 : $I_n = (1,01)^n.I_0$, n allant de 0 à 255. Les pixels les plus brillants sont à peu près douze fois plus intenses que les moins brillants : $(1,01)^{255} \cong 12,6$.

V. Couleurs

Les différentes recherches portant sur la désignation des couleurs en informatique ont donné naissance à plusieurs modèles de représentation des couleurs. Certains sont basés sur la théorie trichromatique des couleurs, et sont issus des travaux de la Commission Internationale de l'Eclairage. Il s'agit des modèles RGB, CMY et YUV, que nous verrons brièvement. D'autres modèles sont basés sur la perception subjective des couleurs. Ils sont basés, en partie, sur les travaux du peintre américain Albert Munsell. Il s'agit des modèles HSL, HCL, HSV et HSB.

A. Modèle (R, G, B) ou (R, V, B)

Le système (R, G, B), pour Red, Green, Blue, ou, en français (R, V, B), pour Rouge, Vert, Bleu, est le plus répandu, car il correspond au fonctionnement des moniteurs couleur. Il définit une couleur par l'addition des trois couleurs primaires : rouge, vert et bleu, selon la synthèse additive.

Le système (R, V, B) est un système à trois dimensions, qui peut être représenté sous la forme d'un cube dont chaque axe correspond à une couleur primaire (figure 2.4). Une couleur particulière est ainsi spécifiée en indiquant les contributions de chaque couleur primaire. Généralement, celles-ci oscillent, en pourcentage, entre 0% et 100%, et en valeur entre 0 et 255. Dans le cube, la diagonale principale, où les trois couleurs primaires contribuent de la même façon à la couleur particulière, représente les niveaux de gris, depuis le noir jusqu'au blanc.

B. Modèle (C, M, Y)

Le cyan, le magenta et le jaune sont respectivement les couleurs complémentaires du rouge, du vert et du bleu. Ils sont appelés couleurs primaires soustractives car leur effet est de soustraire certaines couleurs de la lumière blanche.

D'une manière générale, il n'existe guère de différence entre les systèmes (R, G, B) et (C, M, Y). Ils sont en effet établis tous les deux à partir de trois composantes fondamentales. Le système (C, M, Y) est également représenté sous la forme d'un cube, mais cette fois l'origine est le blanc et les trois axes principaux le cyan, le magenta et le jaune (figure 2.4). La relation entre les deux modèles peut s'exprimer simplement par la relation matricielle :

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix}.$$

Ce procédé est également important dans la pratique, car tous les procédés d'impression sur papier font appel à la synthèse soustractive.

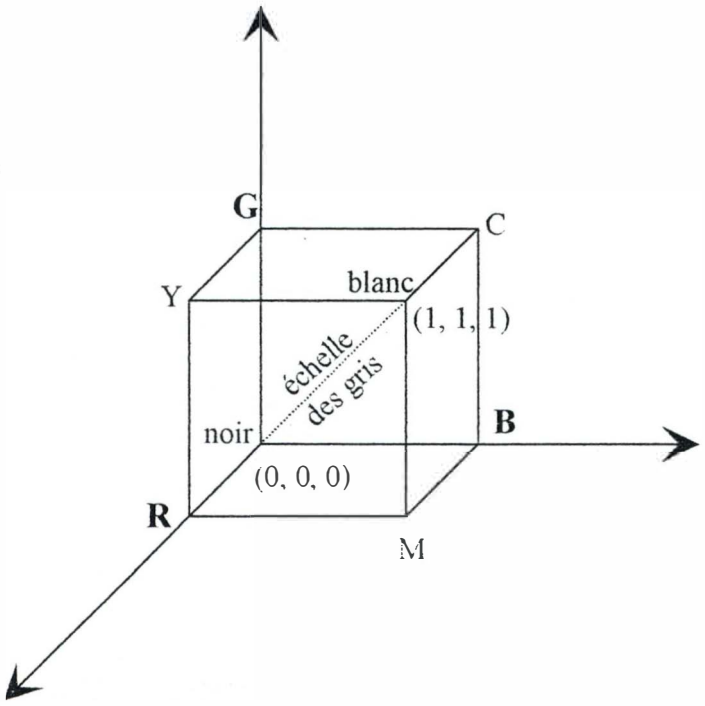


figure 2.4 : cube des couleurs

C. Modèle (Y, U, V) ou (Y, Cr, Cb)

En télévision couleur, par compatibilité avec les différents systèmes de codage vidéo : NTSC, SECAM et PAL, on préfère au système (R, V, B) le système (Y, U, V) = (Y, (R - Y) / 2.03, (B - Y) / 1.14), encore noté (Y, Cr, Cb) et défini matriciellement par :

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ -0{,}148 & -0{,}289 & 0{,}437 \\ 0{,}615 & -0{,}515 & -0{,}1 \end{pmatrix} \begin{pmatrix} R \\ V \\ B \end{pmatrix}.$$

Le signal monochrome Y se nomme **luminance** : c'est une pondération des trois signaux (R, V, B) correspondant à la sensibilité de l'oeil humain. Les deux autres signaux donnent les informations de couleur : ce sont les **chrominances** bleues et rouges.

VI. Lecture de l'image

Après avoir échantillonné et quantifié l'image, on obtient un tableau de valeurs. Or, on désire un flot de bits en sortie. On doit donc se donner un sens de parcours pour la lecture du tableau. Généralement, dans un but non caché de compression ultérieure, on opte pour un sens de lecture qui minimise la variation de données : il faut que le parcours de lecture reflète la géométrie de l'image, les pixels proches dans l'image doivent se retrouver proches dans le flot de sortie. On décrit, ici, quelques choix possibles pour le sens de lecture.

- Un premier choix est celui correspondant à l'ordre de parcours des canons à électrons sur un moniteur non entrelacé. Les pixels sont lus de gauche à droite et de haut en bas (parfois de bas en haut, par exemple les fichiers bitmap BMP). A la fin de chaque ligne, deux

pixels se retrouvent adjacents dans la lecture, alors qu'ils n'étaient pas adjacents dans l'image.

- Une façon de rompre la discontinuité de proximité des pixels, qui surgit en fin de chaque ligne, est de parcourir alternativement le tableau de gauche à droite pour les lignes de numéro impair, et de droite à gauche pour les lignes de numéro pair ; les points adjacents dans l'image sont alors aussi adjacents dans la séquence de lecture. On remarque, pour les deux possibilités que l'on vient de décrire, qu'aucune corrélation verticale n'existe entre les pixels.

- Dans cette troisième option, on essaie de préserver autant que possible, pour la lecture, l'information bi-dimensionnelle de l'image. Ce parcours est connu sous le nom de **courbe de Hilbert** (figure 2.5). Comme dans le choix précédent, les pixels adjacents dans l'image sont adjacents dans la séquence de lecture, mais cette fois, il n'existe plus de direction privilégiée. Cet ordre n'est uniquement défini que lorsque les dimensions de l'image sont des puissances de deux. Cette courbe possède la propriété que le premier quart de l'image (en haut à gauche) correspond au premier quart de pixels dans la séquence de lecture, le deuxième quart de l'image (en haut à droite) correspond au deuxième quart de pixels dans la séquence de lecture, ..., le premier seizième de l'image correspond au premier seizième de pixels dans la séquence de lecture, et ainsi de suite.

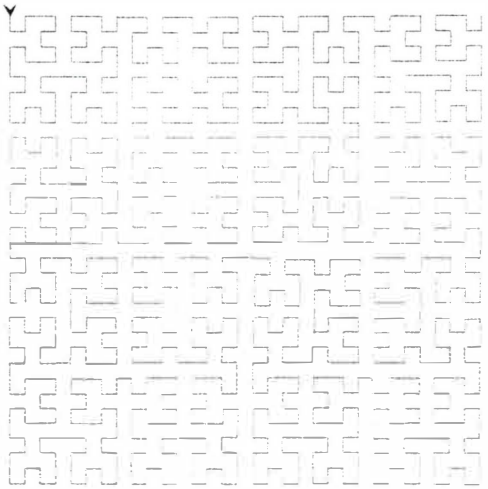


figure 2.5 : courbe de Hilbert

Chapitre 3 : Généralités sur les techniques de compression

Ce chapitre détaille les concepts de base liés aux techniques de compression et décrit quelques-uns des algorithmes le plus souvent rencontrés.

I. Techniques de compression de données

A. Introduction

D'après les travaux de l'ingénieur Claude Shannon, spécialiste de la théorie de l'information, on peut voir l'**information** comme ce qui est nouveau, non déductible d'un certain contexte. Par exemple, lors des réceptions qu'il organisait, Shannon s'amusait à faire deviner, lettre par lettre, les phrases d'un paragraphe d'un livre, donnant chaque fois la solution après avoir pris note des réponses de ses invités ; il remarqua que la proportion de bonnes réponses était de 75%, et en déduisit que la langue anglaise était aux trois quarts redondante, c'est-à-dire que trois lettres sur quatre étaient inutiles puisque les invités pouvaient reconstituer le texte en entier avec seulement une lettre sur quatre. On peut donc affirmer que la mesure de l'information dans les expériences de Shannon est de 1/4, le reste est qualifié de redondant. La redondance permet d'envelopper un message de certaines réitérations, lui permettant ainsi d'être transmis, perçu et reconnu malgré la présence de perturbations. La redondance se traduit par la surabondance de certaines probabilités, comme celles des voyelles dans une langue latine.

Compresser, c'est ne garder que l'essentiel, résumer l'information, éliminer le superflu. Un **algorithme de compression** a pour but de détruire des redondances d'informations, c'est-à-dire des répétitions surabondantes de certaines séquences de symboles. On dit d'une séquence de symboles qu'elle est **redondante** si sa probabilité d'occurence, au sein du fichier, est supérieure à sa probabilité d'occurrence lors d'un tirage aléatoire. La plupart des algorithmes de compression recherchent puis tentent de condenser des séquences de symboles (caractères ou groupes de bits) qui se répètent souvent. En effet, la majorité des données traitées par l'homme ont une structure organisée, de l'ordre, dont une des caractéristiques est la distribution non uniforme de certaines séquences de symboles.

La comparaison de l'efficacité des différentes méthodes de compression de données requiert une mesure du **taux de compression** d'un fichier traité par un algorithme donné. L'usage courant nous amène à donner deux définitions du taux de compression :

taux = 100 x taille du fichier compressé / taille du fichier original,

taux = taille du fichier original / taille du fichier compressé.

Ainsi, un fichier dont la taille avant compression est de 317 Ko, et qui voit sa taille réduite à 126 Ko après compactage, a un taux de compression de 39,7 % ou encore de 2,5 : 1. On rencontre parfois une troisième définition, qui est le « complément » de la première définition, qui donne une idée du gain de place de stockage engendré par la compression. Un taux de 0 % signifie pas de compression. Cette définition est la suivante :

taux = 100 x (1 - taille du fichier compressé / taille du fichier original).

Ainsi, avec notre petit exemple précédent, le taux de compression est de 60,3 %. Par convention, lorsqu'un algorithme de compression appliqué à un fichier donne en sortie un fichier dont la taille est supérieure à l'original (cela existe, voir plus loin : existence de fichiers non compactables), on dit que le taux de compression est de 100 %. D'ailleurs, les logiciels de compression (arj, dix, pkzip, uc2, lharc, compress, ...) gardent le fichier original si celui-ci est non compactable.

B. Classification

On classe les algorithmes de compression selon plusieurs critères :

- Certains algorithmes conservent l'entièreté de l'information, et donnent lieu à ce que l'on nomme une **compression sans perte** (algorithmes statistiques, à base de dictionnaires, ...) D'autres ne conservent qu'une partie significative de l'information, donnant lieu à ce que l'on nomme une **compression avec perte** : les images et les sons sont couramment compactés avec de grandes pertes volumiques d'information, mais avec relativement peu de pertes visuelles ou auditives.

- On distingue ensuite les modes de compression logique et physique des données. La **compression logique**, non algorithmique, consiste à ne coder les données que sur le nombre de bits nécessaires : les lettres de l'alphabet sont codées sur cinq bits au lieu de huit, les chiffres sur quatre bits, ... La **compression physique** consiste à utiliser des algorithmes qui recherchent et réduisent les séquences redondantes, au sein même des données, qu'elles soient déjà compressées logiquement ou non.

- On recense aussi les algorithmes qui travaillent au **niveau statistique** et ceux qui opèrent au **niveau numérique**. Pour les premiers, la valeur des motifs (chaînes de caractères, séquences de bits, ...) ne compte pas, ce sont les probabilités qui comptent, et le résultat est inchangé par substitution des motifs dans le fichier d'origine. Pour les seconds, les valeurs des motifs influent sur la compression (JPEG, ...), et les substitutions sont interdites.

- Il y a enfin les algorithmes dont on connaît à peu près l'efficacité, parce qu'on sait la mesurer simplement (algorithmes statistiques, ...), et d'autres **heuristiques**, qu'il faut exécuter complètement pour en connaître l'efficacité (LZW, ...).

C. Propriétés

EXISTENCE D'UN DECOMPACTEUR ASSOCIE

On pourrait considérer un compilateur comme un décompacteur : à une instruction d'un langage de haut niveau correspondent plusieurs instructions machine. Mais, outre le fait que le temps de décompression peut être très long, le procédé présente l'inconvénient majeur de ne fonctionner que dans le sens de la décompression : retrouver un code source à partir du fichier exécutable est une tâche très difficile. Cette constatation oblige, pour plus de clarté dans la suite de l'exposé, à parler de la notion de compression à partir de la donnée d'un **couple compacteur / décompacteur**, permettant les opérations de compression de façon symétrique. Le compacteur devient dès lors un compacteur bijectif.

EXISTENCE DE FICHIERS NON COMPACTABLES

Quels que soient le compacteur bijectif C et l'entier n, il existe au moins un fichier de taille n non compactable par C. En effet, le nombre de fichiers de taille n bits est 2^n. Supposons que chaque fichier de taille n soit compactable par C. La taille maximale d'un fichier compacté étant de n-1, le nombre de fichiers compactés possibles est $2 + 2^2 + 2^3 + ... + 2^{n-1} = 2^n - 2 < 2^n$. Ce qui signifierait que deux fichiers de longueur n, au moins, correspondent au même fichier compacté, ce qui est impossible en vertu de la bijectivité du compacteur C. De façon plus expéditive, il suffit de remarquer que si tout fichier était compactable, il suffirait d'itérer un nombre fini de fois l'algorithme C pour obtenir un bit, quel que soit le fichier : ce qui est tout aussi impossible, du point de vue de la théorie de l'information...

II. Compression à code minimal (codage statistique)

On va décrire ici quelques-uns des algorithmes statistiques de compression de données, parce qu'ils s'appliquent à n'importe quels types de fichiers, contrairement à la compression fractale, qui, jusqu'à présent, ne s'adapte qu'à la compression d'images. En particulier, ce type d'algorithme pourra être utilisé en dernière étape lors de la compression fractale, comme on le verra plus loin.

A. L'algorithme de base de Huffman

C'est en 1952 que D. A. Huffman publie « A Method for the Construction of Minimum Redundancy Code », qui sera la référence en matière de compression durant 25 ans. A partir d'une analyse statistique des symboles composant le fichier, cette méthode permet de compresser logiquement et physiquement les données sans perte. Elle consiste à attribuer aux symboles des codes binaires de différentes tailles, selon leurs proportions respectives ; plus un symbole est présent au sein d'un fichier, plus son code calculé selon la méthode de Huffman est court. Inversement, les symboles larges se voient attribuer des codes binaires larges. Plutôt que de longues démonstrations, voyons cela sur un exemple.

On va considérer la chaîne de 15 caractères EBCADDBACCDAEDF, choisis dans l'alphabet de six lettres (A, B, C, D, E, F). Si on désigne par f(i) la fréquence d'apparition du symbole i, la répartition des fréquences des lettres dans la chaîne est la suivante : f(A) = 3, f(B) = 2, f(C) = 3, f(D) = 4, f(E) = 2, f(F) = 1. Il suffit alors de suivre trois étapes pour appliquer l'algorithme de Huffman :

- étape 1 : classer les symboles dans l'ordre décroissant des fréquences d'occurrence. Cet ordre ne change pas si on remplace les fréquences par les probabilités $p(i) = f(i) / \Sigma f(i)$.

- étape 2 : regrouper séquentiellement les paires de symboles de plus faible probabilité, en reclassant si nécessaire. Plus précisément, calculer $s = f(i_n) + f(i_{n-1})$ la somme des deux plus faibles fréquences. Choisir le plus petit indice k tel que s soit supérieur ou égal à $f(i_k)$. Si $s = f(i_k)$, remplacer k par k+1. Recomposer la table de fréquences en plaçant à la k-ième position la valeur s et en décalant les autres d'une position vers le bas, puis décrémenter n d'une unité. Pousuivre jusqu'à ce que la table des fréquences ne comporte plus que deux éléments.

- étape 3 : coder avec retour arrière depuis le dernier groupe, en ajoutant un 0 ou un 1 pour différencier les symboles préalablement regroupés.

La succession de ces étapes nous permet de constituer un arbre de Huffman, qui nous donnera le code binaire à associer à chaque symbole. Le procédé est illustré par la figure 3.1.

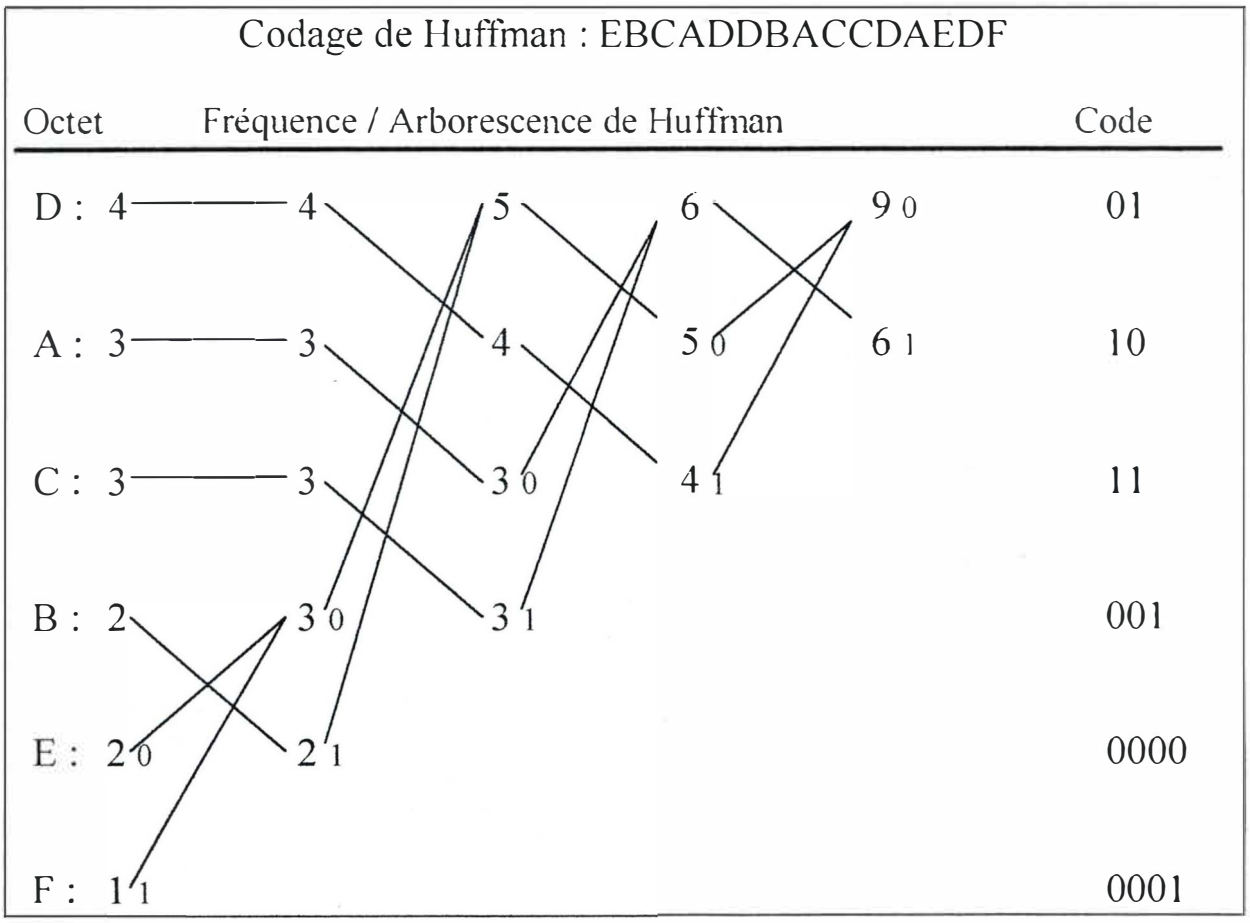


figure 3.1 : codage de Huffman

La taille des données compactées est donc : 4.2 + 3.2 + 3.2 + 2.3 + 2.4 + 1.4 = 38 bits. Si l'on avait adopté un codage tel que chaque lettre soit codée sur trois bits (puisqu'il y a six possibilités), on aurait obtenu 15.3 = 45 bits, c'est-à-dire sept bits en plus. On remarquera que la compression ne dépend pas de l'ordre des symboles, mais seulement de leur répartition statistique.

Si l'on choisit de façon quelconque une série de motifs binaires de différentes tailles, et qu'on les associe, on constate en général qu'il existe plusieurs façons de les dissocier pour obtenir une série originale, puisqu'on peut confondre la terminaison des uns avec le début des autres. Ainsi, en prenant les motifs 01, 11, 011, 10, on peut former la suite 011111011 ; et il y a plusieurs façons de relire les motifs de cette suite, entre autres : 01 | 11 | 11 | 011 et 011 | 11 | 10 | 11. Par contre, les codes binaires résultant de l'algorithme de Huffman possèdent la propriété de **séparabilité**, qui assure la bijectivité du codage. En lisant le fichier compacté bit après bit, on ne peut confondre deux motifs différents, car ils ont chacun un préfixe unique. On peut vérifier que la séquence 0000 001 11 10 01 01 001 10 11 11 01 10 0000 01 0001, qui est la chaîne de départ compressée, donne une relecture unique.

Une particularité des algorithmes statistiques est que, comme les codes séparables ne sont pas connus d'office, le décompacteur doit posséder la table des fréquences pour pouvoir retrouver les octets du fichier d'origine. Cette table est en général sauvée en début de fichier compacté, et s'appelle **en-tête de l'algorithme statistique**.

Des variantes de cet algorithme de Huffman de base permettent d'améliorer le taux de compression, notamment en prenant en compte l'ordre des symboles ou en codant judicieusement l'en-tête. Il existe également ce que l'on appelle des variantes dynamiques ou adaptatives, qui partent d'un table où toutes les fréquences sont nulles et qui sont mises à jour à chaque lecture d'un octet. Ainsi, le code binaire émis pour un octet est fonction des octets lus depuis le début du fichier et correspond au meilleur code binaire de la table pouvant être émis pour cette distribution. On parle d'**algorithme dynamique** ou **adaptatif**, puisque le classement des octets est susceptible de varier à chaque nouvelle lecture d'un octet dans le fichier. Le grand avantage du mode adaptatif, pour un algorithme statistique, est de ne pas nécessiter de prélecture des données pour l'établissement de la table des fréquences.

B. L'algorithme de Shannon-Fano

Il s'agit d'un algorithme concurrent de Huffman, qui donne des résultats quasi équivalents. Le procédé est cependant différent : alors que Huffman élabore une arborescence montante, à partir du bas de l'arbre, le procédé de Shannon-Fano construit un arbre descendant à partir de la racine, par divisions successives. Le seul point commun avec Huffman est le classement des fréquences par ordre décroissant, ce qui suppose, comme précédemment, une première lecture du fichier et la sauvegarde d'un en-tête.

L'algorithme se décompose en quatre étapes :

- étape 1 : classer les symboles dans l'ordre décroissant des fréquences d'occurrence.

- étape 2 : partitionner la table des fréquences en deux sous-tables de fréquences proches. Poursuivre l'arborescence jusqu'à ce que toute fréquence soit isolée.

- étape 3 : attribuer dans l'arborescence le bit 0 à chaque première sous-table.
- étape 4 : attribuer aux symboles les codes binaires correspondant aux bits de description de l'arborescence.

L'application de l'algorithme de Shannon-Fano à l'exemple utilisé pour montrer l'algorithme de Huffman est illustrée par la figure 3.2.

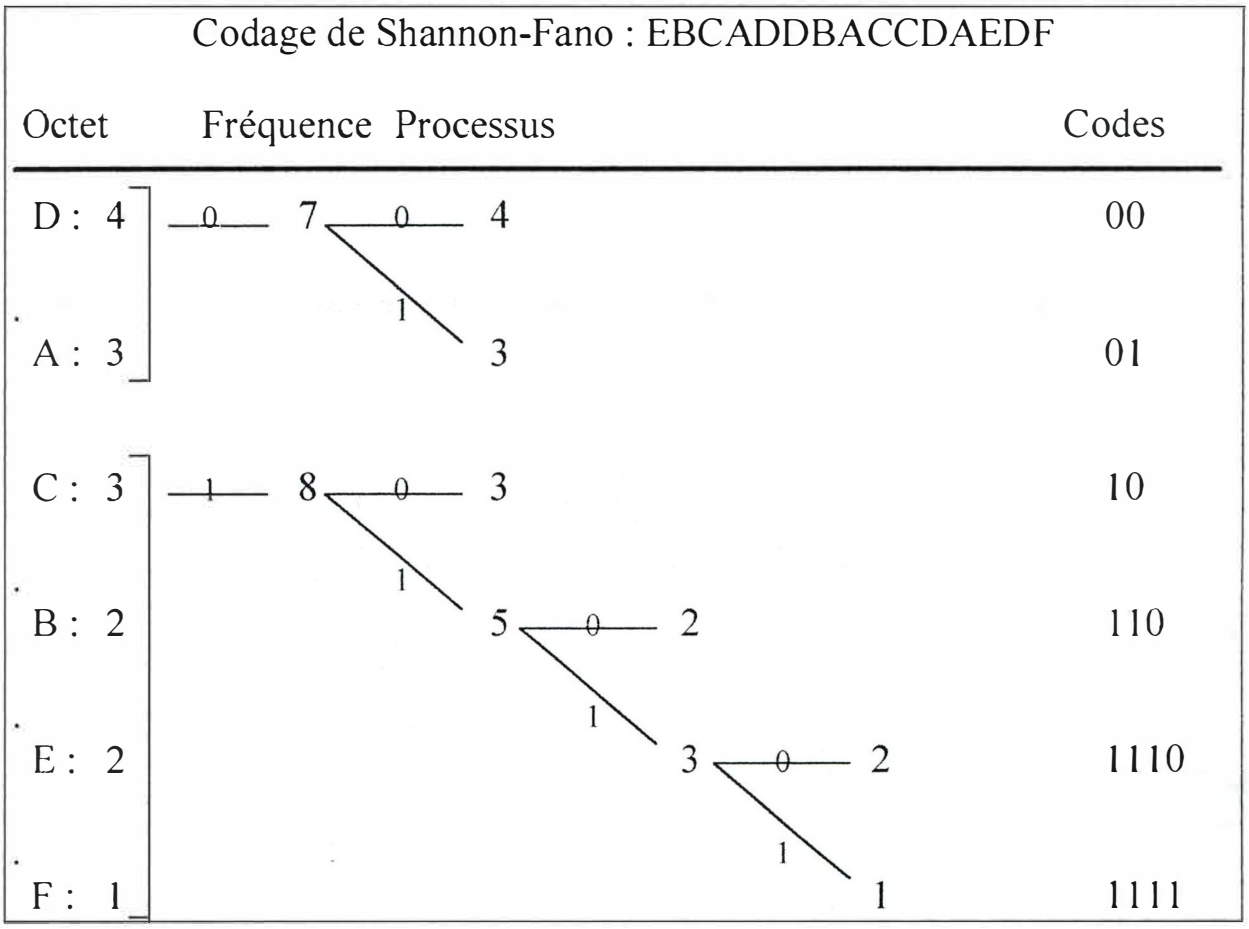


figure 3.2 : codage de Shannon-Fano

Si les fréquences des octets ont un écart-type important, il sera plus intéressant d'utiliser l'algorithme de Shannon-Fano. Dans le cas contraire où il y a peu de dispersion, l'algorithme de Huffman donnera un résultat légèrement meilleur.

C. Compression arithmétique

Contrairement aux algorithmes de Huffman et Shannon-Fano qui associent à des symboles des motifs binaires dont la taille dépend de leur distribution, le codeur arithmétique traite le fichier dans son ensemble, en lui associant un unique nombre décimal rationnel, qui est le **compacté** du fichier. Ce nombre, compris entre 0 et 1, possède d'autant moins de chiffres après la virgule que le fichier dont il est issu est redondant. Ces chiffres décimaux

dépendent non seulement des symboles du fichier dans l'ordre où ils apparaissent, mais également de leur distribution statistique.

A défaut d'être rapide, le codeur arithmétique en mode adaptatif, c'est-à-dire en compression au vol, est très puissant. Le Q-Coding développé par les chercheurs d'IBM représente une version adaptative très puissante du codage arithmétique, nettement plus puissante que l'algorithme de Huffman, particulièrement pour les images binaires de type fax.

On calcule d'abord, comme déjà vu, la table des probabilités des symboles du fichier. Il n'est pas nécessaire de les ordonner tant que le compacteur et le décompacteur utilisent le même ordre de parcours de cette table. On établit ensuite dans une autre colonne la statistique des probabilités cumulées, pour chaque symbole. Par conséquent, à tout symbole est associé un intervalle [a, b] unique dans [0, 1]. Ensuite, on applique les procédures algorithmiques suivantes.

- Pour le compresseur :

```
Au départ, m = 0 et M = 1 ;
Tant que le fichier n'est pas totalement lu, faire
{
  i = octet suivant ;
  Soit [a_i, b_i] l'intervalle associé à i ;
  s ← M - m ;
  M ← m + s.b_i ;
  m ← m + s.a_i ;
} ;
Renvoyer la valeur m qui est le compacté du fichier ;
```

- Pour le décompresseur :

```
N = nombre codé ;
Faire
{
  Trouver l'octet i tel que N ∈ [a_i, b_i] ;
  Emettre l'octet i ;
  s ← b_i - a_i ;
  N ← N - a_i ;
  N ← N / s ;
} Tant qu'il reste un octet à lire ;
```

Le décodage nécessite la connaissance du nombre de symboles codés, pour savoir où arrêter la décompression. En pratique, on utilise un symbole supplémentaire pour décrire la fin du fichier, ce qui évite le décompte et ne consomme que peu de bits de codage.

Malgré la difficulté de manipulations de fractions binaires, cet algorithme est très efficace : on peut obtenir des taux de compression comparables à ceux obtenus avec les méthodes de type « dictionnaire ».

III. Compression à base de dictionnaire

A. Introduction

La domination mondiale des algorithmes statistiques prend fin en 1978, lorsque deux mathématiciens israéliens, Abraham Lempel et Jacob Ziv, publient leur méthode de compression à base de dictionnaire. L'idée de départ des méthodes de type dictionnaire est de coder des grandes quantités de données à l'aide d'une base multi-indexée. On bâtit des listes de séquences redondantes du fichier : syllabes, mots, ... La compression est réalisée en émettant les adresses (index) des séquences dans les listes. Le problème, comme pour les algorithmes statistiques, provient du stockage du dictionnaire dans le fichier compressé, pour retrouver les données à partir des adresses, ce qui rend parfois la compression impossible.

Pour illustrer l'idée principale des algorithmes à base de dictionnaires, considérons la particularité importante qu'ont les langues de n'utiliser qu'un faible nombre de mots par rapport aux possibilités lexicographiques offertes par la mathématique combinatoire. Par exemple, le nombre de mots français de six lettres est de plus ou moins 9000, soit plus de vingt mille fois moins que le nombre de possibilités théoriques. Pour coder une lettre de l'alphabet, il faut n bits, où n est tel que $2^{n-1} < 26 \leq 2^n$, c'est-à-dire n = 5. Pour coder un mot de 6 lettres, on a donc besoin de 6 x 5 = 30 bits. Essayons d'utiliser une compression statistique sur ces mots. Pour cela, il suffit de posséder le dictionnaire des mots de 6 lettres et de renvoyer l'index, la position dans le dictionnaire, du mot à coder. On vérifie que le nombre m de bits tel que $2^{m-1} < 9000 \leq 2^m$ est de 14. On code donc un mot sur 14 bits et non plus sur 30, d'où un taux de compression de plus de 50 %.

Les algorithmes à base de dictinnaires (LZ78, LZSS, LZW, ...) sont tous des variantes (tampons circulaires, réseaux neuronaux, ...) de l'algorithme original LZ77. Nous en décrirons un, le plus répandu.

B. Algorithme LZW

L'algorithme de compression / décompression LZW porte le nom de ses trois inventeurs : Lempel et Ziv, qui l'ont conçu en 1977, et Welch qui l'a amélioré en 1984, avec dépôt de brevet. C'est un algorithme nettement plus performant que les algorithmes statistiques, permettant d'obtenir des gains élevés sans les inconvénients des tables et des dictionnaires préétablis. Il sert de base à la norme V42bis du CCITT pour la compression des données des modems. L'algorithme LZW se distingue des méthodes statistiques sur plusieurs points :

- d'abord parce que le dictionnaire n'a pas a être sauvé avec le fichier comprimé. Il est en effet reconstruit automatiquement à la décompression, en fonction des données du fichier comprimé, ce qui représente un net avantage sur les méthodes statistiques qui traînent souvent derrière elles des tables d'en-tête.

- ensuite parce qu'il représente un **algorithme d'apprentissage**, dans la mesure où les séquences répétitives de symboles sont détectées puis compressées lors de leur prochaine occurrence. Les méthodes statistiques ignorent cela, puisqu'elles travaillent uniquement avec les probabilités de présence des symboles.

- enfin parce qu'il permet le compactage « au vol » : n'ayant pas à lire le fichier au préalable, l'algorithme compresse des séquencces d'octets au fur et à mesure de leur lecture, ce qui est plus rapide.

COMPRESSION

Le dictionnaire, D, est un tableau dans lequel sont rangées des séquences de symboles de taille variable, repérées par leur adresse, c'est-à-dire leur position dans le tableau. Sa taille n'est pas fixe, il comprend en général de 2^8 à 2^{15} adresses. Les adresses 0 à 255 contiennent les codes des ocets de 0 à 255, les séquences de symboles ont des adresses supérieures à 255. Le fichier est lu octet par octet en appliquant l'algorithme suivant, où $\oplus$ est le symbole de concaténation :

```
s = premier octet du fichier ;
Tant que le fichier n'est pas entièremenr lu, faire
{
  t = octet suivant ;
  u ← s ⊕ t ;
  Si u ∈ D,  alors s ← u ;
             Sinon faire
             {
               Emettre adresse (s) ;
               Ajouter u dans D ;
               s ← t ;
             }
}
émettre adresse (s) ;
```

La taille des adresses n'est pas constante durant le processus ; on part avec une adresse sur 8 bits, et dès qu'une adresse à émettre dépasse 255, il faut la coder sur 9 bits. De manière générale, on incrémente de 1 bit la taille des adresses à émettre dès qu'on atteint une nouvelle puissance de 2.

Cet algorithme LZW ne paraît pas compliqué à programmer, à première vue seulement. Cependant, on doit prendre attention aux trois points délicats de sa programmation :

- la manipulation de chaînes de caractères dont la taille est inconnue à l'avance,
- la manipulation du dictionnaire, qui est un tableau de chaînes, dont la longueur est aussi inconnue d'avance,
- et comment savoir, le plus rapidement possible, si une séquence donnée est déjà présente dans le dictionnaire.

DECOMPRESSION

Le décompacteur est extrêmement rapide par rapport au compacteur. On lit les codes du fichier en partant d'une taille de 8 bits. Le principe du décompacteur est de reconstruire le dictionnaire au fur et à mesure du décompactage du fichier d'adresses :

```
a = premier caractère du fichier (8 bits) ;
Emettre (a) ;
Tant que le fichier n'est pas lu entièrement, faire
{
  b = lire_code_suivant ;
  Si b ∉ D, faire s = a ⊕ t ;
  Si b ∈ D, faire s = séquence (b) ;
  Emettre (s) ;
  t = premier caractère de s ;
  Ajouter a ⊕ t dans D ;
  a ← s ;
}
```

UTILISATION COMBINEE DES METHODES STATISTIQUES ET DICTIONNAIRE

On pourrait avoir l'idée de combiner les deux types d'algorithmes, statistique et dictionnaire, pour espérer récupérer la somme de leur efficacité : combiner la puissance de l'algorithme LZW pour repérer les séquences répétitives de taille variable, et l'efficacité optimale des algorithmes dynamiques à codage statistique pour compresser les codes d'adresses sur le nombre de bits qui correspond le mieux à leur fréquence d'apparition. C'est essentiellement cette démarche mixte qu'emploient la plupart des fameux compresseurs du domaine public. Les fichiers compactés requièrent tout de même la présence d'un en-tête, puisqu'ils utilisent une méthode statistique, mais l'emploi d'algorithmes dynamiques évite la lecture de l'entièreté du fichier ; le fichier est lu par blocs de taile fixe, et une méthode de compression adéquate est adoptée localement.

IV. Compression des images fixes

Le domaine de compression des images, de même que la compression des sons, est un champ d'investigation à lui seul. On pourrait recourir à des algorithmes déjà rencontrés pour compresser les fichiers images, mais c'est ne pas profiter du caractère bidimensionnel des images, ce ne seront donc pas ces méthodes qui fourniront les meilleurs taux de compression. De plus, la plupart des variations de couleurs d'une image comportant une très large palette ne sont pas perçues par l'oeil. Si, de plus, on tient compte du fait que la majorité des images possèdent très peu de points de détail, on admettra sans peine que l'on puisse supprimer des informations jugées non indispensables, puisque non perçues par l'oeil. On rencontre alors une compression avec perte ou **compression non conservative**.

Ce paragraphe va décrire, de façon très brève, les idées générales sur quelques algorithmes de compression propres aux images. Pour de plus amples détails ainsi que d'autres techniques récentes (ondelettes, laplaciens, ...), on pourra consulter [XM].

A. Compression par plans images

Le principe très intuitf de cette méthode consiste à séparer l'image en plans binaires. Par exemple, avec 256 niveaux de gris, on obtient 8 plans images correspondant aux 8 bits de définition du niveau de gris. La forte corrélation entre les pixels d'une image se retrouve essentiellement dans les plans images de bits de poids forts (128, 64,32,16). Pour ceux-ci, les algorithmes classiques pourront fournir de très bons taux de compression. Par contre, les plans correspondant aux bits faibles (8, 4, 2, 1) présentent une distribution assez aléatoire qui les rend très peu compactables. Pour les images en couleurs, on applique la méthode sur chacune des composantes.

B. Codage des plages (Run Length Encoding)

S'il est vrai que cette méthode ne présente que peu d'intérêt pour la compression des textes ordinaires, elle s'avère d'une redoutable efficacité lorsqu'il s'agit d'images binaires ou présentant peu de niveaux. Elle s'applique à des images contenant peu d'information, ou présentant des zones colorées très homogènes.

Le codage RLE consiste en une description de l'image ligne par ligne, ou colonne par colonne, des plages de pixels de couleur constante. Le codage d'une image binaire opère de la façon suivante :

- on émet le bit correspondant à la couleur du premier pixel.
- pour chaque segment de k pixels consécutifs de même couleur, on émet le nombre k, suivi d'un séparateur choisi pour ne pas être confondu.

Une variante à deux dimensions de l'algorithme existe également. Un exemple détaillé est fourni en annexe A, dans la description du format d'image BMP.

C. Compression par codage de Freeman

Cette méthode ne s'applique qu'aux images binaires dont les motifs sont des courbes (droites, arcs, ...) ou des surfaces connexes. Le principe consiste à choisir un point d'un tel motif, et à décrire, grâce au codage de Freeman, les pixels adjacents.

3	2	1
4		0
5	6	7

figure 3.3 : codage directionnel de Freeman

Cette technique est souvent utilisée pour compacter des images expérimentales (biologie, ...) ou des dessins comprenant une grande proportion de formes pleines.

Sur une trame carrée, le voisinage immédiat d'un pixel est une couronne de 8 pixels. Le codage de Freeman décrit la position d'un pixel adjacent par un entier de 0 à 7 (figure 3.3).

Pour coder une courbe de faible épaisseur, il suffit de se placer à une extrémité, et de coder successivement tous les points, en donnant leur direction par rapport au pixel précédent. Pour une forme pleine, on ne décrit que son contour.

D. JBIG

Il s'agit d'une norme de **codage progressif** sans perte d'une image binaire comprimée : c'est-à-dire qu'on envoie les données d'une résoluton réduite de l'image, de l'ordre de 10 à 15 points par pouce, puis on transmet les données comprimées additionnelles pour pouvoir, si besoin est, décompresser les résolutions supérieures. Le codage progressif présente l'avantage de lecture rapide et grossière de l'image, souvent suffisante pour de nombreuses applications.

E. Codage par motifs restreints

On peut montrer que si l'on découpe une image binaire en blocs de 8 x 8 pixels, il est possible de ne pas trop changer le contenu visuel de l'image en remplaçant les blocs par d'autres blocs choisis parmi un éventail de 63 blocs au plus. Ces 63 motifs peuvent être choisis une fois pour toutes de façon optimale après analyse de nombreuses images binaires, ou calculés et sauvés en en-tête de l'image courante. Danc ce cas, on passe de 64 bits à 6 bits pour le codage d'un bloc, soit un taux de compression inférieur à 10%.

Pour une fenêtre de 8 x 8 pixels, on se donne un taux de conformité avec les 63 motifs (nombre de pixels égaux, ...). Si le taux est supérieur à la valeur seuil, on émet le code à 6 bits du motif, correspondant dans la table (valeur approchée). Si cela n'est pas le cas, on émet le $64^{ème}$ code de la table, suivi des 64 bits de la fenêtre (valeur exacte), et ainsi de suite. Cette éventualité est, en général, beaucoup moins fréquente que la précédente.

F. JPEG

La norme JPEG, adoptée en 1992, décrit principalement le format des données compressées et le schéma de codage et de décodage. Les algorithmes de compression et de décompression y sont proposés mais n'ont pas de valeur normative. Les taux de compression sont fonction de la qualité souhaitée après décompression et de la vitesse demandée aux processus de codage et de décodage.

Quatre modes de fonctionnement sont spécifiés :

- codage séquentiel par transformée en cosinus discrète (DCT = Discrete Cosine Transform),
- codage progressif par DCT (par approximations successives pour une taille d'image donnée),
- codage hiérarchique (par approximations successives sur plusieurs résolutions),
- codage sans perte.

Nous nous focaliserons sur le codage séquentiel par DCT, il contient les idées essentielles du mode de compression JPEG et, de plus, les applications actuelles proposant une implémentation de JPEG ne connaissent généralement que cette méthode. Pour des détails supplémentaires et une application du codage JPEG par blocs aux images très volumineuses, on consultera [LS].

Le processus de codage JPEG contient trois étapes, illustrées par la figure 3.4. Il est appliqué à des blocs de 8 x 8 pixels en 8 bits par pixel. Les images couleurs voient l'algorithme appliqué à chacune de leurs composantes (Y, Cr, Cb).

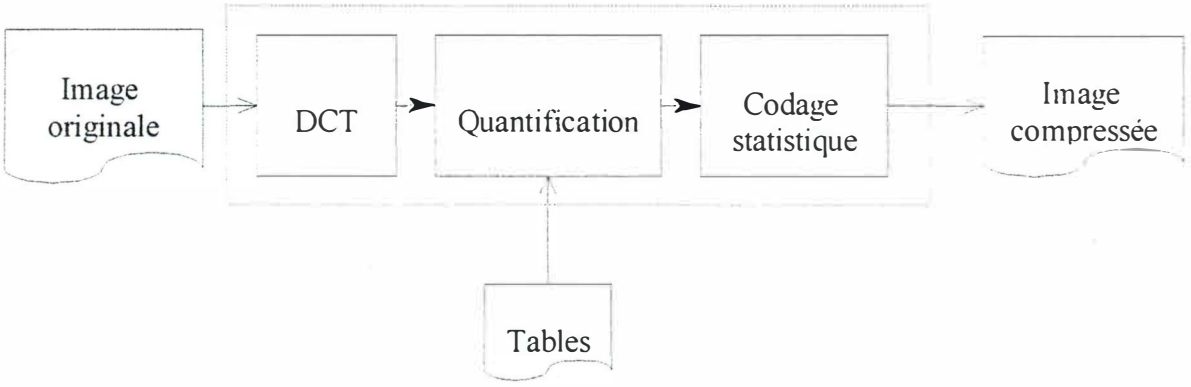


Figure 3.4 : principe de l'algorithme JPEG avec pertes

Le processus de décompression est obtenu en effectuant les mêmes étapes en sens inverse : décompression de Huffman ou arithmétique, déquantification et transformée en cosinus discrète inverse.

TRANSFORMEE EN COSINUS DISCRETE

Le bloc de 8 x 8 pixels est transformé en un ensemble d'amplitudes (64 coefficients) par la **transformée en cosinus discrète** à deux dimensions. La DCT bidimensionnelle de la fonction discrète (8 x 8) f(x,y) est donnée par :

$$F(u,v) = \frac{1}{4} c(u)\, c(v) \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

et la **DCT inverse** est :

$$f(x,y) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} c(u)\, c(v)\, F(u,v) \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

où $c(u) = c(v) = 1/\sqrt{2}$ pour u, v = 0, et $c(u) = c(v) = 1$ pour u, v = 1, ..., 7.

Notons que la DCT transforme simplement le bloc image d'une représentation dans une autre, en principe elle ne perd pas de données. On peut voir la DCT comme l'expression de chaque bloc en terme de combinaison linéaire de 64 blocs de base.

QUANTIFICATION

Chaque coefficient issu de la DCT est quantifié en utilisant les valeurs d'une **table de quantification** prédéterminée Q, chaque composante (Y, Cr, Cb) possédant sa propre table. Ces tables ne sont pas normatives, mais ont été établies à partir de critères expérimentaux psychovisuels, et constituent une base solide pour compresser la plupart des images. Tout utilisateur peut les modifier, les redéfinir et les exporter dans l'en-tête du format d'image compressée. La quantification s'effectue en divisant chaque coefficient de la DCT, F(u, v), par l'élément correspondant de quantification, Q(u, v), et en arrondissant le résultat par l'entier le plus proche :

$$Z(u, v) = \text{round}(F(u, v) / Q(u, v)).$$

L'étape de quantification produit l'essentiel de la compression, compression avec pertes, dans l'algorithme JPEG. Les coefficients DCT de haute fréquence, F(u,v), où u ou v est élevé, tendent à devenir plus petits que ceux de basse fréquence, pour des blocs provenant d'images typiques. Après quantification, beaucoup de ces coefficients deviennent nuls. Les coefficients quantifiés sont ordonnés en zigzag (figure 3.5), de façon à les parcourir dans un ordre plus ou moins croissant de fréquences. Un coefficient non nul est codé en indiquant d'abord le nombre de 0 qui le précèdent (codage par plages), diminuant drastiquement la place requise pour coder l'entièreté des coefficients.

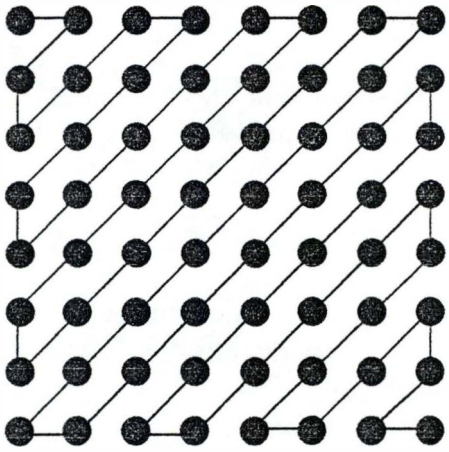


figure 3.5 : parcours d'un bloc DCT en zigzag

CODAGE STATISTIQUE

Les coefficients quantifiés sont enfin compressés en utilisant un codage de Huffman.

G. Compression par fractales : méthodologie générale

Les logiciels de dessin vectoriel en deux ou trois dimensions permettent de concevoir des images extrêmement sophistiquées, à l'aide de formes géométriques simples : segments, boîtes, cercles, cubes, sphères, ... Ces objets primitifs sont stockés dans le fichier image sous forme d'équations, ce qui rend les fichiers images vectoriels très peu gourmands en taille. Cependant, ce type de logiciels, basés sur la géométrie traditionnelle, a le grand désavantage

de n'être adapté qu'à des dessins relativement limités, généralement des constructions humaines : moteurs, plans de villes,... Pas question, donc, d'essayer de représenter un ciel nuageux, un visage humain ou une mer déchaînée à l'aide de ces outils DAO. On doit, pour ces dessins, se tourner vers des outils de type « paint », qui traitent les images en mode point, perdant ainsi tout l'avantage, point de vue place, que nous apportent les vecteurs.

Pour surmonter la difficulté et essayer de concilier les deux possibilités, images quelconques et gain de place, nous avons besoin d'une librairie plus riche de formes géométriques primitives. Ces formes doivent être flexibles et facilement contrôlables, de façon qu'elles puissent élaborer un nuage, un visage, ... La géométrie fractale fournit justement de telles formes.

Le fait d'utiliser des fractales pour dessiner n'est pas nouveau : décors des simulateurs de vols, forêts dans les « Walt Disney », planète Génésis dans « Star Trek », ... Ce qui est nouveau, c'est de partir d'une image réelle, photographie digitalisée, et d'essayer de trouver les fractales qui vont imiter cette image avec un degré de précision désiré. Etant donné que les fractales peuvent se représenter de manière compacte, on aboutira à un fichier image fortement compressé.

On part donc d'une image digitalisée. En utilisant des techniques de traitement d'images, similaires à celles employées pour la colorisation automatique de vieux films en noir et blanc : détection de contours, séparation de couleurs, analyse spectrale, ... , on découpe l'image en **segments**. Un segment peut être une feuille d'arbre, un nuage, ou un groupe de pixels plus complexe. On compare alors les segments avec la librairie de fractales. Cette librairie ne contient évidemment pas les fractales telles quelles, cela exigerait une quantité astronomique de place. Au lieu de cela, la librairie contient une collection relativement compacte de nombres appelés **codes des systèmes de fonctions itérées (IFS codes)**, qui vont reproduire les fractales correspondantes. De plus, le système d'indexation de la librairie est tel que les images qui se ressemblent sont proches : les codes proches correspondent à des fractales proches ; cela permet de construire des procédures automatiques de recherche de fractales qui approximent une image donnée. Un résultat mathématique connu sous le nom de « théorème de collage » garantit qu'il existe toujours un IFS convenable, et donne une méthode pour le trouver. Une fois calculés les codes IFS de tous les segments, on peut se débarrasser de l'image originale, car les codes obtenus suffisent à la restituer.

Chapitre 4 : Rappels mathématiques

Le but de ce chapitre est de fournir certains outils mathématiques indispensables à la bonne compréhension des autres parties. On introduira les notations, définitions et informations relatives à la topologie et à la géométrie, qui vont nous fournir les bases de la description de plusieurs propriétés des images. En particulier, on abordera les transformations affines, qui sont essentielles dans la théorie des IFS.

Certaines notions n'interviennent qu'une seule fois et peuvent paraître superflues, cependant elles sont utiles pour démontrer certains théorèmes énoncés, démonstrations que l'on pourra trouver dans [FE].

I. Espaces et transformations

On appellera **espace**, un ensemble possédant une certaine structure. Des exemples d'espaces sont l'intervalle [0,1], la droite réelle $\Re$ et le plan euclidien $\Re^2$. Une façon de donner une structure à un espace est de lui fournir une métrique.

Définition : Un **espace métrique**, noté (E, d), est un espace muni d'une fonction à valeur réelle d : E X E $\rightarrow \Re$, qui mesure la distance entre paires de points x et y de E. Supposons que d possède les propriétés suivantes :

1) $d(x, y) = d(y, x), \forall x, y \in E$ (symétrique)
2) $0 < d(x, y) < \infty, \forall x, y \in E, x \neq y$ (positive)
3) $d(x, x) = 0, \forall x \in E$
4) $d(x, y) \leq d(x, z) + d(z, y), \forall x, y, z \in E$ (inégalité triangulaire),

alors d est une **distance** sur l'espace X.

Exemples : $(\Re^2, d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}, \forall x, y \in \Re^2)$. Il s'agit du plan euclidien muni de la métrique euclidienne.

$(\Re^2, d(x, y) = \max \{| x_1 - y_1 |, | x_2 - y_2 | \}, \forall x, y \in \Re^2)$. Il s'agit du plan euclidien muni de la métrique absolue.

$(F, d(f(x, y), g(x, y)) = \sqrt{\int_c^d \int_a^b (f(x,y) - g(x,y))\, dx\, dy}, \forall f, g \in F, \forall x, y \in \Re^2)$, où F est l'ensemble des fonctions de $\Re^2$ dans $\Re$. Il s'agit de la métrique **rms** (root mean square) souvent rencontrée dans les applications de compression d'images.

Définition : Soit E un espace. Une **transformation** sur E est une fonction f : E → E. La fonction f est **bijective** si x, y ∈ E avec f(x) = f(y) implique x = y. Si S ⊂ E, f(S) = { f(x) : x ∈ S }. Si f est bijective et si f(E) = E, f est dite **inversible** et il est dès lors possible de définir une transformation f^{-1} : E → E, **appelée transformation inverse** de f, par $f^{-1}(y) = x$, où x ∈ E est l'unique point tel que y = f(x).

Définition : Soit f : E → E, une transformation sur un espace. Les **itérées suivantes** de f sont des transformations $f^{\circ n}$: E → E, définies par $f^{\circ 0}(x) = x$, $f^{\circ 1}(x) = f(x)$, $f^{\circ(n+1)}(x) = f \circ f^{\circ n}(x) = f(f^{\circ n}(x))$, pour n = 0, 1, 2, 3, ... Si f est inversible, alors **les itérées précédentes** de f sont des transformations $f^{\circ(-m)}(x)$: E → E, définies par $f^{\circ(-1)}(x) = f^{-1}(x)$, $f^{\circ(-m)}(x) = (f^{m})^{-1}(x)$, pour m = 1, 2, 3, ...

II. Transformations affines sur $\Re$

Une **transformation affine** sur $\Re$ est une transformation f : $\Re$ → $\Re$ de la forme f(x) = a.x + b, ∀ x ∈ $\Re$, où a et b sont des constantes réelles. Etant donné l'intervalle I = [0, 1], f(I) est un nouvel intervalle de longueur |a| ; f effectue une remise à l'échelle d'un facteur a. L'extrémité gauche de l'intervalle, 0, est déplacée en b, et f(I) figure à gauche ou à droite de b selon que le coefficient a est négatif ou positif, respectivement.

L'action de la transformation affine sur toute la droite réelle peut être décrite de la façon suivante. Toute la droite est « étirée » à partir de l'origine si |a| > 1 et « contractée » vers l'origine si |a| < 1; elle subit une rotation de 180° autour de l'origine si a < 0. La droite est translatée, c.-à-d. déplacée en entier, d'un facteur b. La translation est effectuée vers la gauche ou vers la droite selon que b < 0 ou b > 0, respectivement. Notons que si f : [0, 1] → [0, 1] et g : [0, 1] → [0, 1] sont deux transformations affines sur l'intervalle [0, 1], il en est de même avec f o g.

III. Transformations affines dans le plan euclidien

Définition : Une transformation w : $\Re^2$ → $\Re^2$ de la forme :

$$w(x, y) = (a.x + b.y + e, c.x + d.y + f),$$

où a, b, c, d, e, f sont des nombres réels, est appelée **une transformation affine à deux dimensions**.

Nous utiliserons la notation équivalente :

$$w(\vec{x}) = w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = A.\vec{x} + \vec{T}$$

A est une matrice 2 x 2 réelle et $\vec{T}$ est un vecteur colonne, que l'on ne distinguera pas de la paire de coordonnées (e, f) de $\Re^2$.

La matrice A peut toujours s'écrire sous la forme :

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} r_2 \cos\theta_1 & -r_2 \sin\theta_2 \\ r_1 \sin\theta_1 & r_2 \cos\theta_2 \end{pmatrix},$$

où (r_1, θ_1) sont les coordonnées polaires du point (a, c) et $(r_2, \theta_2 + \pi/2)$ les coordonnées polaires du point (b, d). C'est-à-dire :

$r_1 = \sqrt{a^2 + c^2}$, $\theta_1 = \text{tg}^{-1}(c/a)$ si $a \neq 0$,
$\theta_1 = \pi/2$ si $a = 0$ et $c \geq 0$,
$\theta_1 = 3.\pi/2$ si $a = 0$ et $c < 0$,
$r_2 = \sqrt{b^2 + d^2}$, $\theta_2 = \text{tg}^{-1}(b/d)$ si $d \neq 0$,
$\theta_2 = \pi/2$ si $d = 0$ et $b \geq 0$,
$\theta_2 = 3.\pi/2$ si $d = 0$ et $b < 0$.

La transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow A \begin{pmatrix} x \\ y \end{pmatrix}$$

de $\Re^2$, appelée **transformation linéaire**, transforme tout triangle possédant un sommet à l'origine en un autre triangle possédant, lui aussi, un sommet à l'origine, à condition que $a.d - b.c \neq 0$ (figure 4.1).

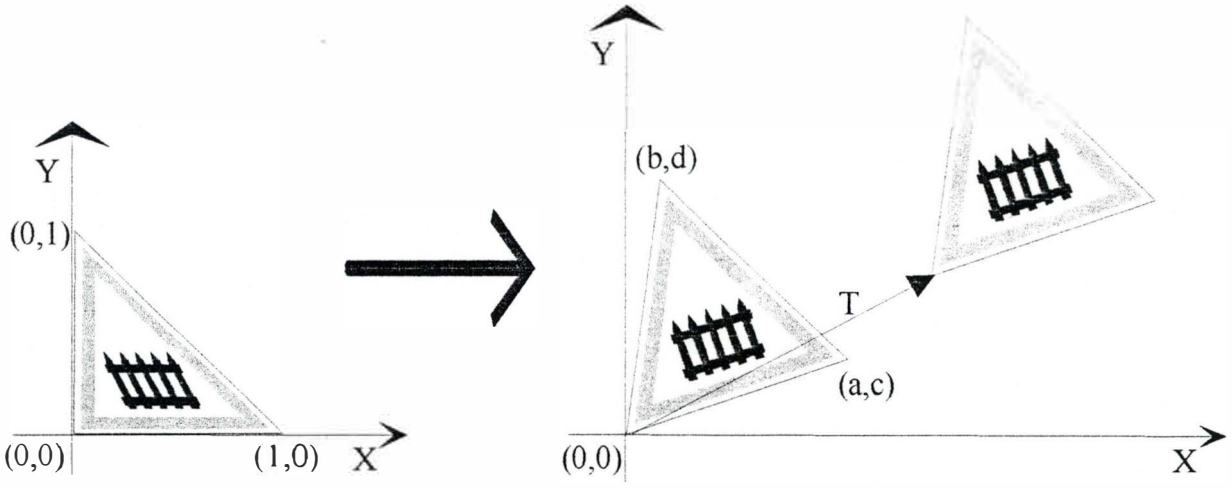


figure 4.1 : transformation affine : transformation linéaire A suivie d'une translation T

La transformation affine $w(\vec{x}) = A.\vec{x} + \vec{T}$ de $\Re^2$ consiste en une transformation linéaire A suivie par une translation spécifiée par le vecteur $\vec{T}$ (figures 4.1 et 4.8).

Il est toujours possible de trouver une transformation affine qui transforme un triangle donné de sommets (x_1, x_2), (y_1, y_2) et (z_1, z_2), en un autre triangle donné de sommets (x'_1, x'_2), (y'_1, y'_2) et (z'_1, z'_2). Les coefficients de la transformation : a, b, c, d, e et f sont obtenus en résolvant le système d'équations linéaires :

$$
\begin{aligned}
x_1.a + x_2.b + e &= x'_1,\\
y_1.a + y_2.b + e &= y'_1,\\
z_1.a + z_2.b + e &= z'_1,\\
x_1.c + x_2.d + f &= x'_2,\\
y_1.c + y_2.d + f &= y'_2,\\
z_1.c + z_2.d + f &= z'_2.
\end{aligned}
$$

L'inverse de la transformation affine $w : \Re^2 \rightarrow \Re^2$ définie par $w(x, y) = (a.x + b.y + e, c.x + d.y + f)$ est représentée par la transformation $w^{-1}(x, y) = (d.x - b.y - d.e + b.f, - c.x + a.y + c.e - a.f) / (a.d - b.c)$, où $(a.d - b.c) \neq 0$.

La transformation affine $w : \Re^2 \rightarrow \Re^2$ définie par $w(x, y) = (x, y)$ est appelée la **transfomation identité**.

Une transformation affine $w : \Re^2 \rightarrow \Re^2$ de la forme $w(x, y) = (r_1.x, r_2.y)$, où r_1 et r_2 sont des constantes positives, est appelée une **dilatation** (figure 4.2).

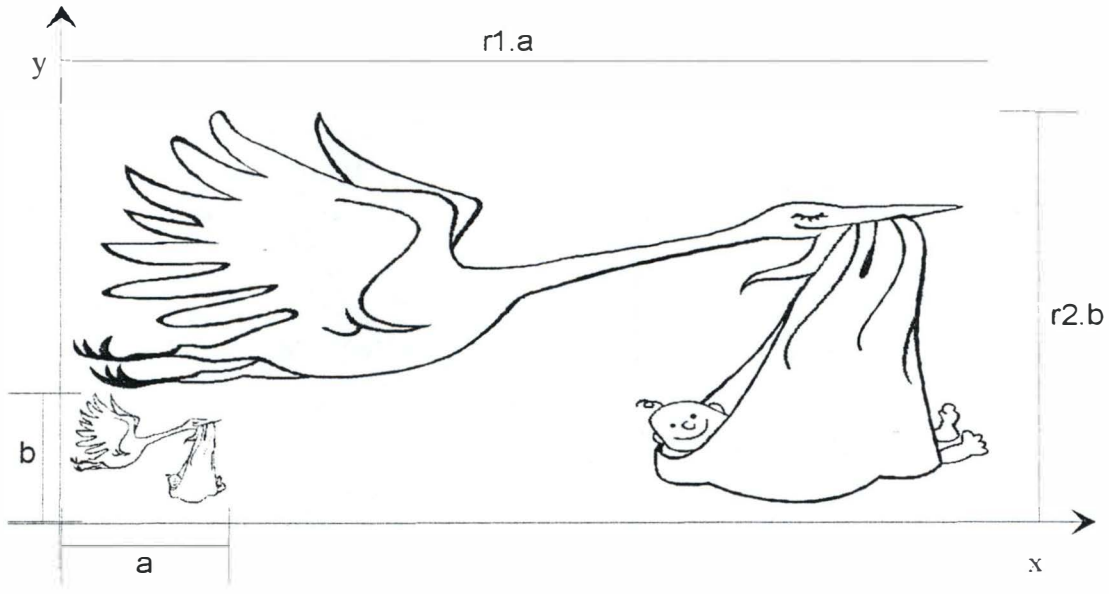


figure 4.2 : dilatation : w(x, y) = (r1.x, r2.x)

Les deux transformations affines de $\Re^2$ dans $\Re^2$ $w(x, y) = (x, -y)$ et $w(x, y) = (-x, y)$ sont des **réflexions**, la première par rapport au deuxième axe de coordonnées (figure 4.4) et la seconde par rapport au premier axe de coordonnées (figure 4.3).

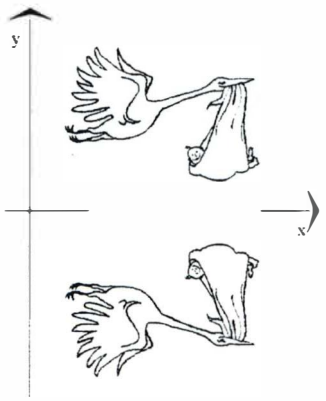


figure 4.3 : symétrie par rapport à l'axe X

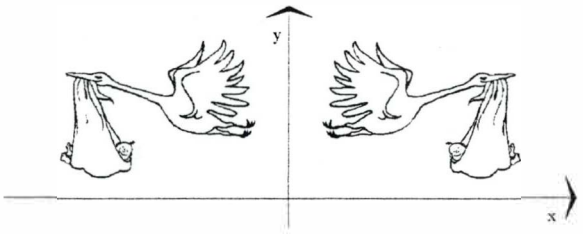


figure 4.4 : symétrie par rapport à l'axe Y

Les transformations affines w : $\Re^2 \rightarrow \Re^2$ de la forme w(x, y) = (x + f, y + g), où f et g sont des constantes positives, sont appelées des **translations** (figure 4.5).

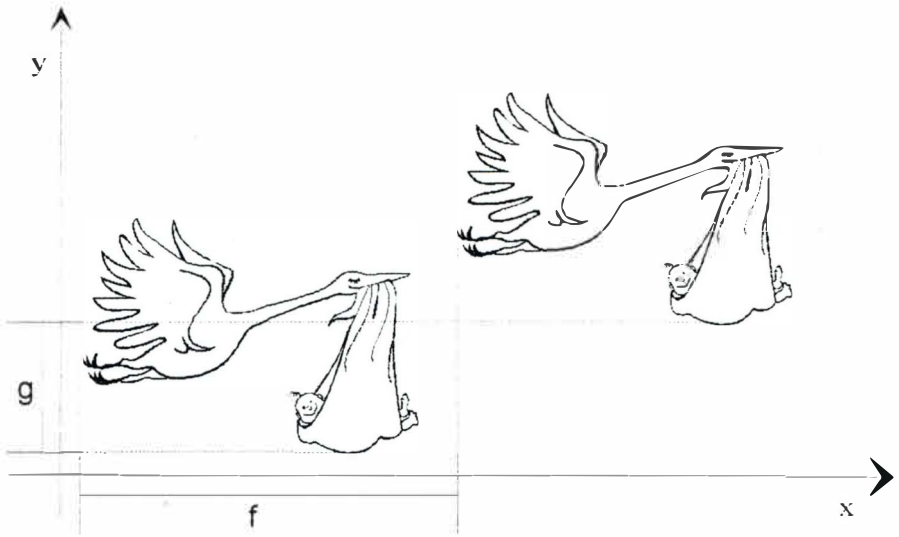


figure 4.5 : translation : w(x, y) = (x+f, y+g)

Une transformation affine w : $\Re^2 \rightarrow \Re^2$ de la forme

$$w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

est appelée une **rotation** (figure 4.6).

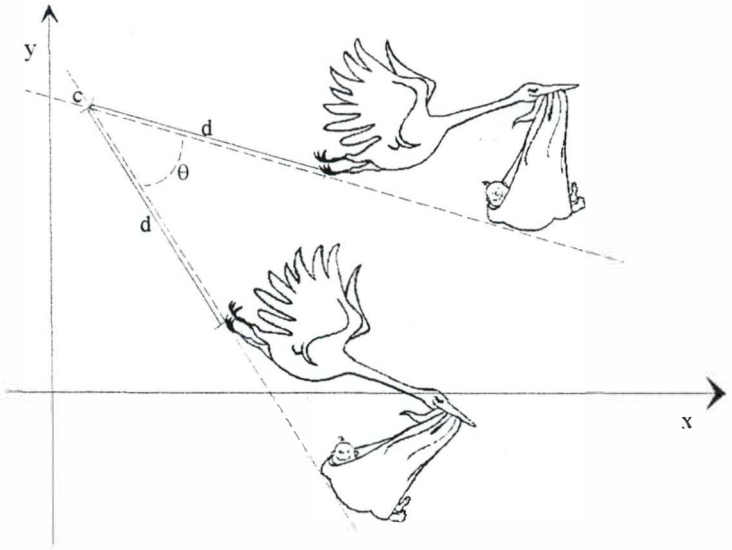


figure 4.6 : rotation d'un angle θ autour d'un point c

Une transformation linéaire possédant une des formes suivantes :

$$w(x, y) = (x + by, y) \text{ ou } w(x, y) = (x, cx + y),$$

où b et c sont des constantes réelles, est appelée **transformation de cisaillement** (figure 4.7). Dans chaque cas, une des deux coordonnées reste inchangée.

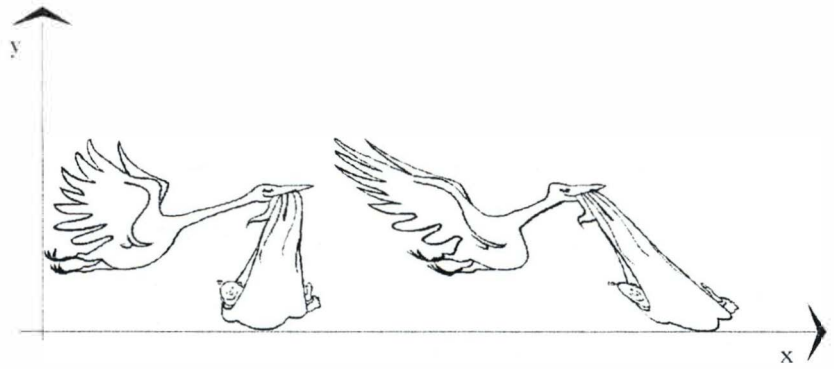


figure 4.7 : transformation de cisaillement w(x, y) = (x + b.y, y)

Une transformation affine w : $\Re^2 \rightarrow \Re^2$ est appelée une **similitude** si elle possède une des formes

$$w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\cos\theta & -r\sin\theta \\ r\sin\theta & r\cos\theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix},$$

$$w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\cos\theta & r\sin\theta \\ r\sin\theta & -r\cos\theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix},$$

où (e, f) $\in \Re^2$ est le **vecteur translation**, r est un nombre réel non nul et θ un angle tel que $\theta \in [0, 2\pi[$. θ est appelé **angle de rotation**.

Si $w_1 : \Re^2 \rightarrow \Re^2$ et $w_2 : \Re^2 \rightarrow \Re^2$ sont deux transformations affines, alors $w_3 = w_2 \circ w_1$ est également une transformations affine. Dès lors, on peut composer des transformations affines pour en construire de nouvelles (voir figure 4.8). Cela mène naturellement à la question de trouver un ensemble de transformations affines élémentaires à partir desquelles toutes les transformations affines pourront être obtenues par composition.

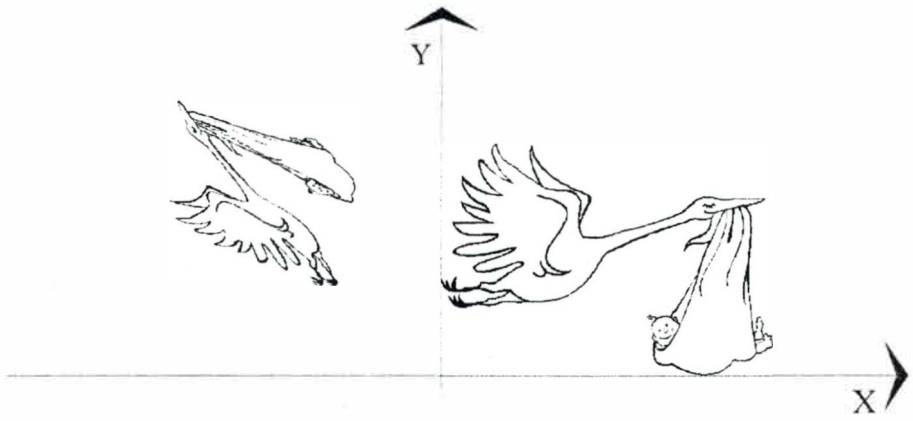


figure 4.8 : transformation affine quelconque

IV. Propriétés topolgiques des espaces métriques et des transformations

Définition : Deux **métriques** d_1 et d_2 sur un espace E sont **équivalentes** s'il existe des constantes c_1 et c_2, $0 < c_1 < c_2 < \infty$, telles que :

$$c_1 d_1(x, y) \leq d_2(x, y) \leq c_2 d_1(x, y), \forall (x, y) \in E \times E.$$

Définition : Deux **espaces métriques** (E_1, d_1) et (E_2, d_2) sont **équivalents** s'il existe une fonction inversible $h : E_1 \rightarrow E_2$, telle que la métrique d_1' sur E_1 définie par

$$d_1'(x, y) = d_2(h(x), h(y)), \forall x, y \in E_1$$

est équivalente à d_1.

Exemple : Si $E_1 = [1, 2]$ et $E_2 = [0, 1]$, si d_1 désigne la métrique Euclidienne et $d_2(x, y) = 2.|x - y|$ dans E_2, alors (E_1, d_1) et (E_2, d_2) sont des espaces métriques équivalents.

Définition : Une **fonction** $f : E \rightarrow F$ d'un espace métrique (E, d) dans un espace métrique (F, e) est **continue** si, pour chaque $\varepsilon > 0$ et $x \in E$, il existe un $\delta > 0$ tel que :

$$d(x, y) < \delta \Rightarrow e(f(x), f(y)) < \varepsilon.$$

Définition : Une **suite** de points notée $\{x_n\}_{n=1}^{\infty}$ dans un espace E est une fonction $f : Z^+ \rightarrow E$, où Z^+ désigne l 'ensemble des entiers positifs, et $f(n) = x_n$ pour tous les $n \in Z^+$. Une suite d'entiers positifs $\{n_i\}_{i=1}^{\infty}$ est une fonction qui préserve l'ordre $g : Z^+ \rightarrow Z^+$ telle que $g(i) = n_i$ pour tous les $i \in Z^+$. On dit que $g : Z^+ \rightarrow Z^+$ préserve l'ordre si elle possède la propriété :

$$m < n \Rightarrow g(m) < g(n), \forall\ m, n \in Z^+.$$

Une **sous-suite** $\{x_{n_i}\}_{n=1}^{\infty}$ est une fonction de la forme $f \circ g : Z^+ \rightarrow E$ avec f et g comme définis précédemment, où x_{n_i} désigne le point $f(g(i))$.

Définition : Une suite de points $\{x_n\}_{n=1}^{\infty}$ dans un espace métrique (E, d) est appelée **une suite de Cauchy** si, pour tout nombre donné $\varepsilon > 0$, il existe un entier $N > 0$ tel que :

$$d(x_n, x_m) < \varepsilon, \forall\ n, m > N.$$

Définition : Une **suite** de points $\{x_n\}_{n=1}^{\infty}$ dans un espace métrique (E, d) **converge** vers un point $x \in E$ si, pour tout nombre donné $\varepsilon > 0$, il existe un entier $N > 0$ tel que :

$$d(x_n, x) < \varepsilon, \forall\ n > N.$$

Dans ce cas, la **suite** est dite **convergente**. Le point $x \in E$ vers lequel converge cette suite est appelé la **limite de la suite**. On utilisera la notation $x = \lim_{n\rightarrow\infty} x_n$.

Théorème : Si une suite de points $\{x_n\}_{n=1}^{\infty}$ dans un espace métrique (E, d) converge en un point $x \in E$, alors $\{x_n\}_{n=1}^{\infty}$ est une suite de Cauchy.

Définition : Un **espace métrique** (E, d) est dit **complet** si toute suite de Cauchy $\{x_n\}_{n=1}^{\infty}$ de E possède une limite $x \in E$.

Exemple : L'espace métrique (Q, d), où Q est l'ensemble des rationnels et $d(x, y) = |x - y|$, n'est pas complet. En effet la suite de Q : 1, 3/2, 7/5, 17/12, 41/29, 99/70, ..., p / q, (p + 2q) / (p + q), ... , converge vers $\sqrt{2}$ qui est un réel non rationnel.

La complétude est une propriété que l'on exigera de l'espace métrique qui servira à modéliser les images du monde réel, car l'on veut évidemment que la suite d'images générées par l'application successive des IFS converge bien vers une image. (voir chapitres suivants)

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). Un point $x \in E$ est appelé un **point limite** de S si il existe une suite $\{x_n\}_{n=1}^{\infty}$ de points $x_n \in S\{x\}$ telle que $\lim_{n\rightarrow\infty} x_n = x$.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). La **fermeture** de S, notée $\overline{S}$ est définie comme $\overline{S} = S \cup \{\text{points limites de S}\}$.

Définition : S est **fermé** s'il contient tous ses points limites, c.-à-d. $S = \overline{S}$.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). S est **ouvert** si, pour chaque $x \in S$, il existe un $\varepsilon > 0$ tel que $B(x, \varepsilon) = \{ y \in E : d(x, y) \leq \varepsilon \} \subset S$. Cette définition est équivalente à : S est ouvert s'il n'est pas fermé.

Définition : S est **parfait** s'il est égal à l'ensemble de ses points limites.

Exemple : $S = [0, 1]$, sous-ensemble de ($\Re$, Euclidienne) est fermé et parfait.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). S est **compact** si toute suite $\{x_n\}_{n=1}^{\infty}$ de S contient une sous-suite possédant une limite dans S.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). S est **borné** s'il existe un point $a \in E$ et un nombre $R > 0$ tels que $d(a, x) < R$, $\forall x \in E$.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). S est **totalement bornée** si, pour chaque $\varepsilon > 0$, il existe un ensemble fini de points $\{ y_1, y_2, \ldots, y_n \} \subset S$ tel que pour tout $x \in E$, $d(x, y_i) < \varepsilon$ pour un $y_i \in \{ y_1, y_2, \ldots, y_n \}$.

Théorème : Soit (E, d) un espace métrique complet. Soit $S \subset E$. Alors, S est compact si et seulement si il est fermé et totalement borné.

Théorème : Si deux espaces métriques sont équivalents, alors les propriétés topologiques suivantes des ensembles sont préservées : ouvert, fermé, complet, compact, borné et parfait.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). Un point $x \in E$ est un **point frontière** de S si pout tout nombre $\varepsilon > 0$, $B(x, \varepsilon) = \{ y \in E : d(x, y) \leq \varepsilon \}$ contient un point de X / S et un point de S.

Définition : L'ensemble de tous les points frontières de S est appelée la **frontière** de S, et est noté δS.

Définition : Soit $S \subset E$, un sous-ensemble d'un espace métrique complet (E, d). Un point $x \in S$ est appelé **un point intérieur** de S s'il existe un nombre $\varepsilon > 0$ tel que $B(x, \varepsilon) = \{ y \in E : d(x, y) \leq \varepsilon \} \subset S$.

Définition : L'ensemble des points intérieurs de S est appelé l'**intérieur** de S, et noté S°.

Definition : Un espace métrique (E, d) est **connexe** si les deux seuls sous-ensembles de E qui soient simultanément ouverts et fermés sont E et {}. Un sous-ensemble $S \subset E$ est connexe si l'espace métrique (S, d) est connexe.

Définition : S est **totalement non connexe** si les seuls sous-ensembles connexes non vides de S sont des sous-ensembles constitués d'un seul point.

Exemples : L'espace métrique ($\Im$, Euclidienne) est connexe.

L'espace métrique ($E = [1, 2] \cup \{3\}$, Euclidienne) est non connexe

V. Théorème de transformation contractive

Définition : Soit $f : E \rightarrow E$ une transformation sur un espace. Un point $x_f \in E$ tel que $f(x_f) = x_f$ est appelé un **point fixe** de la transformation.

Définition : Une **transformation** $f : E \rightarrow E$ sur un espace métrique (E, d) est dite **contractive** (ou **contractante**) ou **transformation de contraction** s'il existe une constante $s : 0 \leq s < 1$ telle que :

$$d(f(x), f(y)) \leq s.d(x, y), \forall x, y \in E.$$

Un tel nombre s est appelé **facteur de contraction** pour la transformation f.

Exemple : La fonction $f(x) = 1/x$ est contractive sur l'intervalle fermé [1, 2], mais pas sur l'intervalle ouvert]0, 1[. En effet, sur [1, 2], $| f(x) - f(y) | = | 1/x - 1/y | = | (x - y) / (xy) | \leq | x - y |$, mais sur]0, 1[, la pente de f devient non bornée et $| 1/x - 1/y |$ peut devenir très grand.

Le résultat suivant étant souvent mentionné, nous donnerons sa démonstration, qui peut servir de modèle à suivre pour prouver d'autres théorèmes, en particulier le théorème d'IFS et le théorème de collage du chapitre suivant.

Théorème : (théorème de transformation contractive) (voir figures 4.9)
Soit $f : E \rightarrow E$ une transformation contractive sur un espace métrique complet (E, d). Alors f possède exactement un point fixe $x_f \in E$, et, de plus, pour tout point $x \in E$, la suite $\{f^{\circ n}(x) : n = 1, 2, 3, ...\}$ converge vers x_f, c.-à-d.

$$\lim_{n \rightarrow \infty} f^{\circ n}(x) = x_f, \forall x \in E.$$

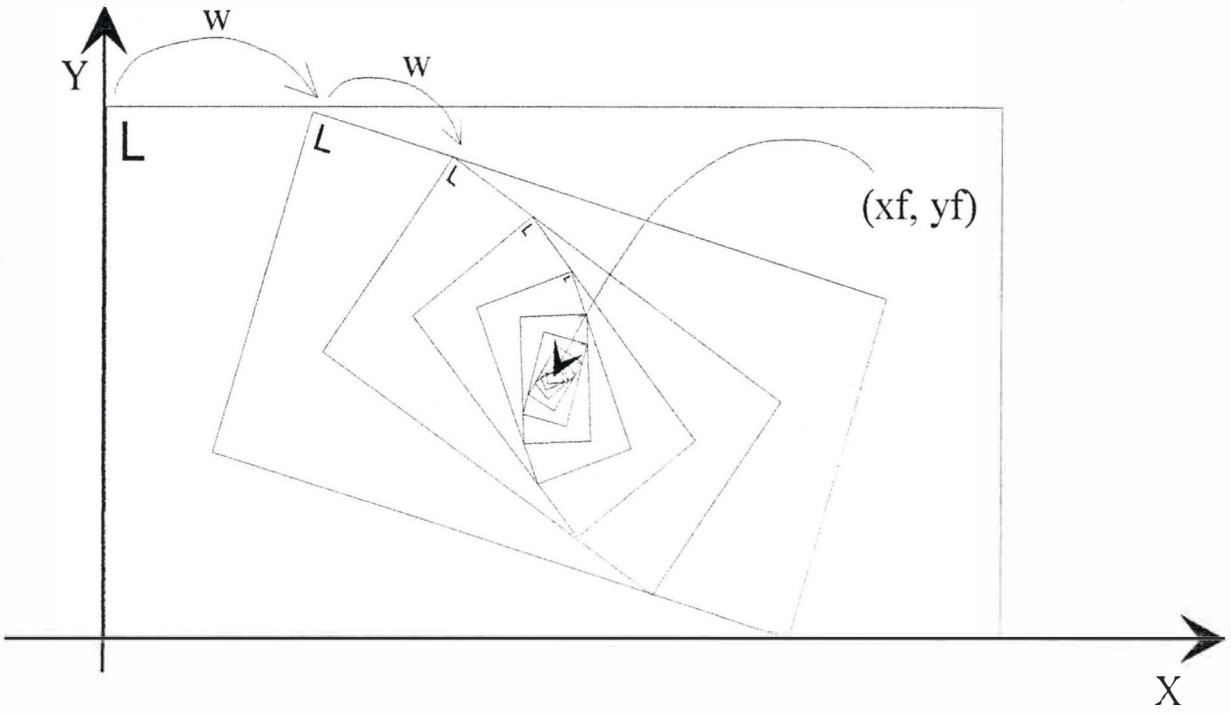


figure 4.9a : point fixe (x_f, y_f) de la transformation affine w

Démonstration : Soit $x \in E$. Soit $s : 0 \leq s < 1$ un facteur de contraction pour f. Alors

$$d(f^{\circ n}(x), f^{\circ m}(x)) \leq s^{\min(m,n)}.d(x, f^{\circ|m-n|}(x)), \forall\ m, n = 1, 2, 3, ...$$

où l'on a fixé $x \in E$. En particulier, pour $k = 0, 1, 2, ...$, on a :

$$d(x, f^{\circ k}(x)) \leq d(x, f(x)) + d(f(x), f^{\circ 2}(x)) + ... + d(f^{\circ(k-1)}(x), f^{\circ k}(x))$$

par inégalité triangulaire, d'où

$$d(x, f^{\circ k}(x)) \leq (1 + s + s^2 + s^3 + ... + s^{k-1}).d(x, f(x))$$

par définition du facteur de contraction, ce qui donne, comme somme des termes d'une progression géométrique

$$d(x, f^{\circ k}(x)) \leq (1 - s^k)/(1 - s)$$

or, $0 \leq s < 1 \Leftrightarrow 0 \leq s^k < 1, k = 1, 2, 3, ... \Leftrightarrow 0 < 1 - s^k \leq 1, k = 1, 2, 3, ...$, d'où

$$d(x, f^{\circ k}(x)) \leq (1 - s)^{-1}.$$

Si on substitiue ce résultat dans l'inéquation de départ, on obtient

$$d(f^{\circ n}(x), f^{\circ m}(x)) \leq s^{\min(m,n)}.(1 - s)^{-1}.d(x, f(x)), \forall\ m, n = 1, 2, 3, ...,$$

qui montre que la suite $\{f^{\circ n}(x)\}_{n=0}^{\infty}$ est une suite de Cauchy. Comme E est complet, cette suite de Cauchy possède une limite $x_f \in E$, et on a :

$$\lim_{n\to\infty} f^{\circ n}(x) = x_f.$$

Nous allons montrer que x_f est un point fixe de f. Etant donné que f est contractive, elle est continue, et donc

$$f(x_f) = f(\lim_{n\to\infty} f^{\circ n}(x)) = \lim_{n\to\infty} f^{\circ(n+1)}(x) = x_f.$$

Il reste à démontrer qu'il n'existe qu'un point fixe. Supposons qu'il en existe deux : soient x_f et y_f, les deux points fixes de f. Alors, $f(x_f) = x_f$ et $f(y_f) = y_f$, et

$$d(f(x_f), f(y_f)) \leq s.d(x_f, y_f) = d(x_f, y_f),$$

d'où l'on tire que $(1 - s).d(x_f, y_f) \leq 0$, qui implique $d(x_f, y_f) = 0$, et donc $x_f = y_f$ et il n'existe donc qu'un seul point fixe, ce qui complète la démonstration.

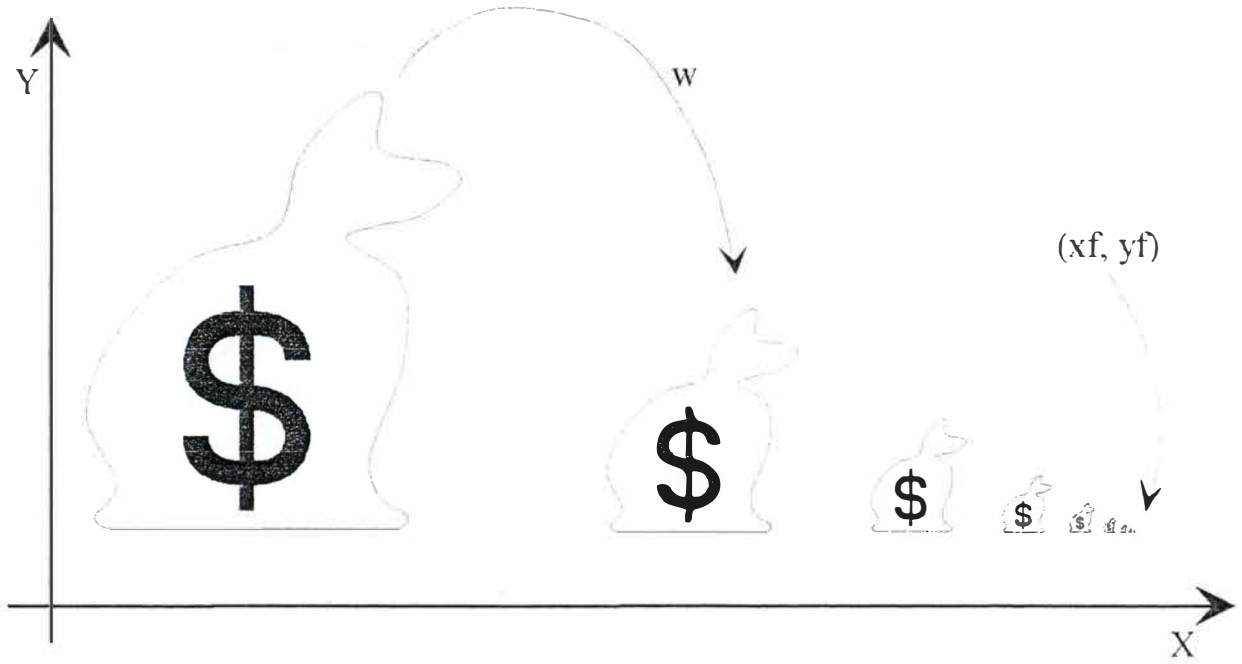


figure 4.9b : point fixe (xf, yf) de la transformation w

Exemple : La transformation affine $f : \Re \to \Re$ définie par $f(x) = x/2 + ½$ est une transformation contractive, de facteur de contraction $s = 0,5$ et de point fixe $x_f = 1$. Donc, on a

$$\lim_{n\to\infty} f^{\circ n}(x) = 1, \forall\, x \in \Re.$$

En particulier, on en déduit que $½ + (½ + (½ + (½ + (\ldots))))) = 1$.

VI. Mesures

On utilisera parfois, pour la modélisation des images du monde réel, la notion de mesure. On n'expose évidemment pas ici une théorie complète de la mesure, on essaie juste de donner quelques idées ainsi qu'une définition suffisamment claire pour que l'on puisse l'appliquer de façon intuitive lorsqu'on y fait appel, en particulier lors de l'intégration sur une mesure, que l'on nomme intégration selon Lebesgue, et que l'on introduit ici par un exemple.

Les deux patrons du bistrot « L'intégrale » font leur caisse en fin de journée, mais séparément et de façon différente.

Le premier, Riemann, a crocheté sur une épingle tous les tickets de la journée, et comptabilise simplement en additionnant les totaux inscrits sur les différents papiers. Pour lui, la recette est de

$$S_R = t_1 + t_2 + \ldots + t_N,$$

où N est le nombre de tickets.

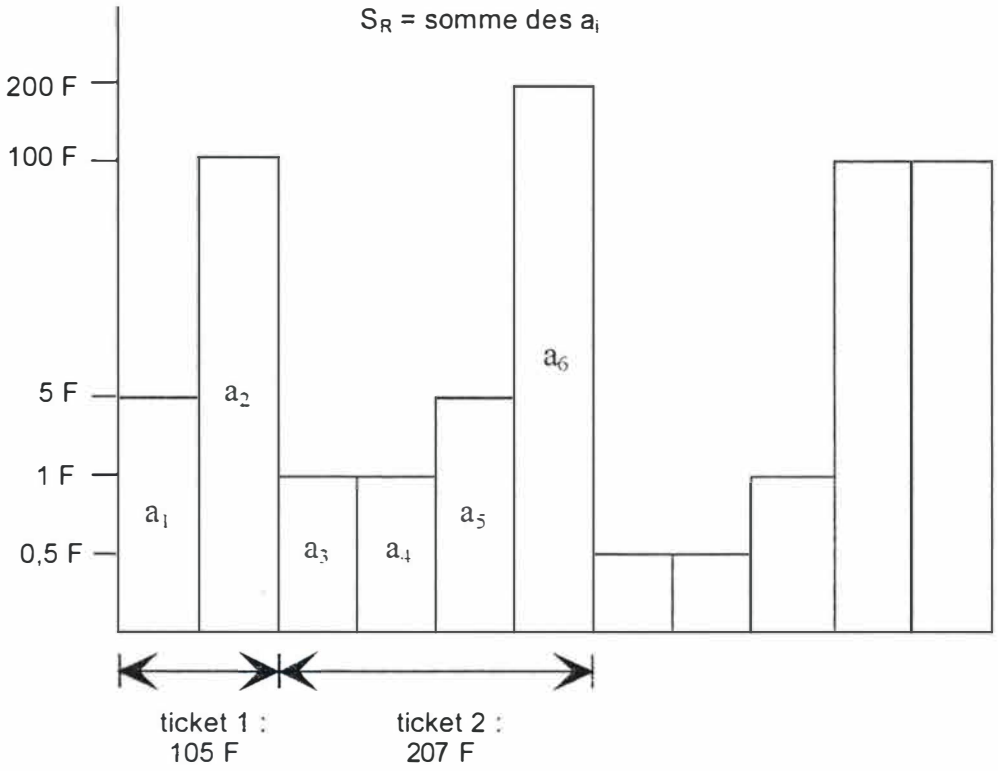
Le second, Lebesgue, recense simplement le contenu de chaque petit compartiment du tiroir caisse, sachant que chaque compartiment contient la totalité des pièces de monnaie ou billets d'une somme donnée : un compartiment pour les pièces de 50 centimes, un autre compartiment pour les pièces de 1 franc, un pour les pièces de 5 francs, ..., un pour les billets de 100 francs, encore un autre pour les billets de 200 francs, ... Sa recette calculée est de

$$S_L = n_{0,5} \times 0,5 + n_1 \times 1 + n_5 \times 5 + \ldots + n_{100} \times 100 + n_{200} \times 200 + \ldots$$

où n_i est le nombre de pièces ou de billets de montant i franc(s).

On trouve évidemment, s'il n'existe pas de caisse noire, $S_R = S_L$. La figure 4.10 représente cette situation. On peut y voir que la façon de mesurer n'est pas la même pour les deux graphes : en particulier, dans le second cas, on mesure des images réciproques $f^{-1}(0,5)$, $f^{-1}(1)$, $f^{-1}(5)$, ... Plus précisément, on dénombre, on « mesure » ces ensembles.

Point de vue de Riemann



Point de vue de Lebesgue

$S_L = (0{,}5\text{ F} \times 2) + (1\text{ F} \times 3) + (5\text{ F} \times 2) + (100\text{ F} \times 3) + (200\text{ F} \times 1)$

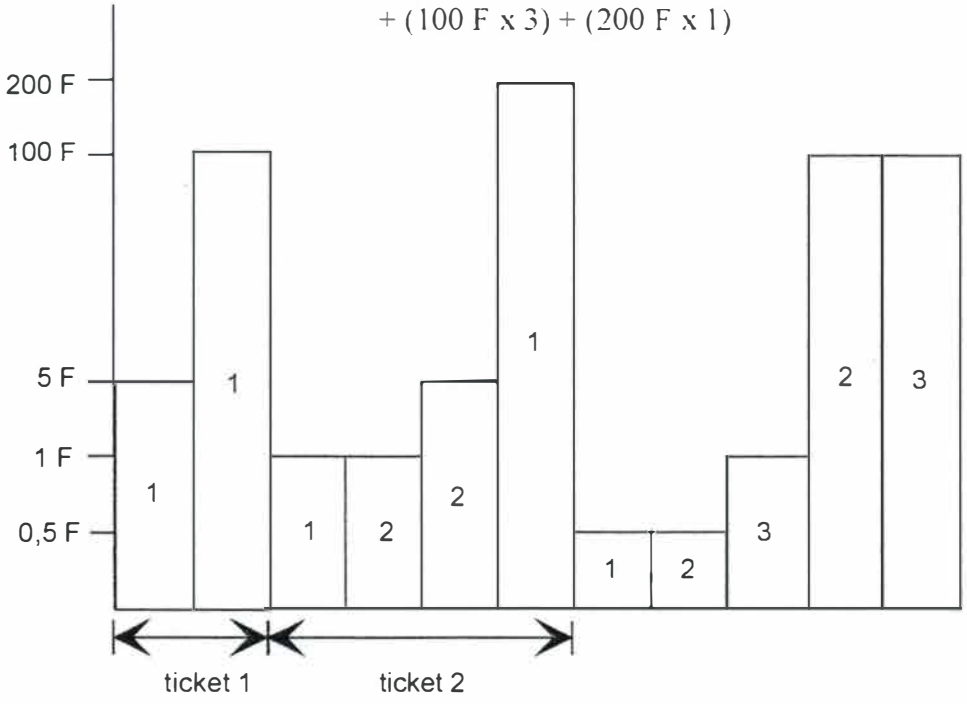


figure 4.10 : intégration selon Riemann et Lebesgue

La théorie de la mesure est fondée sur l'idée d'étendre la notion de longueur d'un intervalle dans $\Re$, d'aire d'un rectangle dans $\Re^2$, ...

Définition : Soit X un ensemble, et P(X) l'ensemble des parties de X. $T \subset P(X)$ est une **tribu** si et seulement si

1) $\varnothing$ et X sont dans T
2) $S \in T \Rightarrow X \setminus S \in T$ (T est fermée par complémentaire)
3) $S_1, S_2, \ldots \in T \Rightarrow \bigcup_{n=1}^{\infty} S_n \in T$ (T est fermée par union dénombrable)

Exemple : La **tribu des boréliens** est la tribu engendrée par les sous-ensembles ouverts de l'espace métrique X. Un **borélien** est un élément de cette tribu.

Définition : Soit T une tribu sur X. Les éléments de T sont appelés **ensembles mesurables**, et le couple (X, T) **espace mesurable**.

Définition : Soit T une tribu sur X. Une **mesure** sur X est une application $\mu : X \to \Re_+ \cup \{+\infty\}$ possédant les propriétés suivantes :

1) $\mu(\varnothing) = 0$
2) si S_n est une suite d'ensembles mesurables deux à deux disjoints, on a

$$\mu\left(\bigcup_{n=1}^{\infty} S_n\right) = \sum_{n=1}^{\infty} \mu(S_n).$$

Le triplet (X, T, μ) est un **espace mesuré**.

Exemple : Soient X un ensemble, T une tribu sur X et a un élément de X. On pose

1) $\mu_a(S) = \begin{cases} 1 \text{ si } a \in S \\ 0 \text{ sinon} \end{cases}$

2) $\mu_d(S) = \begin{cases} \text{nombre d'elements de } S \text{ s'il est fini} \\ +\infty \text{ sinon} \end{cases}$

μ_a et μ_d sont des mesures sur T

Exemple : Une **mesure de Borel** sur X est la mesure définie sur les boréliens.

En fait, les mesures ne sont en général pas définies sur une tribu, mais on les explicite d'abord sur des tribus et on les prolonge ensuite sur des ensembles. C'est le cas pour la mesure de Lebesgue sur $\Re^n$.

On dit qu'un ensemble E est de mesure nulle si pour tout $\varepsilon > 0$, il existe un ensemble ouvert de mesure inférieure ou égale à ε qui contient E. Ainsi, un point est un ensemble de mesure nulle.

On peut démontrer que la réunion d'une infinité dénombrable d'ensembles de mesure nulle est aussi de mesure nulle. Ainsi, l'ensemble des nombres rationnels est de mesure nulle.

L'expression **presque partout** signifie sauf sur un ensemble de mesure nulle.

On dit qu'une fonction est **mesurable** si elle est presque partout la limite d'une suite de fonctions continues.

Si f et g sont mesurables, $f + g$, $f.g$, f/g (pour $g \neq 0$ partout) sont mesurables.

Les fonctions sommables sont par définition celles pour lesquelles on peut définir un nombre qui est leur intégrale de Lebesgue ; elles constituent une généralisation des fonctions pour lesquelles on a pu définir l'intégrale ordinaire de Riemann.

Si f est une fonction intégrable au sens de Riemann sur un intervalle [a, b] fini et nulle à l'extérieur, son intégrale de Lebesgue coïncide avec son intégrale de Riemann.

Chapitre 5 : Systèmes de fonctions itérées (IFS)

Le but de ce chapitre est de présenter la théorie de base des systèmes de fonctions itérées (ou itératives) (IFS). Nous nous focaliserons sur les IFS constitués de transformations affines et sur les fractales IFS qui en résultent. De telles fractales IFS peuvent être utilisées pour approximer les images du monde réel, en particulier parce qu'elles peuvent être contrôlées pour se rapprocher visuellement des éléments constitutifs d'images, à l'aide du théorème de collage.

On rencontrera ici les métriques de Hausdorff et de Hutchinson, souvent utilisées dans les démonstrations pour leurs bonnes propriétés de convergence, mais d'autres pourront évidemment convenir.

Comme pour le chapitre précédent, la démonstration des théorèmes énoncés peut se trouver dans [FE], en particulier pour le théorème d'IFS et le théorème de collage.

I. L'espace de Hausdorff

Définition : Soit (E, d) un espace métrique complet. Alors, **H(E)** désigne l'espace dont les points sont les sous-ensembles compacts de E, autres que l'ensemble vide.

Définition : Soit (E, d) un espace métrique complet, $x \in E$ et $B \in H(E)$. Définissons

$$d(x, B) = \min\{d(x, y) : y \in B\}$$

comme étant la **distance d'un point** x à **l'ensemble** B.

Définition : Soit (E, d) un espace métrique complet. La **distance de Hausdorff** entre les points A et B de H(E) est définie par

$$h(A, B) = \max\{d(A, B), d(B, A)\}.$$

On nomme aussi h la **métrique de Hausdorff** sur H.

Théorème : Soit (E, d) un espace métrique complet. Alors, H(E, h) est un espace métrique complet. De plus, si $\{A_n \in H(E) : n = 1, 2, 3, ...\}$ est une suite de Cauchy, alors

$$A = \lim_{n \to \infty} A_n \in H(E)$$

peut être caractérisée par

$$A = \{x \in E : \exists \text{ suite de Cauchy } \{x_n \in A_n\} \text{ convergeant vers } x\}.$$

Soit E un espace métrique. On désignera par (H(E), h(d)) l'espace de Hausdorff associé, avec la métrique de Hausdorff h(d) décrite précédemment. On utilise la notation h(d) pour montrer que d est la métrique sous-jacente à la métrique de Hausdorff h.

II. Systèmes de fonctions itérées, attracteurs

Définition : Un **système de fonctions itérées (ou système de fonctions itératives)** est constitué d'un espace métrique complet (E, d) et d'un ensemble fini de transformations contactives $w_n : E \to E$, de facteurs de contraction respectifs s_n, pour n = 1, 2, 3, ..., N. La notation pour ce système de fonctions itérées est $\{E ; w_n, n = 1, 2, 3, ..., N\}$ et son **facteur de contraction** est $s = \max \{s_n : n = 1, 2, 3, ..., N\}$.

On utilisera préférentiellement la notation IFS, « Iterated Functions System », pour désigner un système de fonctions itérées.

Théorème *(Le théorème d'IFS)* : Soit $\{E ; w_n, n = 1, 2, 3, ..., N\}$ un IFS de facteur de contraction s. Alors, la transformation $W : H(E) \to H(E)$ définie par

$$W(B) = \bigcup_{n=1}^{N} w_n(B),$$

pour chaque $B \in H(E)$, est une transformation contractive sur l'espace métrique complet (H(E), h(d)) de facteur de contraction s, c.-à-d.

$$h(W(B), W(C)) \leq s.h(B,C), \forall B, C \in H(E).$$

Elle possède un seul point fixe, $A \in H(E)$, qui vérifie

$$A = W(A) = \bigcup_{n=1}^{N} w_n(A),$$

et est donnée par $A = \lim_{n\to\infty} W^{\circ n}(B)$ pour chaque $B \in H(E)$.

Définition : Le point fixe $A \in H(E)$ décrit dans le théorème d'IFS est appelé l'**attracteur** de l'IFS.

Considérons l'IFS $\{\Re^2 ; w_1, w_2, w_3\}$, où les w_i sont les transformations affines de $\Re^2$ suivantes :

$$w_1 = \begin{pmatrix} 0{,}5 & 0 \\ 0 & 0{,}5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

$$w_2 = \begin{pmatrix} 0,5 & 0 \\ 0 & 0,5 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 10 \\ 0 \end{pmatrix},$$

$$w_3 = \begin{pmatrix} 0,5 & 0 \\ 0 & 0,5 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 6 \\ 6 \end{pmatrix}.$$

L'attracteur de cet IFS est ce que l'on appelle un **triangle de Sierpinski** (Voir figure 5.1), de sommets (0, 0), (20, 0), (12, 12). Les différentes étapes de construction sont illustrées en annexe C.

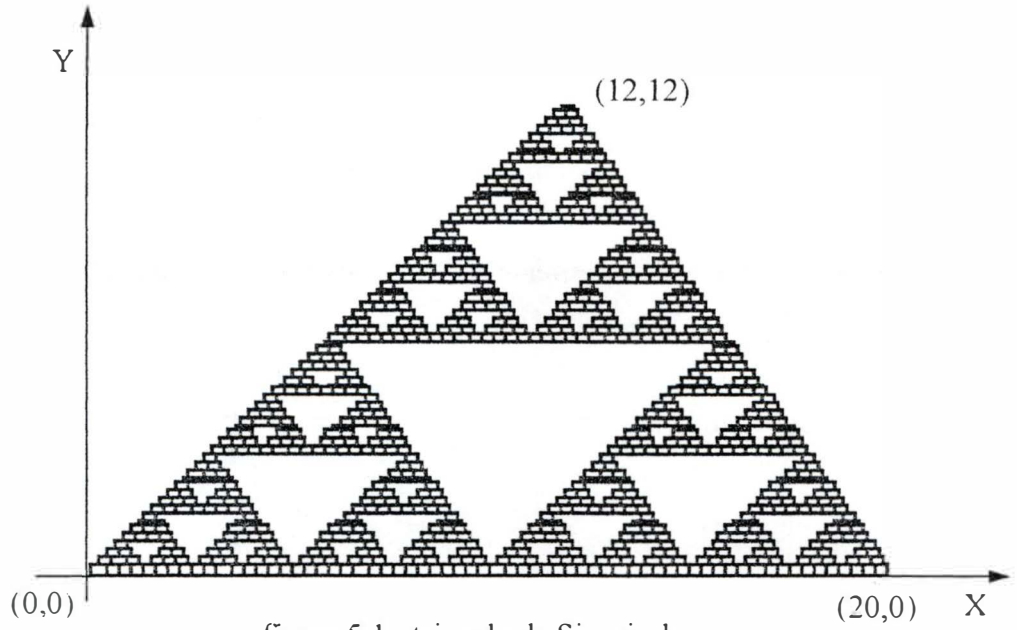


figure 5.1 : triangle de Sierpinsky

La notation matricielle d'un IFS de transformations affines est assez peu pratique. Donnons-en une plus conviviale. Ecrivons :

$$w_i(\vec{x}) = w_i\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ c_i & d_i \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e_i \\ f_i \end{pmatrix} = A_i\vec{x} + \vec{t}_i,$$

alors, l'IFS $\{\Re^2 ; w_1, w_2, w_3\}$ peut se représenter à l'aide du tableau 5.2 suivant :

w	a	b	c	d	e	f	p
1	0,5	0	0	0,5	0	0	0,34
2	0,5	0	0	0,5	10	0	0,33
3	0,5	0	0	0,5	6	6	0,33

tableau 5.2 : l'IFS $\{\Re^2 ; w_1, w_2, w_3\}$

On y remarque, en dernière colonne, la présence d'un nombre p_i, associé à la transformation w_i, pour i = 1, 2, 3. Ces nombres sont des probabilités. Dans le cas le plus général d'un IFS $\{\Re^2 ; w_i, i = 1, 2, ..., N\}$, on trouvera N nombres p_i qui vérifient :

$$p_1 + p_2 + p_3 + ... + P_N = 1 \text{ et } p_i > 0 \text{ pour } i = 1, 2, ..., N.$$

Ces nombres jouent un rôle important dans le calcul de l'attracteur d'un IFS en utilisant l'algorithme de la photocopieuse en niveaux de gris que nous décrirons.

III. Calcul de l'attracteur d'un IFS : algorithme de la photocopieuse (MRCM)

Le théorème d'IFS nous donne une méthode simple pour trouver l'attracteur d'un IFS. Soit $\{\Re^2 ; w_i, i = 1, 2, ..., N\}$ un IFS. On part d'un ensemble compact $A_0 \subset \Re^2$. On calcule ensuite successivement $A_n = w^{on}(A)$ selon la procédure récursive :

$$A_{n+1} = \bigcup_{k=1}^{N} w_k(A_k), \text{ pour n = 1, 2, ...,}$$

on construit donc une suite $\{A_n : n = 1, 2, 3, ...\} \subset H(\Re^2)$ qui converge vers l'attracteur de l'IFS dans la métrique de Hausdorff. Ceci fournit un procédé pour le calcul de l'attracteur d'un IFS par approximations successives. On prend donc une image tout à fait quelconque, à laquelle on fait subir chacune des N transformations affines pour donner une deuxième image. On répète ce processus en remplaçant l'image de départ par celle que l'on vient d'obtenir. La différence entre deux images successives calculées par ce processus devient de plus en plus petite : on obtient une suite convergente d'images. L'image finale reste inchangée lorsqu'elle subit à nouveau la procédure : elle représente l'attracteur. On trouvera en annexe C le code source illustrant cette démarche pour quelques attracteurs connus.

IV. Le théorème de collage

Le théorème suivant est essentiel pour la conception d'IFS's dont les attracteurs devront être proches d'ensembles donnés.

Théorème : Soit (E, d) un espace métrique complet. Soient $T \in H(E)$ et $\varepsilon \geq 0$ donnés. Choisissons un IFS $\{E ; w_1, w_2, ..., w_N\}$ de facteur de contraction $0 \leq s < 1$ de telle façon que :

$$h\left(T, \bigcup_{n=1}^{N} w_n(T)\right) \leq \varepsilon,$$

où h(d) est la métrique de Hausdorff. Alors

$$h(T, A) \leq \varepsilon/(1 - s),$$

où A est l'attracteur de l'IFS.

Ce théorème de collage nous dit que, pour trouver un IFS dont l'attracteur « ressemble » à un ensemble donné, on doit chercher un ensemble de transformations contractives dont l'union, ou le collage, des images obtenues par l'application de ces transformations à l'ensemble donné, produit une image « proche » de l'ensemble donné. Le degré de « ressemblance » entre les deux images est mesuré en utilisant la métrique de Hausdorff.

V. IFS avec probabilités pour les images en tons de gris

Une description plus intuitive des propos qui constituent cette partie sera donnée dans la partie suivante.

Définition : Un **système de fonctions itérées avec probabilités** est constitué d'un IFS $\{E ; w_1, w_2, ..., w_N\}$ et d'un ensemble ordonné de nombres $\{p_1, p_2, ..., p_N\}$ tels que :

$$p_1 + p_2 + p_3 + ... + P_N = 1 \text{ et } p_i > 0 \text{ pour } i = 1, 2, ..., N.$$

La probabilité p_i est associée à la transformation w_i.

Dans la suite de ce chapitre, nous ne nous intéresserons qu'au cas où E est l'ensemble $\mathfrak{I} \subset \mathfrak{R}^2$.

Définition : Soit μ une mesure de Borel sur $\mathfrak{I} \subset \mathfrak{R}^2$. Si $\mu(\mathfrak{I}) = 1$, μ est dite **normalisée**.

Définition : Soit P l'ensemble des mesures de Borel normalisées sur $\mathfrak{I}$. La **métrique de Hutchinson d_H** sur P est définie par

$$d_H(\mu, \nu) = \sup\{ \int_{\mathfrak{I}} f \, d\mu - \int_{\mathfrak{I}} f \, d\nu : \forall \mu, \nu \in P$$
$$f : \mathfrak{I} \rightarrow \mathfrak{R} \text{ est continue et vérifie } | f(x) - f(y) | \leq d(x, y), \forall x, y \in \mathfrak{I} \}$$

Théorème : Soit P l'ensemble des mesures de Borel normalisées sur $\mathfrak{I}$ et soit d_H la métrique de Hutchinson. Alors (P, d_H) est un espace métrique complet.

Soit B les sous-ensembles de Borel de $\mathfrak{I}$ et soit $w : \mathfrak{I} \rightarrow \mathfrak{I}$ continue. On peut alors prouver que w^{-1} va de B dans B. Il s'ensuit que, si ν est une mesure de Borel normalisée sur $\mathfrak{I}$, il en est de même pour $\nu o w^{-1}$. Cela implique que l'opérateur défini ci-après va de P dans P.

Définition : Soit $\{\mathfrak{I} ; w_1, w_2, ..., w_N ; p_1, p_2, ..., p_N\}$ un IFS avec probabilités. L'**opérateur de Markov** associé à l'IFS est la fonction $M : P \rightarrow P$ définie par

$$M(\nu) = p_1.\nu \, o \, w_1^{-1} + p_2.\nu \, o \, w_2^{-1} + ... + p_N.\nu \, o \, w_N^{-1}$$

pour tout $\nu \in P$.

Théorème (théorème de Hutchinson) : Soit $M : P \rightarrow P$ l'opérateur de Markov associé à un IFS avec probabilités, où chaque transformation possède un facteur de contraction $0 \leq s < 1$. Alors M est une transformation contractive, de facteur de contraction s, par rapport à la métrique de Hutchinson sur P ; c.-à-d.,

$$d_H(M(\mu), M(\nu)) \leq s.d_H(\mu, \nu), \forall\ \mu, \nu \in P.$$

En particulier, il n'existe qu'une seule mesure $\mu \in P$, telle que $M\mu = \mu$. De plus, si $\nu \in P$, alors

$$\lim_{n\rightarrow\infty} M^{on}(\nu) = \mu,$$

où la convergence s'effectue par rapport à la métrique de Hutchinson.

Définition : Soit μ le point fixe de l'opérateur de Markov, promis par le théorème de Hutchinson, μ est appelée la **mesure invariante** de l'IFS avec probabilités.

Théorème : Soit $\{\Im ; w_1, w_2, ...,w_N ; p_1, p_2, ...,p_N\}$ un IFS avec probabilités. Soit μ la mesure invariante associée. Alors, le support de μ est l'attracteur de l'IFS $\{\Im ; w_1, w_2, ...,w_N\}$.

Théorème (Théorème de collage pour les mesures) : Soit $\{\Im ; w_1, w_2, ...,w_N ; p_1, p_2, ...,p_N\}$ un IFS avec probabilités. Soit μ la mesure invariante associée. Soit $0 \leq s < 1$ le facteur de contraction de l'IFS. Soit $M : P(\Im) \rightarrow P(\Im)$ l'opérateur de Markov associé. Alors

$$d_H(\mu, \nu) \leq \frac{d_H(\nu, M(\nu))}{(1-s)}.$$

VI. Algorithme de la photocopieuse en tons de gris

Cette section va décrire de façon intuitive, en termes d'images, en quoi consistent les mesures de Borel sur $\Im$. On va décrire, ici, l'algorithme de la photocopieuse en niveaux de gris. Cette procédure fonctionne à l'aide d'une photocopieuse conceptuelle peu commune, qui peut traiter des images en niveaux de gris en plus des images noir et blanc.

Considérons une photographie en niveaux de gris sur $\Im$. En une seconde, l'image réfléchit une certaine quantité de lumière ; nous supposerons que le nombre de photons réfléchis par seconde est unitaire. Selon le caractère plus ou moins sombre des différentes zones de l'image, les photons seront renvoyés à différents taux.: les régions foncées réfléchiront relativement peu de photons par seconde, au contraire des régions plus claires. Considérons une surface S de l'image. La mesure $\mu(S)$ va représenter le nombre total de photons réfléchis par la surface S en une seconde. L'avantage d'utiliser la théorie des mesures, et pas celle des fonctions, pour les images en tons de gris, est de pouvoir considérer

des sous-ensembles de l'image et pas seulement des points. Notons que pour une région S' de même superficie que S, qui comporterait plus de régions sombres, il est tout à fait possible que $\mu(S') = \mu(S)$, si les zones claires de S' renvoient plus de photons que celles de S. Donc , les mesures assignent des valeurs à des ensembles. Les sous-ensembles auxquels les mesures assignent des valeurs sont restreints aux ensembles de Borel.

La photographie est représentée par $\mu \in P$, où P est l'ensemble de toutes les mesures de Borel normalisées sur $\Im$. Le nombre total de photons réfléchis par $\Im$ étant unitaire, on a $\mu(\Im) = 1$. Une mesure de Borel normalisée peut aussi être vue comme une densité de probabilités : on peut dire qu'un photon émis par l'image est pris au hasard sur la surface de l'image, alors $\mu(S)$ est la probabilité que le photon ait été émis par la surface S de $\Im$. Le fait que $\mu(\Im) = 1$ indique que le photon arrive bien d'un endroit de l'image, et pas d'ailleurs.

La photocopieuse en niveaux de gris possède N systèmes de lentilles, accompagnés chacun de leur filtre. Elle modélise un système de fonctions itérées avec probabilités $\{\Im ; w_i ; p_i, i = 1, 2, ..., N\}$. Une image en niveaux de gris est mise à l'entrée de la photocopieuse, où elle est éclairée, cet éclairage pouvant être contôlé ; cette image correspond à une mesure $\nu \in P$. La lumière est réfléchie par les différentes portions de l'image en fonction de leur blancheur. Cette lumière est ensuite captée par les N systèmes de lentilles, qui appliquent chacun une des N transformations affines w_i, i = 1, 2, ..., N, qui peuvent être sélectionnées et paramétrées à l'aide de N boutons de contrôle. Ces N transformations affines de l'image d'entrée vont être projetées en sortie sur une feuille de papier photographique spécial, après être passées à travers les filtres accollés à chaque système de lentilles, et qui permettent de régler l'intensité lumineuse. Chaque filtre atténue donc la lumière qui le traverse par un facteur p_i. L'image qui tombe sur le papier photographique est telle que dans les régions, résultats de transformations affines, qui se superposent, la brillance s'additionne. La brillance totale de l'image de sortie est ajustée de manière que le nombre total de photons émis par cette image en une seconde est le même que celui de l'image d'entrée. L'image de sortie est notée $M(\nu)$. La mécanique que l'on vient de décrire constitue une implémentation physique de l'opérateur de Markov.

Décrivons maintenant l'algorithme de la photocopieuse en niveaux de gris. On ajuste d'abord correctement les différents paramètres de la machine selon les spécifications de l'IFS avec probabilités. On place en entrée de la machine une image arbitraire $\nu \in P$, qui après opération de la photocopieuse, donne une image de sortie $M(\nu)$. On prend cette image résultat comme nouvelle image d'entrée de la machine, qui après photocopie en niveaux de gris, donnera une image de sortie $M^{\circ 2}(\nu)$. On continue ainsi le processus itérativement. Du fait que M est une transformation affine contractive sur l'espace métrique complet P, la suite d'images $M^{\circ 1}(\nu)$, $M^{\circ 2}(\nu)$, $M^{\circ 3}(\nu)$, $M^{\circ 4}(\nu)$, ... converge vers l'unique image en niveaux de gris ou mesure μ telle que $M(\mu) = \mu$, le point fixe de l'opérateur de Markov, c'est-à-dire la mesure invariante de l'IFS avec probabilités. En pratique, on obtiendrait une suite d'images qui changent de moins en moins : deux images successives de la suite se ressemblent de plus en plus au fur et à mesure que l'on avance dans la suite, la suite d'images se rapproche de plus en plus vers une image qui ne change plus si l'on continue les itérations de l'algorithme. Cette image finale est invariante sous la photocopieuse en niveaux de gris, c'est-à-dire sous l'opérateur de Markov. On notera que l'image invariante μ ne dépend pas de l'image de départ ν.

Cet algorithme peut évidemment être implémenté sur un ordinateur à écran graphique, pour permetre de suivre les différentes étapes correspondant aux applications successives de

l'opérateur de Markov, et c'est exactement ce processus qui est utilisé dans le programme de décompression de l'annexe D, où l'on trouvera, par ailleurs, une illustration sur des images réelles. La résolution des images digitales étant finie, on pourra arrêter le processus lorsque deux images digitales de la suite seront identiques, de plus, le résultat de l'application de l'algorithme est une approximation, dépendant de la résolution, de la mesure invariante de l'attracteur de l'IFS avec probabilités. Le théorème de collage pour les mesures nous assure que les erreurs introduites suite à la discrétisation des images tendent vers zéro lorsque la résolution tend vers l'infini ; ces erreurs sont donc contrôlables.

Chapitre 6 : La transformation fractale

Le présent chapitre concerne la théorie des transformations fractales pour son application à la compression d'images. La théorie des transformations fractales, inventée par M. Barnsley en 1986, traite des systèmes locaux de fonctions itérées (IFS locaux). A la différence des IFS déjà évoqués, les IFS locaux ont des domaines restreints à des sous-ensembles de l'espace. Une transformation fractale d'une image sera constituée de codes IFS et de domaines de transformation.

I. IFS locaux

Définition : Soit (E, d) un espace métrique compact. Soit R un sous-ensemble non vide de E. Soit $w : R \rightarrow E$ et soit s un nombre réel tel que $0 \leq s < 1$. Si

$$d(w(x), w(y)) \leq s.d(x, y) \; \forall \; x, y \in E,$$

alors w est une **transformation contractive locale** sur (E,d). Le nombre s est le **facteur de contraction** pour w.

Définition : Soit (E, d) un espace métrique compact. Soient R_i des sous-ensembles non vides de E, pour i allant de 1 à N, N entier positif. Soit $w_i : R_i \rightarrow E$ une transformation contractive locale sur (E, d) de facteur de contraction s_i, pour i = 1, 2, ... N. Alors, $\{w_i : R_i \rightarrow E : i = 1, 2, ... N\}$ est appelé un **système local de fonctions itérées (LIFS ou IFS local) ou système de fonctions itérées partiel (PIFS).** Le nombre $s = \max\{s_i : i = 1, 2, ... N\}$ est le **facteur de contraction** de l'IFS local.

Les IFS locaux peuvent être utilisés pour définir des opérateurs de contraction sur l'espace des images. Considérons l'exemple suivant. Soit S l'ensemble de tous les sous-ensembles de E. Définissons l'opérateur $W_{local} : S \rightarrow S$ de la façon suivante :

$$W_{local}(B) = \bigcup_{i=1}^{N} w_i(R_i \cap B), \; \forall B \subset S.$$

Définition : On dit qu'un sous-ensemble non vide A de E est un **attracteur** ou **ensemble invariant** de l'IFS local si $W_{local}(A) = A$.

Un IFS local peut n'avoir aucun attracteur, il peut aussi en avoir plusieurs. Si A et B sont deux attracteurs, il en est de même pour $A \cup B$, c'est-à-dire que s'il existe au moins un attracteur, il en existe un qui est le plus grand : il contient tous les autres. On obtient cet attracteur en prenant l'union de tous les attracteurs de W_{local}. C'est celui-là que nous considérerons quand nous parlerons de l'attracteur de W_{local}.

Soit $\{w_i : R_i \rightarrow E : i = 1, 2, ... N\}$ un IFS local où les R_i sont compacts. On peut définir une suite de sous-ensembles compacts de E, $\{A_n : n = 0, 1, 2, 3, ...\}$, par

$$A_0 = E,$$

$$A_n = \bigcup_{i=1}^{N} w_i(R_i \cap A_{n-1}) \text{ pour } n = 1, 2, 3, ...$$

On vérifie que $A_0 \supset A_1 \supset A_2 \supset A_3 \supset ...$ C'est-à-dire que $\{A_n : n = 0, 1, 2, 3, ...\}$ est une suite décroissante d'ensembles compacts. En particulier, il existe un ensemble compact $A \subset E$ tel que

$$\lim_{n\rightarrow\infty} A_n = A$$

et

$$A = \bigcup_{i=1}^{N} w_i(R_i \cap A) = W_{local}(A).$$

S'il n'est pas vide, A est un attracteur, en fait l'**attracteur maximal**, de l'IFS local.

II. Le théorème de collage pour les IFS locaux

Bien que l'opérateur de contraction n'agisse pas sur l'entièreté du support de l'image, il est quand même possible de traiter les IFS locaux comme s'il s'agissait d'IFS ordinaires. Essayons de voir ce qu'il se passe quand on applique à ces IFS locaux le théorème de collage déjà rencontré avec les IFS ordinaires.

Soit une image binaire en noir et blanc, représentée par un sous-ensemble G de $\Im$. Pour créer un modèle fractal de G, partitionnons $\Im$ à l'aide de petits carrés. On notera D_i, i = 1, 2, 3, ..., N ceux qui ont une intersection commune avec G. Pour chaque i, on cherche une transformation affine locale $w_i : R_i \rightarrow \Im$, de facteur de contraction s, telle que

$$w_i(R_i) = D_i$$

et

$$w_i(R_i \cap G) \approx D_i \cap G.$$

On peut imposer comme critère d'approximation pour l'égalité que la distance de Hausdorff entre $w_i(R_i \cap G)$ et $D_i \cap G$ est petite, c'est-à-dire

$$h(w_i(R_i \cap G), D_i \cap G) < \varepsilon, \text{ pour } i = 1, 2, ..., N \text{ et } \varepsilon \text{ choisi petit.}$$

En pratique, la transformation peut être choisie de la forme $w(\bullet) = 0{,}5.A.\bullet + t$, où $A : \Im \rightarrow \Im$ est une des transformations symétriques affines du tableau 6.1 suivant, et t une translation.

Matrice	Description
$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	Identité
$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$	Réflexion sur l'axe y
$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	Réflexion sur l'axe x
$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$	Rotation de +180°
$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	Réflexion sur la droite y = x
$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$	Rotation de +90°
$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$	Rotation de +270°
$\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$	Réflexion sur la droite y = -x

tableau 6.1 : transformations affines symétriques

On peur choisir les R_i comme des petits blocs de taille double de D_i. Si tous les blocs D_i ont la même taille, pour i entier allant de 1 à N, l'IFS local est complètement spécifié en fournissant pour chaque i les coordonnées (D_x, D_y) du coin inférieur droit de D_i, les coordonnées (R_x, R_y) du coin inférieur droit de R_i, et un nombre entier indiquant le choix de la transformation affine A parmi les huit possibilités. Le résultat est appelé un **code d'IFS local** (figure 6.1).

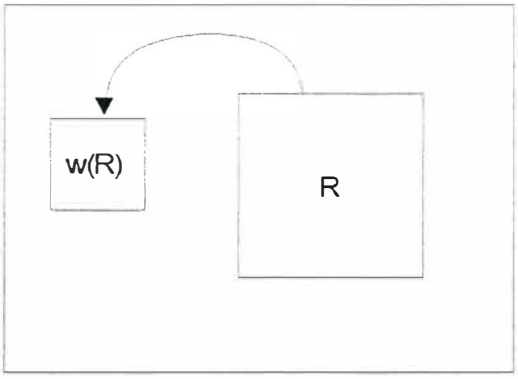


figure 6.1 : une région R et son image w(R) dans un IFS local

III. La transformation fractale en noir et blanc

Nous allons ici décrire une méthode générale de compression d'images binaires par fractales. Elle est constituée de cinq étapes.

I. Entrer une image binaire G, un sous-ensemble de $\mathfrak{I} \subset \mathfrak{R}^2$.

II. Recouvrir G à l'aide de blocs D_i carrés, appelés **blocs domaine** (domain blocks). Recouvrir signifie que l'ensemble des blocs domaine $\{D_i : i = 1, 2, ..., N\}$ doit couvrir entièrement G, et que ces blocs ne se superposent pas.

III. Introduire une série de blocs appelés **blocs portée** (range blocks) $R \subset \mathfrak{I}$, chacun tel que $R \cap G \neq \varnothing$. Il s'agit de blocs carrés dont la longueur des côtés est double de la longueur des côtés des blocs domaines. Les coordonnées possibles (R_x, R_y) du coin inférieur gauche de chaque bloc portée sont confinées à l'intérieur d'un ensemble fini L. On définit T, une collection de transformations affines contractives locales, transformant les blocs portée R en blocs domaine D_i. C'est-à-dire, pour i entier allant de 1 à N,

$$T_i = \{w(D_i, R_x, R_y, j) : (R_x, R_y) \in L ; j = 1, 2, ...,8\},$$

où $w(D_i, R_x, R_y, j)$ est la transformation affine contractive de domaine R d'ensemble d'arrivée D_i, de la forme $0,5.A.\bullet + t$, et A(j) désigne la $j^{ème}$ transformation symétrique de la table précédente.

IV. Exécuter le processus de transformation fractale suivant. Pour chaque i, choisir $w_i \in T_i$ pour minimiser la distance de Hausdorff

$$h(w_i(R \cap G), D_i \cap G).$$

C'est-à-dire que pour chaque bloc domaine, on choisit un bloc portée et une transformation symétrique, de telle façon que la partie transformée de l'image dans le bloc portée ressemble le plus à la partie de l'image dans le bloc domaine. L'ensemble

$$w_{local}(G) = \cup\, w_i(R \cap G)$$

est appelé le **collage** de l'image G correspondant à l'IFS local. Le nombre

$$h(w_i(R \cap G), D_i \cap G)$$

est l'**erreur de collage** correspondante.

V. Ecrire les données sous forme d'un code d'IFS local.

VI. Eventuellement, appliquer un algorithme de compression sans perte sur ces codes.

Pour décompresser l'image, on peut appliquer l'algorithme de la photocopieuse.

IV. La transformation fractale en tons de gris

On va maintenant mettre sur pied une structure d'IFS local appropriée à une méthode automatique de compression d'images par fractales. On va utiliser comme modèle pour l'ensemble des images du monde réel U, celui qui consiste en l'ensemble V de toutes les fonctions à valeur réelle $\phi : \mathfrak{I} \rightarrow I$, où $I = [a, b] \subset \mathfrak{R}$ est l'intervalle qui représente les valeurs possibles d'intensités de l'image, tel l'intervalle [0, 255]. C'est un modèle du type U_2 décrit au chapitre 2.

Pour transformer V en espace métrique complet, on définit une distance entre deux fonctions ϕ_1 et $\phi_2 \in V$ de la façon suivante

$$d(\phi_1, \phi_2) = \sup\{ | \phi_1(x, y) - \phi_2(x, y) | : (x, y) \in \mathfrak{I} \}.$$

La métrique d ainsi définie est la métrique l^∞, elle n'est pas le plus adéquate mais elle permet l'élaboration d'un théorème général de convergence. Expérimentalement, la transformation fractale montre de bonnes propriétés de convergence par rapport à d'autres métriques plus naturelles, que nous appliquerons en annexe D, mais alors les théorèmes de convergence sont plus compliqués.

Désignons par P une **partition** de $\mathfrak{I}$, constituée d'une collection d'ensembles finis $P_i \subset \mathfrak{I}$, i = 1, 2, ..., M ; c'est-à-dire

$$\mathfrak{I} = \bigcup_{i=1}^{M} P_i ,$$

où $P_i \cap P_j = \varnothing$ pour $i \neq j$. Pour chaque i, soit $f_i : P_i \rightarrow \mathfrak{I}$, avec $f(P_i) = R_i$, et soit $v_i : \mathfrak{R} \rightarrow \mathfrak{R}$ une transformation affine contractive de facteur de contraction s, avec $0 \leq s < 1$. C'est-à-dire

$$| v_i(r_1) - v_i(r_2) | < s.| r_1 - r_2 |, \forall r_1, r_2 \in \mathfrak{R},$$

pour i = 1, 2, ..., M.

On définit alors $T : V \rightarrow V$ par

$$T(\phi)(x, y) = v_i(\phi(f_i(x, y))), (x, y) \in P_i.$$

On appellera T l'**opérateur de transformation fractale**, et on dit que s est le **facteur de contraction** deT.

Théorème : (Convergence des transformations fractales) Soit l'espace métrique complet (V, d) et l'opérateur $T : V \rightarrow V$ défini comme précédemment. Alors, T est une transformation contractive affine sur V ; c'est-à-dire, pour tout $\phi_1, \phi_2 \in V$, on a

$$d(T(\phi_1), T(\phi_2)) \leq s.d(\phi_1, \phi_2),$$

où s est le facteur de contraction de T.

Nous avons maintenant tous les ingrédients pour un système de compression fractale : un modèle pour l'espace des images du monde réel, une métrique sur cet espace et un opérateur de contraction sur cet espace. En particulier, il existe une fonction unique $\phi \in V$ telle que $T(\phi) = \phi$. La fonction ϕ est appelée l'**attracteur de la transformation fractale**. Pour calculer cette fonction ϕ, on peut utiliser la propriété que si une fonction $\varphi \in V$, le résultat de l'application successive de T à φ converge uniformément vers l'attracteur ϕ ; c'est-à-dire que l'on a

$$\lim_{n\to\infty} T^{\circ n}(\varphi) = \phi,$$

de plus, on a l'estimation d'erreur

$$| T^{\circ n}(\varphi)(x, y) - \phi(x, y) | \leq s^n . | b - a |, \forall (x, y) \in \Im, \text{ où } I = [a, b].$$

La distance entre $\varphi \in V$ et l'attracteur de l'opérateur de transformation fractale T est bornée :

$$d(\phi, \varphi) \leq \frac{d(\varphi, T(\varphi))}{1 - s}.$$

C'est le **théorème de collage** pour l'opérateur de transformation fractale T.

Chapitre 7 : Compression fractale

On sait maintenant que le principe de compression par fractales est de considérer que l'image est le point fixe d'un IFS à déterminer. Plus précisément, on découpe l'image en segments D_i, et l'on cherche, pour chaque D_i, un segment R_i plus grand et une transformation affine contractive T_i tels que $T_i(R_i)$ soit à peu près égal à D_i, selon une métrique donnée. Ce principe laisse beaucoup de choix possibles quant à la métrique à adopter (euclidienne, Hutchinson, rms, ...), au choix de la technique de segmentation, ...

On trouvera en annexe la description complète d'un petit algorithme de compression / décompression fractale d'une image BMP en niveaux de gris.

I. Choix des segments

A. Partitions de taille fixe

Un choix simple pour le partitionnement de l'image à compresser est de considérer des blocs portées D_i de taille fixe. L'ennui avec ce choix est que certaines régions de l'image assez étendues (ciel sans nuage) possèdent peu de variations, et accepteraient donc volontiers des segments de grandes dimensions, réduisant ainsi le nombre de transformations w_i nécessaires et diminuant encore la taille de l'image compressée ; à l'inverse, d'autres régions présentent beaucoup de variations sur une petite étendue, requérant, pour des résultats convenables, des dimensions de blocs plus petites. Choisir d'emblée des petits blocs fonctionne évidemment, mais cela augmente alors le nombre de transformations et la taille du fichier compressé.

B. Partitions en quadr-arbre

Un choix plus judicieux est une partition en quadr-arbre (quad-tree) de l'image. Dans une telle partition, une image carrée est subdivisée en quatre sous images carrées toutes de même taille. Chacune de ces quatre images peut être subdivisée de la même façon récursivement.

Voyons une idée d'algorithme qui va appliquer ce partitionnement à une image de 256 x 256 pixels. On choisit comme domaines possibles D_i, les sous-régions carrées de l'image de taille 8, 12, 16, 24, 32, 48 et 64 pixels. On partitionne l'image par quadr-arbre jusqu'à obtenir des carrés de taille 32. Pour chaque domaine carré de la partition, on essaie de le couvrir par une transformation affine d'un bloc portée plus large. Si une tolérance d'approximation, dépendant de la métrique utilisée, fixée d'avance est atteinte, alors on

appelle le carré D_i et le bloc portée associé R_i. Sinon, on subdivise le bloc portée en quatre et on répète le procédé.

C. Partitions H-V

La faiblesse de la partition en quadr-arbre est qu'elle requiert un grand nombre de blocs portée, pour avoir une chance de rencontrer une bonne approximation d'un bloc parmi les domaines : aucun effort n'est fait au départ pour construire des blocs portée ayant d'emblée un « bon » contenu. Une façon de remédier à cela est d'utiliser une partition H-V. Dans une telle partition, une image rectangulaire est partitionnée de manière récursive horizontalement ou verticalement pour donner deux nouvelles images. Ce schéma est plus flexible, étant donné que la position de la partition est variable. On peut alors essayer de faire en sorte que les partitions possèdent des structures auto-similaires. Par exemple, on peut essayer d'arranger les partitions pour que les contours dans l'image forment une diagonale de partition. On aura ainsi plus de chances de trouver des partitions larges dont les transformations vont recouvrir des partitions plus petites. La figure 7.1 illustre cette idée. En (a) est montrée une partie d'une image ; en (b), cette image est partitionnée en deux rectangles R_1 et R_2, R_2 a comme diagonale un contour ; en (c) et (d), les partitionnements 2, 3 et 4 donnent quatre nouveaux rectangles : deux d'entre eux peuvent être couverts par R_2 (parce qu'ils ont un contour comme diagonale), et les deux autres peuvent être couverts par R_1 (car ils ne contiennent pas de contour).

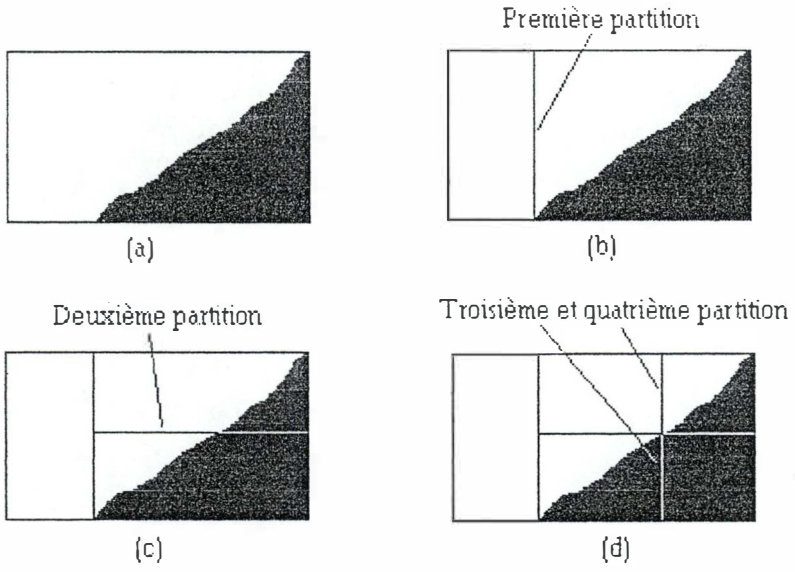


figure 7.1 : partitionnement H-V

D. Partitions par classification des zones

Les recherches les plus prometteuses sur le partitionnement se situent au niveau de l'analyse de l'image par des techniques qui permettent une classification des zones selon leur nature, en guise de prétraitement. Par exemple, voici quelques classes possibles :

- *fond* : ce type de zone est caractérisé par une faible variance de niveaux de gris sur tous les points du bloc. Si cette variance est inférieure à un seuil prédéfini, ce bloc appartient à la classe des fonds.

- *dégradé* : ce type de zone est caractérisé par une variation constante des niveaux de gris selon un plan.

- *contour* : ce type de zone est caractérisé par une nette séparation de deux régions par une courbe de niveau de gris plus ou moins constante.

- *texture* : si un bloc est inclassable comme fond, contour ou dégradé, il est classé comme texture

Une fois les blocs catégorisés, l'appariement des blocs D et R ne se fait qu'entre blocs de même classe. On notera que cette façon de procéder réduit le temps de calcul.

II. Compression fractale de Dudbridge

D.M. Monroe et F. Dudbridge (voir [MD1] et [MD2]) ont implémenté une méthode différente pour la compression d'images basée sur les IFS avec probabilités. Nous allons en brosser rapidement le principe.

La méthode consiste à considérer chaque bloc de l'image comme une image à part entière. Par exemple, une image 256 x 256 pixels en niveaux de gris est divisée en blocs de 8 x 8 pixels ; chaque bloc est représenté par un IFS avec probabilités. Dans ce cas, $\mathfrak{I}$ désigne un bloc image de dimensions doubles, et la petite image correspondante est représentée par l'IFS

$$\{\mathfrak{I} ; w_1, w_2, w_3, w_4 ; p_1, p_2, p_3, p_4\}.$$

Les ensembles d'arrivée de l'application des transformations affines sur $\mathfrak{I}$ divisent le bloc $\mathfrak{I}$ en quatre parties égales, comme montré sur la figure 7.2 .

$W_1(\mathfrak{I})$	$W_2(\mathfrak{I})$
$W_3(\mathfrak{I})$	$W_4(\mathfrak{I})$

figure 7.2 partitionnement de $\mathfrak{I}$

Les valeurs de niveaux de gris du bloc image sont représentées par une mesure μ. Alors on peut calculer les **moments**

$$M(m, n) = \int_{\Im} x^m.y^n \, d\mu(x, y)$$

pour tout enier m, n $\in$ {0, 1, 2, 3, ...}. Désignons par μ' la mesure invariante de l'IFS. Monroe et Dudbridge montrent qu'il est alors possible de calculer explicitement des formules pour les moments de μ' en terme des coefficients et des probabilités de l'IFS. En fait, chacun des nombres

$$M'(m, n) = \int_{\Im} x^m.y^n \, d\mu'(x, y)$$

est linéaire par rapport aux probabilités. Il s'ensuit que l'on peut résoudre l'ensemble d'équations

$$M(m, n) / M(0, 0) = M'(m, n),$$

avec des indices m et n appropriés, pour conduire à des valeurs explicites des coefficents de w_1, w_2, w_3, w_4 et des probabilités p_1, p_2, p_3 et p_4. On obtient donc une approximation fractale du bloc de l'image.

III. La décompression

On a vu au chapitre 5, avec l'algorithme de la photocopieuse (MRCM), que le théorème d'IFS nous fournissait une première méthode pour restituer l'image, attracteur de l'IFS, à partir des codes IFS. Cette démarche est illustrée en annexe D.

Cependant, il existe une deuxième façon de procéder pour reconstruire l'image à partir des codes IFS fournis lors de la compression, il s'agit de l'**algorithme d'itération aléatoire (Random Iteration Algorithm)**. Supposons que l'IFS soit constitué de n transformations affines w_1, w_2, ..., w_n avec probabilités p_1, p_2, ..., p_n, alors l'algorithme suivant va générer en sortie une image qui est l'attracteur de l'IFS :

```
initialisation : x ← 0 et y ← 0;
Pour i allant de 1 à IMAX (≈ 5000), faire
{
  choisir aléatoirement k ∈ {1, ..., n} de probabilité pk;
  calculer (x', y') = wk(x, y);
  (x, y) ← (x', y');
  si i > 10, afficher (x, y);
};
```

Augmenter le nombre d'itérations i ajoute des points à l'image. Les quelques premiers points calculés ne sont pas dessinés, pour laisser le point aléatoire (x, y) « s'installer » à l'intérieur de l'image. C'est comme lancer une balle sur un plateau de kicker où jouent des professionnels : tant que la balle n'est sous le contrôle d'aucun des joueurs, son mouvement est imprévisible, ou du moins incontôlable par les joueurs ; mais dès que ce

ballon est capté par quelqu'un, son mouvement n'est plus dépendant que de l'habileté des joueurs. Le fait que les transformations affines soient contractives garantit que la balle ne sort pas de la surface de jeu.

La supposition selon laquelle l'image rendue est indépendante de la suite de points générés, et donc qu'elle sera toujours la même à chaque application de l'algorithme, a d'abord été suggérée accidentellement par diverses expériences de mathématiques sur ordinateur, avant de recevoir un fondement rigoureux par le mathématicien John Elton.

IV. La compression des images en couleurs

L'algorithme de la photocopieuse (MRCM) est bien adapté aux images en niveaux de gris, car les différentes nuances peuvent se superposer et « s'additionner ». Prenons cependant garde qu'au blanc est associé le nombre 255, tandis que le noir correspond à 0, et non l'inverse ; les tons s'additionnent donc seulement de façon visuelle.

Les images en couleurs peuvent cependant, elles aussi, bénéficier des techniques de compression fractale. Pour cela, il suffirait d'appliquer les mêmes principes de compression que pour les images en tons de gris, mais séparément, à chaque composante de couleur fondamentale (R, V, B), dont les différentes nuances peuvent subir l'algorithme de la photocopieuse. C'est malheureusement une technique « bête et méchante », qui n'utilise aucunement les liens qui doivent exister entre les composantes (en scindant l'image en ses trois composantes de couleurs, l'image originale est toujours reconnaissable dans les trois images obtenues).

Un cas particulier intéressant sont les images en 256 niveaux de couleurs. On pourrait appliquer la méthode décrite dans le paragraphe précédent pour les compresser, mais cela conduirait certainement à une taille triple de la même image repésentée en tons de gris et compressée. D'où, l'idée de convertir cette image de départ en 256 niveaux de gris (correspondance non biunivoque). Cela se fait aisément en donnant une notion de « clair-foncé » par le calcul de la somme des carrés des 3 composantes de couleurs de chaque pixel. Pour ne pas perdre les données relatives aux couleurs de l'image d'origine, il suffirait alors de conserver une table de correspondance couleur-ton de gris dans le fichier compressé. La palette de couleurs d'origine serait ainsi restituée à la décompression.

Il est bien difficile, ici, de dire quelle méthode est effectivement appliquée dans les logiciels de compression d'images commerciaux utilisant la transformation fractale, car, si la littérature ne manque pas de références en la matière, les auteurs ne donnent jamais que des idées générales et se gardent bien de divulguer leurs arcanes. Cela est d'autant plus vrai que les spécialistes du domaine (M. Barnsley, A. Jacquin, ...) ont fondé « Iterated Systems », la firme qui commercialise ces logiciels et dépose les brevets sur la transformation et la compression fractale.

V. Performances

Le gros problème des algorithmes de compression d'images par fractales est le **temps de calcul** pour obtenir le meilleur collage : plusieurs heures pour des images courantes sur des machines performantes. Si N est le nombre de pixels de l'image, la décompression se fait en un temps polynomial (k.N), tandis que la compression requiert un temps quartique ($k.N^4$). Bien que les techniques soient paramétrables, on cherche évidemment encore à réduire le temps d'exploration des blocs, tout en conservant un bon taux de compression et une qualité quasi égale lors de la reconstruction. Une façon de réduire ce temps de codage consiste à simplifier la nature de la transformation géométrique, par exemple en se limitant à des opérations simples : symétrie par rapport à un axe, rotation d'un angle multiple de $\pi/2$, ... On peut aussi comme déjà mentionné, utiliser des techniques de codage de « textures » pour rendre plus rapide la recherche de l'appariement des blocs D et R.

L'évaluation du **taux de compression** peut se faire si l'on connait le nombre B(w) de bits servant à exprimer bijectivement une transformation w de l'IFS, le nombre C de niveaux de l'image originale, sa résolution X x Y pixels et le nombre N de blocs de la partition R :

$$taux = (\log_2(C).X.Y) / (B(w).N)$$

Par exemple, avec une image en 256 niveaux de gris (8 bits), de résolution 256 x 256, en considérant que 32 bits suffisent à décrire w, pour 8 transformées géométriques possibles, et un nombre de blocs R de 4 x 4 pixels égal à 64^2, on obtient un taux de

$$taux = (\log_2(2^8).256.256) / (32.64^2) = 4,$$

c'est à dire relativement faible pour une compression avec perte d'information. Mais on doit remarquer que les coefficients de w peuvent encore être compressés efficacement par un algorithme de compression sans perte de type Huffman. D'autre part, il semble qu'on utilise souvent un partitionnement quad-tree de l'image, ce qui diminue le nombre de transformations à coder et augmente le taux de compression.

On retiendra qu'à l'heure actuelle, les taux moyens de compression (source [XM]) obtenus avec les codages fractals les plus performants sont de l'ordre de :

- 8 à 11 pour les images en niveaux de gris ;
- 25 à 30 pour les images en couleur.

Il faut cependant regarder les valeurs mentionnées dans les publicités avec un oeil critique. Une image compressée par fractales peut, à la restitution, montrer plus de détails que l'image originale ; cela permet, lors d'un zoom, d'éviter **l'effet « gros pixels »**. Cependant, les détails créés n'existent pas forcément, et ce nombre de pixels supplémentaires pourrait être utilisé pour justifier des taux de compression supérieurs.

Bibliographie

[FE] M. Barnsley, « Factals Everywhere », *Academic Press*, Boston (1988).

[BS] M. Barnsley et A. Sloane, « A Better Way to Compress Images », *Byte*, Volume 13, pp 215-223 (janvier 1988).

[JS] J.R. Storer, « Data Compression : Methods and Theory », *Computer Science Press*, Rockville, MD (1988).

[MB] M. Barnsley, « Lecture Notes on Iterated Function Systems », *Proceedings of Syposia in Applied Mathematics*, Volume 39 (1989).

[VR] E.R. Vrscay et C.J. Roerhig, « Iterated Function Systems and the Inverse Problem of Fractal Construction Using Moments », *Computers and Mathematics*, pp 250-259, Springer-Verlag, New York (1989).

[JB] J.M. Beaumont, « Image Data Compression Using Fractal Techniques », *B.T. Technology Journal*, Volume 9, N°4, pp 93-108 (1991).

[MN] M. Nelson, « The Data Compression Book », *M&T Books*, Redwood City, CA (1991).

[CF] H.-O. Peitgen, H. Jürgens, D. Saupe, « *Chaos and Fractals : New Frontiers of Science* », Springer-Verlag (1991).

[AJ] A. Jacquin, « Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformation », *IEEE Transaction on Image Processing*, 1 pp18-30 (1992).

[BH] M. Barnsley et L. Hund, « Fractal Image Compression », AK Peters Ltd, Wellesley USA (1992)

[MD1] D.M. Monroe et F. Dudbridge, « Fractal Blocks Coding of Images », *Electronics Letters*, Volume 28, N°11, pp 1053-1055 (mai 1992).

[MD2] D.M. Monroe et F. Dudbridge, « Fractal Approximation of Image Blocks », *Multidimensionnal Signal Processing*, Volume 3, (1992).

[JW] J. Waite, « A Review of Iterated Function System Theory for Image Compression », Preprint, *British Telecom Research Laboratories*, Martelsham Heath, U.K. (1992).

[JC] J. Corrigan, « Computer Graphics Secrets and Solutions », *Sybex*, U.S.A. (1994).

[XM] X. Marsault « Compression et Cryptage des Données Multimédias », Hermès, Paris (1995).

[LS] J. Lammi, T. Sarjakoski, « Image Compression by the JPEG Algorithm », *Photogrammetric Engineering & Remote Sensing*, Volume 61, N°10, pp 1261-1266 (1995).

Contenu des annexes

Annexe A : Le format de fichier BMP

Puisque c'est ce format d'image BMP non compressé que va utiliser le programme de compression/décompression d'images de l'annexe D, cette partie va le décrire aussi précisément que possible. En annexe B se trouve un petit listing illustrant l'affichage d'un tel fichier image.

I. Introduction

Un fichier BMP peut schématiquement se décomposer en quatre parties successives. La première, appelée "BITMAPFILEHEADER" fournit des informations générales sur le fichier.

La deuxième, "BITMAPINFOHEADER", contient essentiellement des informations sur l'image proprement dite (dimensions, couleurs, ...).

Si les couleurs ne sont pas codées en absolu sur 24 bits (16 millions de couleurs), la troisième partie fait correspondre à chaque code de couleur présent dans le fichier une valeur absolue sur 24 bits.

Enfin, la dernière partie, "BITMAP", rassemble le codage complet de chacun des pixels de l'image.

Ces quatre parties sont détaillées dans le texte qui suit, un tableau récapitulatif apparaîtra en fin d'annexe.

On remarquera que la plupart des informations sont stockées sous forme d'un mot de 32 bits, afin de mieux coller à l'architecture 32 bits des processeurs 386, 486 et pentiums, et d'utiliser au mieux leurs registres internes.

II. L'en-tête du fichier : BITMAPFILEHEADER

Les fichiers BMP sont identifiables par leurs deux caractères ASCII "BM" qui occupent les deux premiers octets. La taille totale du fichier est ensuite inscrite sur quatre octets, autorisant ainsi une taille maximale de quatre gigaoctets. Viennent ensuite quatre octets réservés à zéro, puis la valeur de l'offset de la partie BITMAP à partir du début de fichier. Cet en-tête de fichier est de longueur fixe : 14 octets.

III. Les informations sur l'image : BITMAPINFOHEADER

La première valeur, de 4 octets, indique la taille consacrée à la partie BITMAPINFOHEADER. Elle est fixée à 40 octets pour un BMP Windows 3.0, mais peut évoluer dans le futur. Ensuite viennent la largeur et la hauteur de l'image exprimées en pixels. Après deux octets réservés à un, on en trouve deux autres mentionnant le nombre de bits nécessaires pour représenter un pixel. Quatre valeurs peuvent en fait être utilisées : 1 pour les images monochromes, 4 pour les images en 16 couleurs, 8 pour les images en 256 couleurs, et 24 pour les images en couleurs réelles. Six données de 4 octets terminent cette deuxième partie. Ces données peuvent être initialisées à 0 (ce qui est d'ailleurs souvent rencontré). Dans le cas où il s'agit d'un fichier 16 ou 256 couleurs, les données BITMAP peuvent être compressées par un algorithme RLE, c'est ce que mentionne la première donnée. Cette possibilité est cependant rarement utilisée et les quatre octets sont mis à zéro. La deuxième donnée est en fait redondante, car elle indique la taille de l'image, qui peut être obtenue à partir des dimensions de de celle-ci et du nombre de bits par pixel. Les deux informations suivantes sont les résolutions horizontale et verticale en pixels par mètre. Elles donnent la possibilité à une application de choisir parmi plusieurs BMP celui qui convient le mieux à un périphérique de sortie fixé. Les deux dernières données sont le nombre de couleurs utilisées (parmi la panoplie offerte) et le nombre de couleurs considérées comme importantes. Elles peuvent faciliter le transcodage effectué par un programme d'affichage de fichiers BMP, quand le BMP a une résolution de couleurs plus importante que le dispositif d'affichage présent.

IV. Les informations sur les couleurs utilisées

Un codage visuellement satisfaisant des couleurs impose 256 couleurs sur chacune des composantes élémentaires RVB, donnant 16 millions de couleurs possibles. Dans la plupart des cas, en fonction de la taille mémoire présente sur la carte vidéo, celle-ci permet, selon la résolution de l'écran, l'affichage simultané de 16, 256 ou 64K couleurs parmi une palette plus importante. Lorsque le fichier BMP est créé en utilisant une certaine palette, il est nécessaire, afin d'obtenir une image indépendante du périphérique d'affichage, de retraduire les index de couleurs utilisés en "valeurs absolues" codées sur 24 bits. Quatre octets seront utilisés, car il est plus facile de manipuler des mots de 32 bits. Le premier octet spécifie la composante bleue, le deuxième le vert, le troisième le rouge et le dernier octet doit être laissé à zéro. Le fichier BMP comprend donc une liste variable de 0, 2, 16 ou 256 couleurs (dépendant du nombre de bits utilisés pour coder chaque pixel) décrites chacune par 4 octets.

Lorsque le nombre de couleurs utilisées et le nombre de couleurs considérées comme importantes sont mentionnées (différentes de 0), la table des couleurs doit être ordonnée en plaçant les couleurs utilisées au début, classées par ordre d'importance.

V. Données BITMAP et compression

Chaque pixel de l'image est représenté par 1, 4, 8 ou 24 bits. L'origine est définie comme étant le coin inférieur gauche de l'image et les pixels sont analysés ligne par ligne, de la gauche vers la droite. Tous les bits représentant les pixels d'une même ligne sont concaténés entre eux. Cependant, la représentation d'une ligne entière devra, suivant les cas, être complétée avec des bits à zéro afin de toujours occuper un nombre entier de mots de 32 bits.

Windows supporte une compression des images BMP si celles-ci sont en 16 ou 256 couleurs. Il utilise un encodage par plages ("run length") efficace quand de nombreux pixels consécutifs ont la même couleur. L'idée de base consiste à coder sur deux octets, le premier représentant le nombre de pixels identiques, le second spécifiant leur couleur.

De plus, lorsque le premier octet est mis à 0, le second indique une action spéciale : 0 pour fin de ligne, 1 pour fin de bitmap, 2 pour indiquer un déplacement relatif décrit par les deux octets suivants et 3 à FFh pour indiquer une séquence absolue non compressée de longueur variant de 3 à 255 octets. Cette séquence doit toujours être alignée sur un nombre d'octets pair.

A titre d'exemple, la séquence 03 04 00 03 45 56 67 00 02 78 00 02 05 01 02 78 00 00 04 1E 00 01 donnerait les valeurs non compressées suivantes :

04 04 04	trois fois la valeur 04 (03 04)
45 56 67	séquence absolue de trois octets (plus 00 d'alignement) (00 03 45 56 67 00)
78 78	deux fois la valeur 78 (02 78)
-	déplacement de 5 pixels vers la droite et de 1 pixel vers le bas (00 02 05 01)
78 78	deux fois la valeur 78 (02 78)
-	fin de ligne (00 00)
1E 1E 1E 1E	quatre fois la valeur 1E (04 1E)
-	fin de fichier bitmap (00 01)

Les fichiers bitmap compressés portent généralement l'extension RLE (Run Length Encoded)

Remarque

Les fichiers BMP définis sous OS/2 sont légèrememnt différents de ceux définis sous Windows 3.0. La partie BITMAPINFOHEADER n'inclut que les cinq premières valeurs, et les dimensions de l'image sont stockées sur deux octets au lieu de quatre. La longueur du bloc comprend donc 12 octets au lieu de 40. De surcroît, les informations sur les couleurs utilisées ne sont plus données sur quatre octets, mais seulemnt sur les trois octets réellement utiles.

VI. Récapitulatif

En-tête du fichier : BITMAPFILEHEADER, total : 14 octets

2 octets : bfType	Identification du fichier.
4 octets : bfSize	Longueur totale du fichier.
4 octets : bfReserved	Zone réservée et mise à zéro.
4 octets : bfOffBits	Offset des données bitmap.

Informations sur l'image : BITMAPINFOHEADER, total : 40 octets

4 octets : biSize	Taille occupée par BITMAPFILEHEADER.
4 octets : biWidth	Largeur de l'image exprimée en pixels.
4 octets : biHeight	Hauteur de l'image exprimée en pixels.
2 octets : biPlanes	Nombre de surfaces de l'image (toujours égal à un).
2 octets : biBitCount	Nombre de bits de couleur (1, 4, 8 ou 24).
4 octets : biCompression	Méthode de compression (0 = non compressé).
4 octets : biSizeImage	Taille de l'image en octets.
4 octets : biXpelsPerMeter	Résolution horizontale en pixels par mètre.
4 octets : biYpelsPerMeter	Résolution verticale en pixels par mètre.
4 octets : biClrUsed	Nombre de couleurs utilisées (0 = toutes).
4 octets : biClrImportant	Nombre de couleurs importantes (0 = toutes).

Informations sur les couleurs utilisées

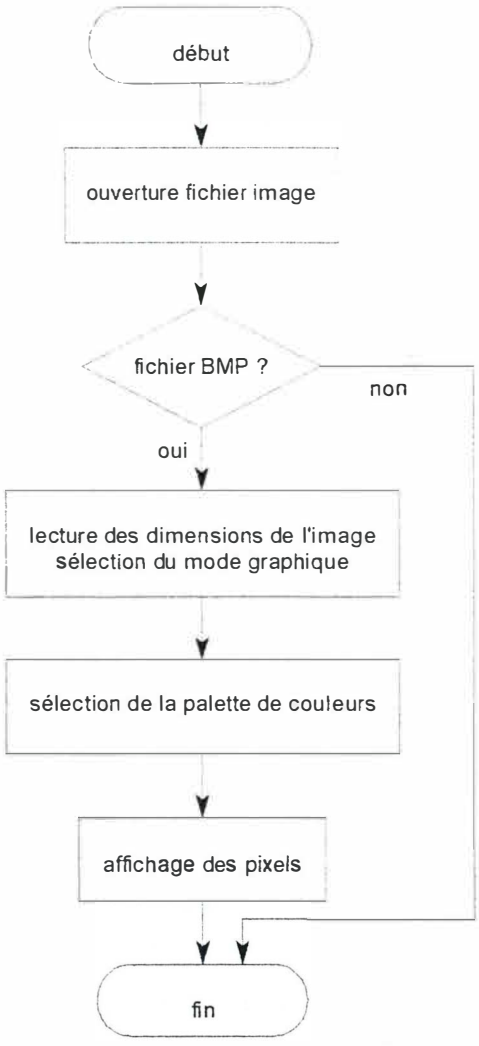
Elles ne sont présentes que si les pixels ne sont pas codés sur 24 bits (vraies couleurs).
Selon le cas, on trouve 2, 16 ou 256 groupes de 4 octets.

1 octet : spécifiant la composante bleue.
1 octet : spécifiant la composante verte.
1 octet : spécifiant la composante rouge.
1 octet : réservé à 0.

Annexe B : Affichage d'une image BMP 256 niveaux

Cette partie présente, comme illustration de l'annexe A, un petit programme d'affichage d'une image au format bitmap BMP non compressé 256 niveaux de couleurs.

I. Organigramme



II. Codes sources

Les sources en C tiennent en un seul fichier, sont très explicites et ne présentent aucune difficulté, si ce n'est que les routines graphiques du compilateur ne sont pas appelées ; à la place, on utilise les services d'interruptions. Au départ, cela avait été choisi ainsi pour garantir la portabilité d'un compilateur vers un autre (PowerC 2.01 et Borland C/C++ 4.5 ont été testés), ainsi que pour illustrer ce mécanisme de sélection et d'affichage vidéo. Ce choix n'étant pas franchement indispensable, l'annexe C présentera le code source d'un autre programme utilisant les bibliothèques graphiques propres au Borland C/C++ 4.5.

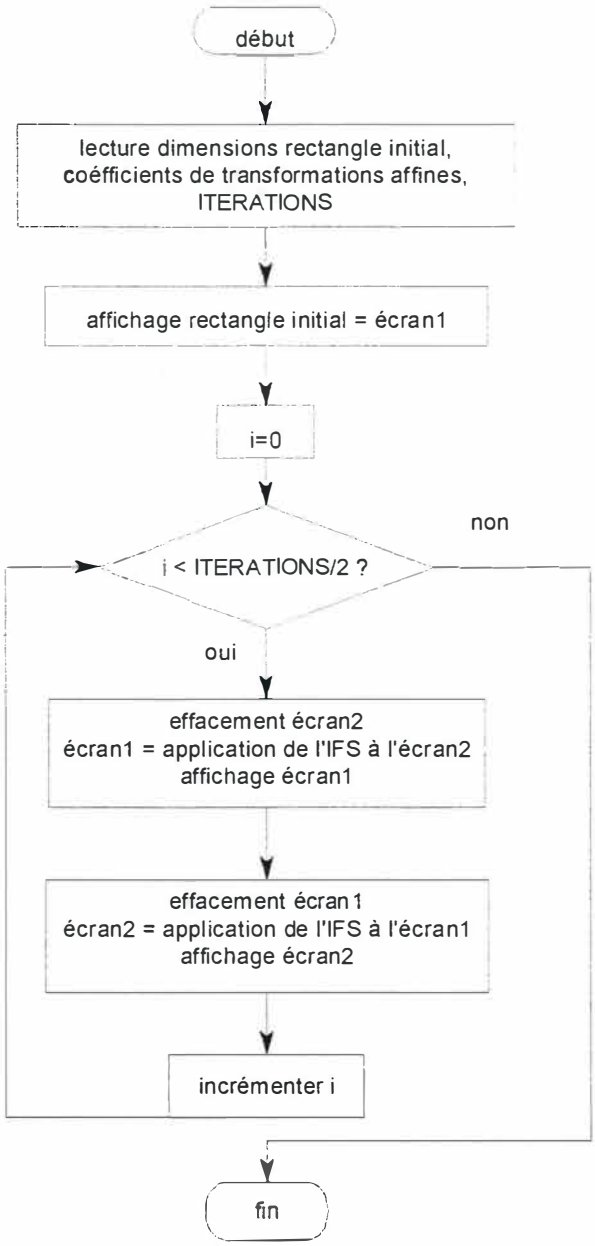
```
/**  fichier BMP256.CPP  **/

/*************************************************/
/*****   Affichage d'une image BMP 256 niveaux   *****/
/*****   M. LUPPI 1995                           *****/
/*************************************************/

/* Compilateur Borland C 4.5 */
/* Routines graphiques du compilateur non utilisées */

#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <bios.h>

FILE *f;
enum {true,false} test;
int xm,ym,xmax,ymax;
int mode;

/******************/
/****   main   ****/
/******************/
void main()
{
    char nom_fic[9];
    char extens[]=".bmp";

    void InitTexte(),TestImg(),InitMode(unsigned char),
         RectifPalette(),LitImage(),LectureDim();

    InitTexte();
    printf("Affichage d'une image BMP 256 niveaux de couleurs \n");
    printf("                                   M. LUPPI 1995 \n\n\n\n\n");
    printf("Entrez le nom du fichier BMP à visualiser (sans extension): ");
    scanf("%s",nom_fic);
    strncat(nom_fic,extens,5);
    f=fopen(nom_fic,"rb");
    if (f != NULL)
    { rewind(f);
      TestImg();
      if (test == true)
      { LectureDim();
        InitMode(mode);
        RectifPalette();
        LitImage();
        while(!kbhit());  /* rien tant que touche non pressée */
        InitTexte();
      } /* if (test == true) */
      else printf("Erreur, cette image n'est pas en 256 niveaux.");
    } /* if (f != NULL)  */
    else printf("Fichier inexistant.");
    fclose(f);
} /* fin main */

/************************************/
/****  Initialisation mode texte   ****/
/************************************/
void InitTexte()
{
    union REGS regs;

    /* interr 10h fct 0h : sélection et initialisation mode vidéo */
    regs.h.ah=0x00;
    regs.h.al=0x03;       /* caractères 80 colonnes * 25 lignes */
    int86(0x10,&regs,&regs);
} /* fin InitTexte */

/*********************************/
/****   Test du fichier BMP   ****/
/*********************************/
void TestImg()
{
  int i,j;

  /* Lecture des 2 premiers octets du fichier correspondant  */
  /* à l'identificateur du type d'image.                     */
  i=getc(f);
  j=getc(f);
  test=false;
  if(i==66 && j==77) test=true;
} /* fin TestImg */

/*********************************/
/****   Lecture dimensions   ****/
/*********************************/
void LectureDim()
{
  int i,pin,ps;

  for (i=1;i<=16;i++)  getc(f);

    /***  Dimension X  ***/
  pin=getc(f);
  ps=getc(f);
  xmax=ps*256+pin;
  for (i=1;i<=2;i++)  getc(f);

    /***  Dimension Y  ***/
  pin=getc(f);
  ps=getc(f);
  ymax=ps*256+pin;

    /***  Sélection automatique du mode graphique en fonction  ***/
    /***  des dimensions de l'image                            ***/
    /** Résolution 320x200 (VGA) **/
  mode=19;

    /** Résolution 640x480 (VESA) **/
  if (xmax>=321 || ymax>=201) mode=1;

    /** Résolution 800x600 (VESA) **/
  if (xmax>=641 || ymax>=481) mode=3;

    /** Résolution 1024x768 (VESA) **/
  if (xmax>=801 || ymax>=601) mode=5;
```

```
  for (i=1;i<=30;i++) getc(f);
}  /* fin LectureDim */

/*****************************************/
/***   Initialisation mode graphique   ****/
/*****************************************/
void InitMode(unsigned char valeur)
{
  union REGS regs;

//  unsigned char TesteMode();

  if (valeur == 19)
  {
    /* interr 10h fct 0h :
       sélection et initialisation mode vidéo VGA */
    regs.h.ah=0x00;
    regs.h.al=valeur;  /* valeur : amenée par main */
  }
  else
  {
    /* interr 10h fct 4Fh ss-fct 02h :
       sélection et initialisation mode vidéo VESA */
    regs.h.ah=0x4f;
    regs.h.al=0x02;
    regs.h.bh=0x01;
    regs.h.bl=valeur;  /* valeur : amenée par main */
  }
  int86(0x10,&regs,&regs);
} /* fin InitMode */

/*********************************/
/****   Lecture de la palette   ****/
/*********************************/
void RectifPalette()
{
    union REGS regs;
    int r,g,b,i;

    for (i=0;i<=255;i++)
    {
     /* interr 10h fct 10h ss-fct 10h
     définition du contenu de l'un des 256 registres de couleur
                        bx : numéro de registre de couleur (0-255)
                        ch : vert
                        cl : bleu
                        dh : rouge       */
      regs.h.ah=0x10;
      regs.h.al=0x10;
      regs.x.bx=i;
      b=getc(f)/4;
      g=getc(f)/4;
      r=getc(f)/4;
      getc(f);
      regs.h.ch=g;
      regs.h.cl=b;
      regs.h.dh=r;
      int86(0x10,&regs,&regs);
    } /* for */
} /* fin RectifPalette */

/***************************/
/****   Lecture image   ****/
/***************************/
void LitImage()
{
  int coul;
  int x,y;

  void Affiche(int, int, int);

  for(y=ymax;y>0;y--)
    { for(x=0;x<xmax;x++)
      { coul=getc(f);
        Affiche(x,y,coul);}}
} /* fin LitImage */

/*********************************/
/****   Affichage d'un point   ****/
/*********************************/
void Affiche(int xg, int yg, int co)
{
  union REGS regs;

  /* interr 10h, fct 0Ch : écriture d'un point graphique :
                  cx : colonne de l'écran,
                  dx : ligne de l'écran,
                  al : valeur de couleur */
  regs.h.ah=0x0c;
  regs.h.al=co;
  regs.x.cx=xg;
  regs.x.dx=yg;
  regs.h.bh=0x00;
  int86(0x10,&regs,&regs);
} /* fin Affiche */
```

Annexe C : Attracteur d'un IFS

Cette annexe a pour but de présenter un petit programme d'illustration de la construction de l'attracteur d'un système de fonctions itérées.

I. Organigramme



II. Codes sources

Le code en C est constitué de deux fichiers : IFS.H et IFS.CPP. Le premier permet de paramétrer entièrement les transformations affines (coefficientss a, b, c, d, e et f), le nombre d'itérations à appliquer et les dimensions du rectangle de départ ; attention cependant à la limitation de 64 Ko pour la réservation de place d'un tableau en C sous DOS.

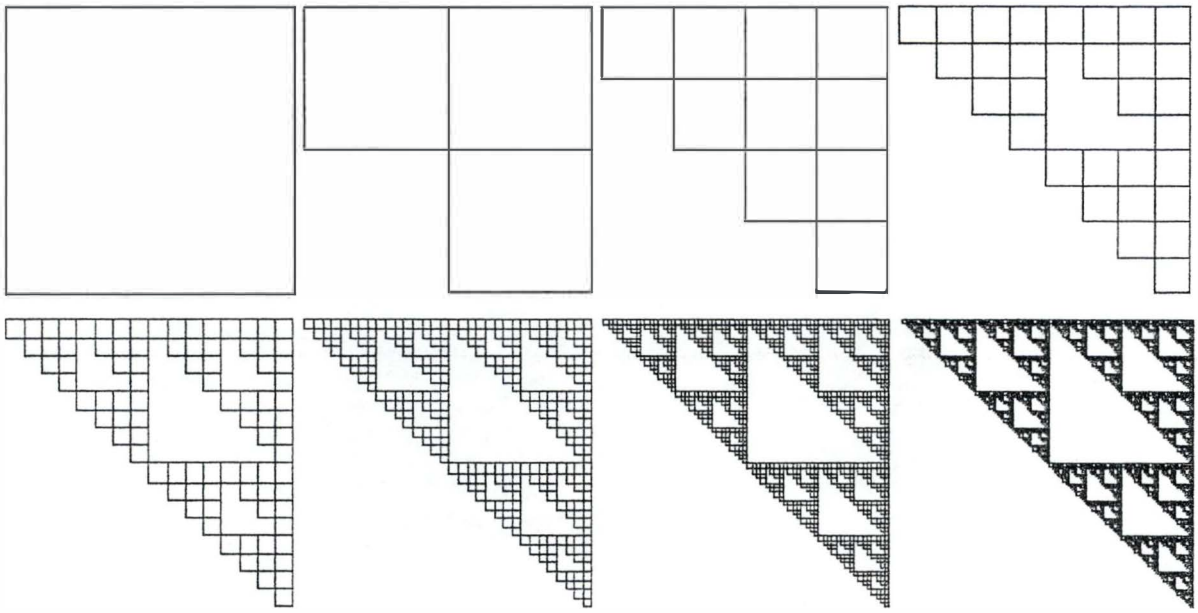
Le code est court et simple à suivre, il utilise pour l'affichage les fonctions offertes par le compilateur Borland C/C++ 4.5.

```
/**  fichier ifs.h  **/

#define ITERATIONS 15
#define LONG 150
#define HAUT 150

typedef unsigned char Pixel;
typedef Pixel Rectangle[LONG][HAUT];

void affiche_rect(Rectangle ecran);
void transfo(Rectangle ecran1, Rectangle ecran2);
void efface_rect(Rectangle ecran);

/**  Triangle de Sierpinski  **/
/*
#define TRANSFOS 3

static float a[TRANSFOS] = {0.5, 0.5, 0.5},
             b[TRANSFOS] = {0, 0, 0},
             c[TRANSFOS] = {0, 0, 0},
             d[TRANSFOS] = {0.5, 0.5, 0.5},
             e[TRANSFOS] = {0, 0.5*LONG, 0},
             f[TRANSFOS] = {0, 0, 0.5*HAUT};
*/

/**  Fougère de Barnsley  **/
/*
#define TRANSFOS 4

static float a[TRANSFOS] = {0, 0.849, 0.197, -0.15},
             b[TRANSFOS] = {0, 0.037, -0.226, 0.283},
             c[TRANSFOS] = {0, -0.037, 0.226, 0.26},
             d[TRANSFOS] = {0.16, 0.849, 0.197, 0.237},
             e[TRANSFOS] = {0.5*LONG, 0.075*LONG, 0.4*LONG, 0.575*LONG},
             f[TRANSFOS] = {0, 0.183*HAUT, 0.049*HAUT, -0.084*HAUT};
*/

/**  Brin d'herbe  **/
/*
#define TRANSFOS 3

static float a[TRANSFOS] = {-0.468, 0.387, 0.441},
             b[TRANSFOS] = {0.02, 0.43, -0.091},
             c[TRANSFOS] = {-0.113, 0.43, -0.009},
             d[TRANSFOS] = {0.015, -0.387, -0.322},
             e[TRANSFOS] = {0.4*LONG, 0.256*LONG, 0.4219*LONG},
             f[TRANSFOS] = {0.4*HAUT, 0.522*HAUT, 0.5059*HAUT};
*/

/**  Sapin de Noël  **/
/*  !! LONG = HAUT */
/*
#define TRANSFOS 3

static float a[TRANSFOS] = {0, 0, 0.5},
             b[TRANSFOS] = {-0.5, 0.5, 0},
             c[TRANSFOS] = {0.5, -0.5, 0},
             d[TRANSFOS] = {0, 0, 0.5},
             e[TRANSFOS] = {0.5*LONG, 0.5*LONG, 0.25*LONG},
             f[TRANSFOS] = {0, 0.5*HAUT, 0.5*HAUT};
*/

/**  Cristal à 6 branches  **/
/*  !! LONG = HAUT */
/*
#define TRANSFOS 4

static float a[TRANSFOS] = {0.255, 0.255, 0.255, 0.370},
             b[TRANSFOS] = {0, 0, 0, -0.642},
             c[TRANSFOS] = {0, 0, 0, 0.642},
             d[TRANSFOS] = {0.255, 0.255, 0.255, 0.370},
             e[TRANSFOS] = {0.3726*LONG, 0.1146*LONG, 0.6306*LONG,
                            0.6356*LONG},
             f[TRANSFOS] = {0.6714*HAUT, 0.2232*HAUT, 0.2232*HAUT,
                            -0.0061*HAUT};
/*

/**  Arbre **/
/*  !! LONG = HAUT */
/*
#define TRANSFOS 5

static float a[TRANSFOS] = {0.195, 0.462, -0.058, -0.035, -0.637},
             b[TRANSFOS] = {-0.488, 0.414, -0.070, 0.070, 0},
             c[TRANSFOS] = {0.344, -0.252, 0.453, -0.469, 0},
             d[TRANSFOS] = {0.443, 0.361, -0.111, -0.022, 0.501},
             e[TRANSFOS] = {0.4431*LONG, 0.2511*LONG, 0.5976*LONG,
                            0.4884*LONG, 0.8562*LONG},
             f[TRANSFOS] = {0.2452*HAUT, 0.5692*HAUT, 0.0969*HAUT,
                            0.5069*HAUT, 0.2513*HAUT};
*/

/**  Labyrinthe de Cantor  **/

#define TRANSFOS 3

static float a[TRANSFOS] = {0.336, 0, 0},
             b[TRANSFOS] = {0, 0.333, -0.333},
             c[TRANSFOS] = {0, 1, 1},
             d[TRANSFOS] = {0.335, 0, 0},
             e[TRANSFOS] = {0.333*LONG, 0.667*LONG, 0.333*LONG},
             f[TRANSFOS] = {0.667*HAUT, 0, 0};



/**  fichier ifs.cpp  **/

/*************************************************/
/*****     Calcul de l'attracteur d'un IFS     *****/
/*****     M. LUPPI 1995                       *****/
/*************************************************/

/**  compilateur Borland C 4.5  **/

#include <alloc.h>
#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include "ifs.h"
```

```
/****************/
/***   main   ***/
/****************/
main()
{
     Rectangle ecran1, ecran2;
     int i;
     int gdriver = DETECT, gmode, errorcode; /* vidéo */

     /* Initialisation routines graphiques */
     initgraph(&gdriver, &gmode, "");
     errorcode = graphresult();
     if(errorcode != grOk)
     {
        printf("Erreur d'initialisation graphique : %s\n",
                grapherrormsg(errorcode));
        exit(1);
     }

     /* Initialisation ecran1 */
     efface_rect(ecran1);

     /* rectangle initial */
     for (i = 0; i < LONG; i++)
        ecran1[i][0] =
        ecran1[i][HAUT - 1] = WHITE;
     for (i = 0; i < HAUT; i++)
        ecran1[0][i] =
        ecran1[LONG - 1][i] = WHITE;
     affiche_rect(ecran1);
     getch(); /* Pause */

     /* Boucle principale */
     for(i = 0; i < ITERATIONS / 2; i++)
     {
        efface_rect(ecran2);
        transfo(ecran1, ecran2);
        affiche_rect(ecran2);
        getch(); /* Pause */
        efface_rect(ecran1);
        transfo(ecran2, ecran1);
        affiche_rect(ecran1);
        getch(); /* Pause */
     }

     /* Fermeture routines graphiques */
     closegraph();
     return(0);
} /* fin main */


/*********************/
/***  efface_rect  ***/
/*********************/
void efface_rect(Rectangle ecran)
{
     int x, y;

     for(x = 0; x < LONG; x++)
        for(y = 0; y < HAUT; y++)
           ecran[x][y] = BLACK;
} /* fin efface_rect */


/**********************/
/***  affiche_rect  ***/
/**********************/
void affiche_rect(Rectangle ecran)
{
     int x, y;

     for(x = 0; x < LONG; x++)
        for(y = 0; y < HAUT; y++)
           putpixel(x, y, ecran[x][y]);
} /* fin affiche_rect */


/******************/
/***   transfo   ***/
/******************/
void transfo(Rectangle ecran_orig, Rectangle ecran_dest)
{
     int x, y, tr, t1, t2;

     for(x = 0; x < LONG; x++)
        for(y = 0; y < HAUT; y++)
        {
            if(ecran_orig[x][y] == WHITE)
            {
               for(tr = 0; tr < TRANSFOS; tr++)
               {
                 t1 = int(a[tr] * x + b[tr] * y + e[tr]) % LONG;
                 t2 = int(c[tr] * x + d[tr] * y + f[tr]) % HAUT;
                 ecran_dest[t1][t2] = WHITE;
               }
            }
        }
} /* fin transfo */
```

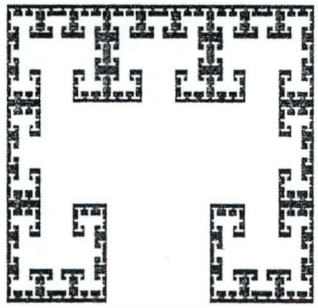
III. Exemples de sortie du programme

Le dessin suivant montre les étapes successives de construction d'un triangle de Sierpinski.



Les différentes figures qui suivent sont des copies écran de l'attracteur de divers IFS, dont on peut trouver les valeurs de paramètres dans le fichier IFS.H.

arbre

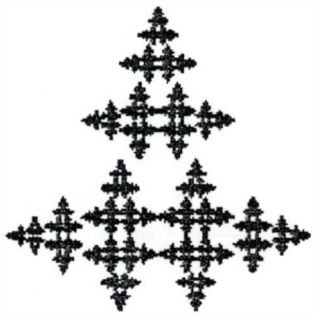


labyrinthe de Cantor

fougère de Barnsley

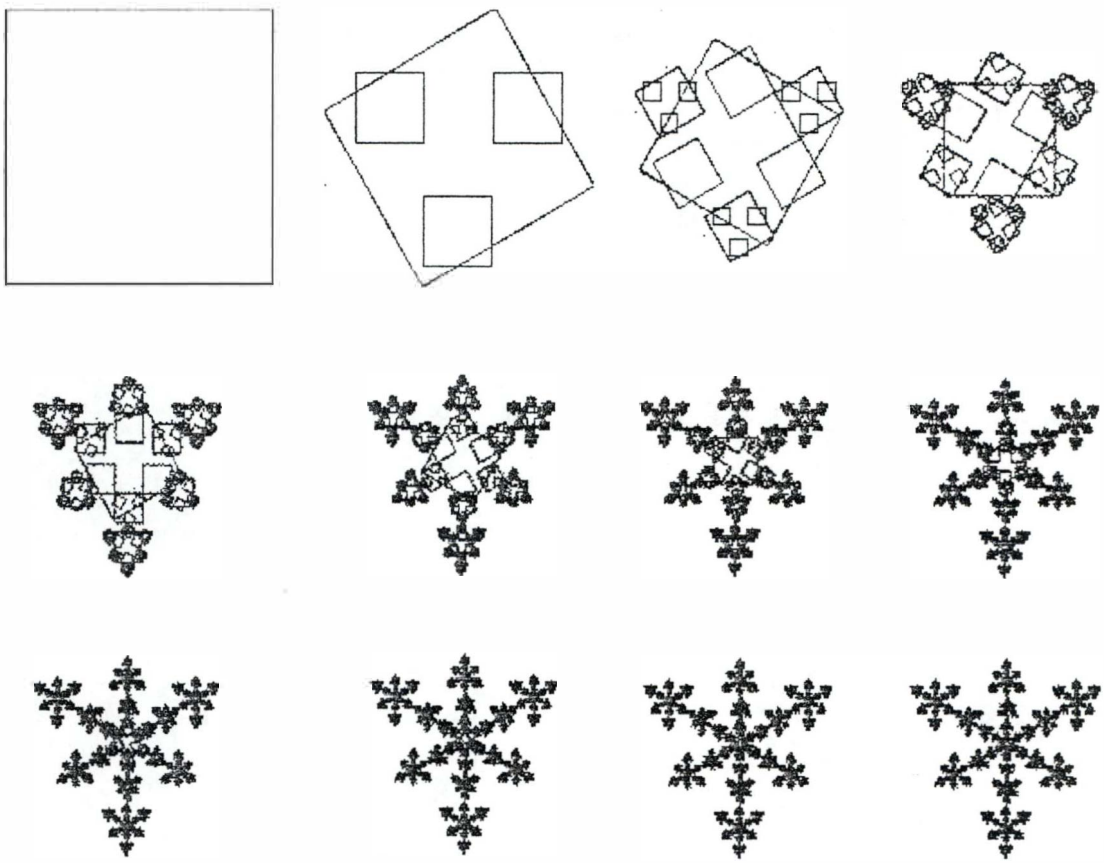
brin d'herbe

Annexes : attracteur d'un IFS



sapin de Noël

La dernière illustration présente les stades successifs de construction d'un cristal à six branches.



On remarquera le très faible nombre d'informations à posséder (6 coefficientss réels par transformation) pour aboutir à des images qui peuvent parfois présenter une très grande complexité, c'est là l'avantage du stockage d'une image sous forme d'IFS.

Annexe D : Exemple de compression / décompression d'images gray-scale par fractales

Cette dernière annexe a pour but d'appliquer les idées sur la compression fractale qui ont été developpées au cours de ce travail. L'algorithmique employée ici est inspirée de celle proposée dans [CF] ou [BH]. Il ne s'agit évidemment pas de concevoir une application ultra-sophistiquée et performante, capable de rivaliser avec les logiciels commerciaux existants, mais, malgré la relative simplicité de l'algorithme mis en oeuvre, on pourra ainsi se persuader que l'exposé que l'on a tenu jusqu'à présent tient la route et pas seulement sur papier.

I. Philosophie générale

La méthodologie générale de compression fractale a été exposée aux chapitres 6 et 7. Voyons quel est l'algorithme global qui a été implémenté.

A. Les segments domaine

La découpe en blocs domaine s'effectue de façon « statique » : elle est la même pour toute image. L'image est partitionnée, tel un damier, en blocs carrés de longueur fixée (on a pris arbitrairement 8), ce qui impose, dans notre cas, des dimensions de l'image multiples de 8. Le seul avantage de ce choix, mis à part la plus grande facilité de mise en oeuvre, est la non obligation de stocker la position et les dimensions des blocs domaine dans le fichier qui recevra les coefficients d'IFS.

B. Les segments portée

Les blocs portée vont avoir aussi des dimensions fixées : elles sont le double des dimensiions des blocs domaine, soit, pour nous, 16 pixels sur 16 pixels. Tous les blocs carrés de 16 sur 16 de l'image originale peuvent être des blocs portée. Cela signifie qu'il existe (H-15).(L-15) blocs portée possibles, où L et H sont repectivement la longueur et le hauteur de l'image. Le seul avantage de ce choix est de ne nécessiter que l'indication de position du bloc dans le fichier compressé, les dimensions étant implicites.

C. Les transformations affines contractives

La transformation affine d'un bloc carré de longueur 16 en un bloc carré de longueur 8 requiert d'abord une contraction d'un facteur 2, facilement implémentable : chaque pixel (i, j) du bloc portée transformé (ayant subi une transformation affine de contraction) sera calculé comme moyenne de quatre pixels ((2i, 2j), (2i + 1, 2j), (2i, 2j + 1), (2i +1, 2j +1)) du bloc portée initial. Ensuite, les transformations affines que l'on appliquera feront partie des huit symétries possibles d'un carré, détaillées dans le tableau 6.1.

D. La compression

L'algorithme traite successivement tous les blocs domaine. Pour chaque bloc domaine, on parcourt tous les blocs portée possibles, et pour chaque bloc portée, on applique les symétries permises. On calcule la distance entre ce bloc portée transformé et le bloc domaine. On conserve dans un fichier de sortie la position du bloc portée et la symétrie tels que la distance entre ce bloc transformé et le bloc domaine soit minimale.

L'étape finale d'un algorithme de compression fractale devrait être un codage statistique, mais comme tel n'était pas le but ici, nous ne l'implémenterons pas.

E. La décompression

On ne va pas appliquer l'algorithme d'itération aléatoire proposé au chapitre 7, mais bien ce bon vieux mécanisme de la photocopieuse en niveaux de gris, proposé au chapitre 4, ce qui nous permettra, en passant, de nous étonner de voir comment, à partir de n'importe quelle image choisie au départ pour la décompression, on peut reconstruire l'attracteur de l'IFS qu'était l'image originale. On part donc d'une image quelconque, dans laquelle on considère chaque bloc de dimension 16 x 16 dont la position est mentionnée dans le fichier compressé. On applique à ce bloc une transformation de contraction de facteur deux et les symétries renseignées dans le fichier compressé. Le bloc de 8 x 8 ainsi obtenu est ensuite translaté vers une position qui correspond à l'ordre d'apparition des données du bloc dans le fichier compressé : ainsi, l'image étant partitionnée en zones de 8 x 8, le bloc dont les coordonnées et les symétries apparaissent en $17^{ème}$ position dans le fichier compressé ira, après transformation, se loger en $17^{ème}$ place sur la grille de l'image. Une fois tous les blocs traités, on a accompli une itération de l'algorithme. On réitère le processus autant de fois que nécessaire, c'est-à-dire que l'on ne s'arrête que quand deux itérations succesives donnent la même image.

F. Quelques détails

Mentionnons maintenant quelques « astuces » mises en place pour permettre un gain de temps de calcul ou de compression.

- L'algorithme cherche la plus petite distance entre un bloc domaine et un bloc portée transformé, mais cette distance est évidemment bornée inférieurement par zéro, il ne faut

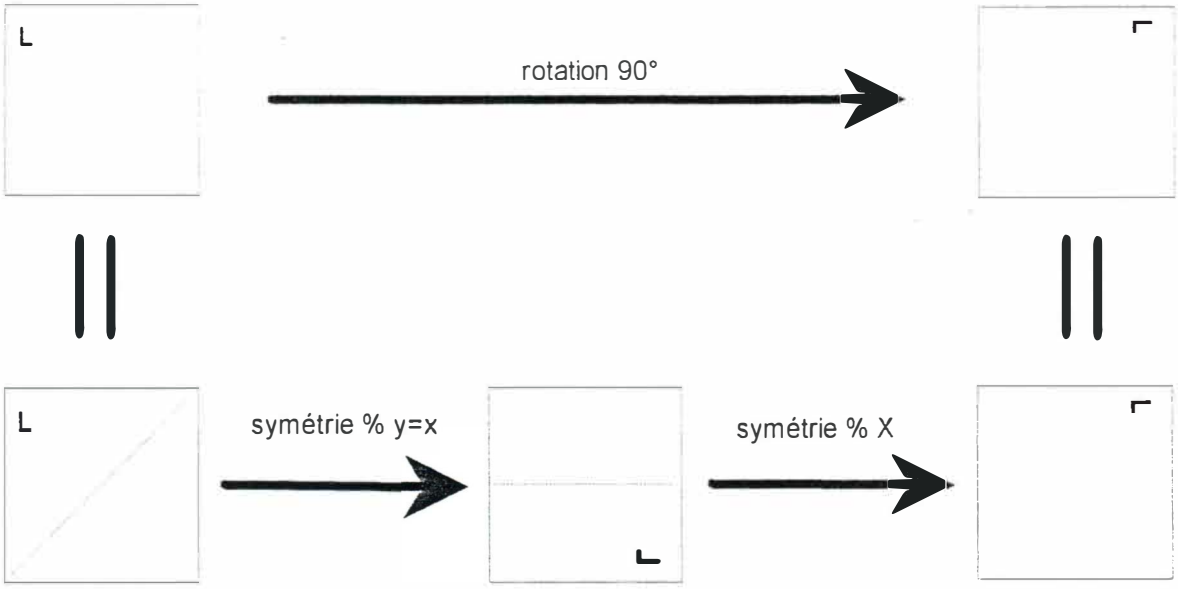
donc pas continuer à chercher un bloc portée convenable quand on a trouvé une distance nulle.

- On aurait pu (c'est réalisé dans les codes sources en commentaire) stocker les positions et symétries dans le fichier compressé sous forme d'entiers, mais l'algorithme et la taille des fichiers se prêtaient bien à un codage des coefficients simplement sous le format de caractères ASCII, réduisant en moyenne d'un facteur trois la place occupée par ces termes.

- Puisque chaque bloc portée devra être contracté d'un facteur deux avant de se voir appliquer les symétries, pourquoi ne pas réduire, une fois pour toutes, l'image de départ d'un facteur deux, sans oublier un décalage d'intensité, et en extraire les blocs portée de 8 x 8 ? C'est cette méthode plus économique qui a été implémentée.

- Une fois l'image réduite, on peut diminuer considérablement le temps de parcours des blocs portée en ne les considérant pas tous, mais seulement une partie d'entre eux. De façon qualitative, si le parcours se fait par pas de un, si un bloc portée n'est pas « convenable », il y a de fortes chances que le bloc suivant, qui possède 7/8 de ses pixels en commun, ne soit pas plus convenable que lui. D'où l'idée de parcourir les blocs portée avec un pas supérieur à un, disons quatre. Cela possède l'énorme avantage de diminuer d'un facteur 16 (4^2) le temps de calcul, sans forcément dégrader davantage la qualité de l'image de sortie. On pourrait même optimiser en choisissant de parcourir, en seconde passe, par pas de un, les blocs avoisinant un certain nombre de meilleurs blocs portée obtenus en première passe par pas de quatre, pour vérifier si un de leurs voisins ne convient pas mieux.

- Un codage simple des symétries peut se réaliser si l'on comprend que les 7 opérations de symétrie du tableau 6.1 autres que l'identité peuvent s'obtenir, par exemple, par combinaison des trois transformations suivantes : symétrie par rapport à l'axe X, symétrie par rapport à l'axe Y, symétrie par rapport à la droite d'équation y = x. Illustrons cela pour la rotation d'un angle de 90°. D'abord graphiquement, avec la figure suivante :



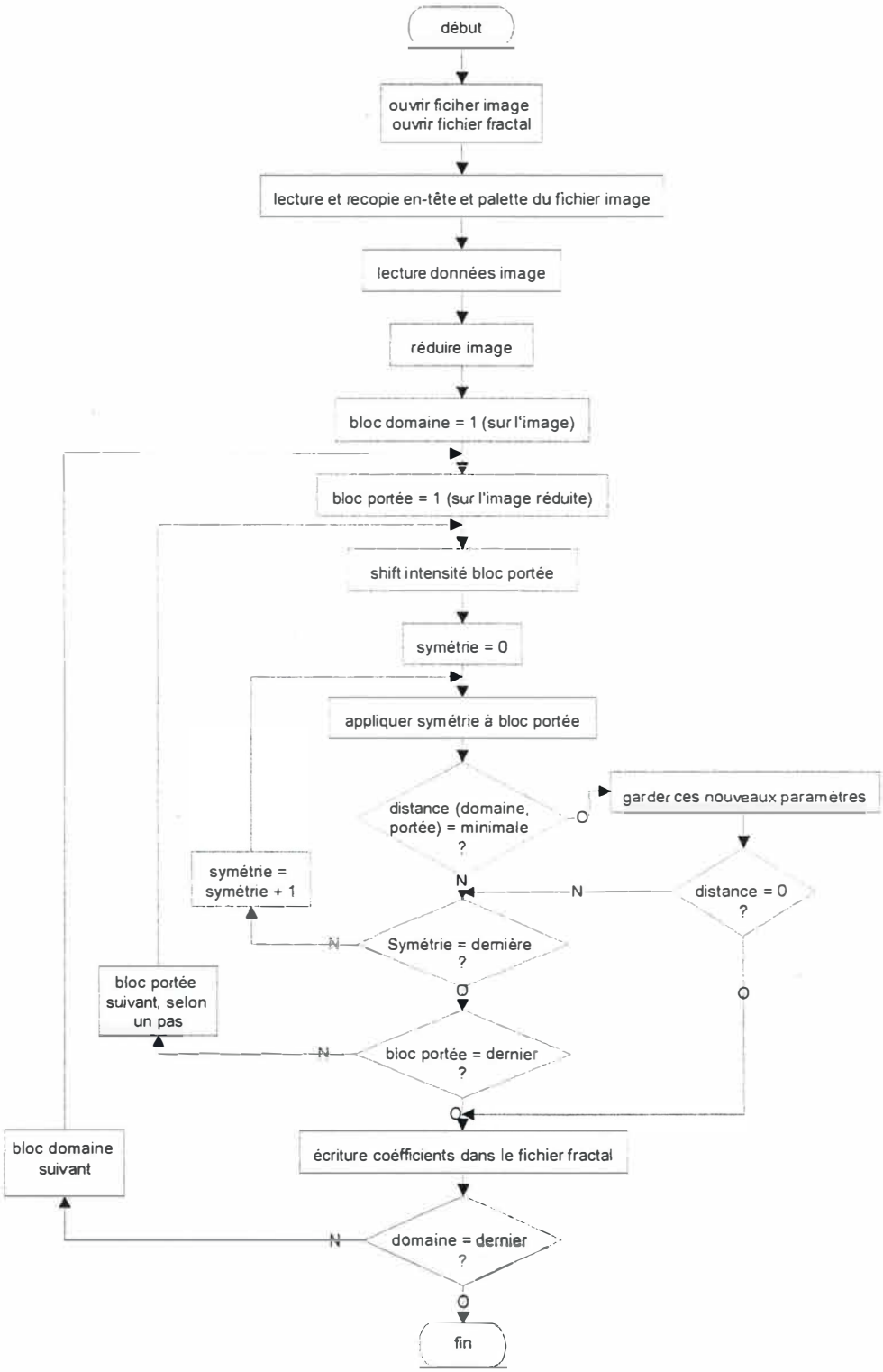
où l'on voit qu'une rotation de 90° peut s'obtenir par composition d'une symétrie par rapport à l'axe diagonal d'équation y = x et d'une symétrie par rapport à l'axe X. On peut aussi s'en persuader mathématiquement en faisant appel aux matrices représentant les transformations (tableau 6.1). Appelons R la matrice de rotation, D la matrice de symétrie diagonale et SX la matrice de symétrie par rapport à l'axe X.

$$R = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, D = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, SX = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

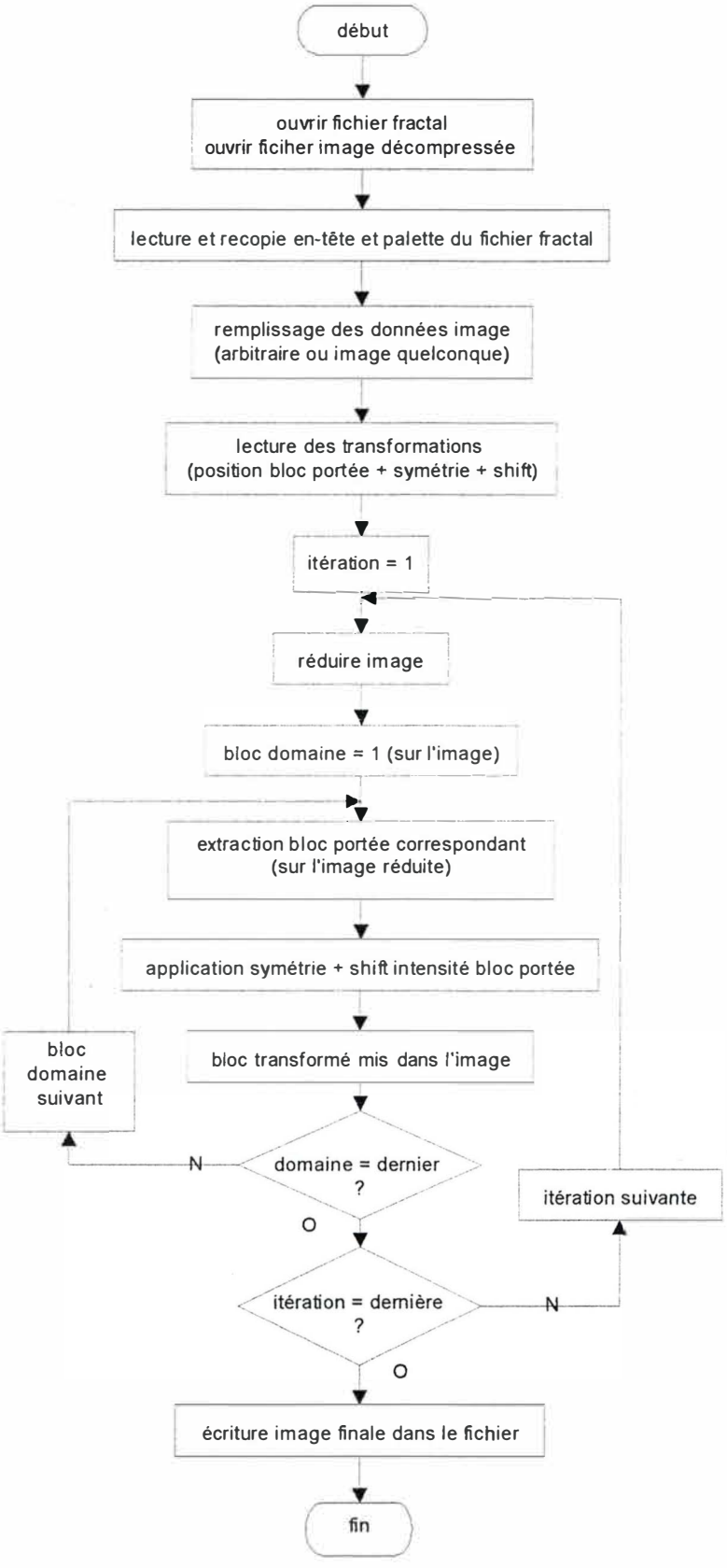
On vérifie aisément que D.SX = SX.D = R. On peut dès lors utiliser cette propriété pour représenter les symétries comme suit : chaque symétrie est codée sur 3 bits, la « base » du système étant les trois transformations SX, SY (symétrie par rapport à Y) et D, elles seront codées respectivement comme 001, 010, 100. Les autres symétries seront codées comme un OU logique entre ces symétries de base ; par exemple la rotation de 90° R vaut 101.

II. Organigrammes

A. Compression



B. Décompression



III. Codes sources

Les sources en C se composent de 5 fichiers clairement commentés :

- BMP.H : contient la structure d'en-tête et de palette de fichier bitmap BMP Windows.
- COMMUN.H : contient quelques macros et définitions de types, ainsi que les prototypes des fonctions communes appelées par les programmes de compression et de décompression.
- COMMUN.C : contient les définitions des fonctions communes appelées par les programmes de compression et de décompression.
- COMPRIME.C : le programme de compression proprement dit.
- DECOMP.C : le programme de décompression associé au programme de compression.

/** fichier bmp.h **/

```
/*-------------------------------------------------------------*/
/* Structure de l'en-tête d'un fichier bitmap BMP 256 niveaux. */
/* Noms définis par Microsoft dans le kit de développement de  */
/* Windows.  Taille : 54 octets.                               */
/*-------------------------------------------------------------*/

typedef struct
{
    /* BitmapFileHeader : 14 octets */

    unsigned int bfType;  /* BM */
    unsigned long bfSize,bfReserved,bfOffBits;

    /* BitmapInfoHeader : 40 octets */

    unsigned long biSize,biWidth,biHeight;
    unsigned int biPlanes,biBitCount;
    unsigned long biCompression,biSizeImage;
    unsigned long biXpelsPerMeter,biYpelsPerMeter;
    unsigned long biClrUsed,biClrImportant;
} Bmp_header;


/*-------------------------------------------------------------*/
/* Suivent, dans le fichier, 256*4 octets permettant d'indiquer, */
/* parmi la panoplie de 2exp32 (16 millions) de couleurs,      */
/* lesquelles vont être utilisées.                             */
/*-------------------------------------------------------------*/

typedef struct
{
    unsigned long coul_palette[256];
} Bmp_palette;


/*-------------------------------------------------------------*/
/* Enfin, arrivent les données proprement dites : chaque pixel */
/* codé sur 1 octet.                                           */
/*-------------------------------------------------------------*/
```

/** fichier commun.h **/

```
#include <stdio.h>
#include "bmp.h"

#define COTE 8
#define NB_SYM 8 /* combinaisons à partir de 3 symétries */
#define SYM_X 1
#define SYM_Y 2
#define SYM_DIAG 4

typedef unsigned char Symetrie;
typedef unsigned char Pixel;

typedef struct
{
     unsigned short largeur, hauteur;
     unsigned long longueur; /* nombre de pixels */
     Pixel *pixel;
} Rectangle;

typedef struct
{
     unsigned short ech_x, ech_y; /* position du bloc portée */
     short shift;
     Symetrie symetrie;
} Tr_affine;

typedef struct
{
     unsigned short largeur, hauteur;
} Entete;


extern Pixel moy(Rectangle *rectangle);

extern long distance(Rectangle *rect1, Rectangle *rect2);
```

```
extern void extirp_rect(Rectangle *rect_src, unsigned short x_src, unsigned short y_src,
                        Rectangle *rect_dest, unsigned short x_dest, unsigned short y_dest,
                        short largeur, short hauteur);

extern void reduire_image(Rectangle *rect_src, Rectangle *rect_dest);

extern void sym(Rectangle *bloc_ech, Rectangle *bloc_ech_transf,
                Symetrie symetrie);

extern void decal_intens(Rectangle *rectangle, short decal);

extern void lit_entete_bmp(FILE *fic_image,FILE *fic_frac,Entete *entete);
```

/** fichier commun.c **/

```
#include "commun.h"


/** lecture et recopie de l'en-tête d'un fichier BMP **/
void lit_entete_bmp(FILE *fic_1, FILE *fic_2, Entete *entete)
{
    Bmp_header bmphd;

    /* lecture en-tête */

    if(1 != fread(&bmphd, sizeof(Bmp_header), 1, fic_1))
    {
        printf("Erreur lecture en-tête BMP. \n");
        exit(1);
    }
    if((bmphd.bfType != 0x4D42) || (bmphd.biBitCount != 8))
    {
        printf("Erreur, fichier d'entrée invalide. \n");
        exit(1);
    }

    /* ecriture en-tête */

    if(1 != fwrite(&bmphd, sizeof(Bmp_header), 1, fic_2))
    {
        printf("Erreur ecriture en-tête BMP. \n");
        exit(1);
    }

    entete->largeur = bmphd.biWidth;  /* pour usage ultérieur */
    entete->hauteur = bmphd.biHeight;

} /* fin lit_entete_bmp */


/** valeur moyenne des pixels sur un rectangle **/
Pixel moy(Rectangle *rectangle)
{
    int i;
    long somme=0;

    for(i=0; i<rectangle->longueur; i++)
      somme += rectangle->pixel[i];
    return(somme / rectangle->longueur);

} /* fin moy */


/** "distance" entre deux rectangles de même "superficie" **/
long distance(Rectangle *rect1, Rectangle *rect2)
{
    int i;
    long diff, distance=0;

    for(i=0; i<rect1->longueur; i++)
    {
        /* euclidienne : heuristiquement moins "uniforme"  */
/*
        diff = rect1->pixel[i] - rect2->pixel[i];
        distance += diff * diff;
*/

        diff = abs(rect1->pixel[i] - rect2->pixel[i]);
        distance += diff;
```

```
    }
    return(distance);

} /* fin distance */


/** décalage de la valeur de chaque pixel d'un même facteur **/
void decal_intens(Rectangle *rectangle, short decal)
{
    int i;

    for(i=0; i<rectangle->longueur; i++)
      rectangle->pixel[i] = rectangle->pixel[i] + decal;

} /* fin decal_intens */


/** copie d'une zone rectangulaire d'un rectangle dans une zone **/
/** rectangulaire identique d'un autre rectangle **/
void extirp_rect(Rectangle *rect_src, unsigned short x_src,
                 unsigned short y_src, Rectangle *rect_dest,
                 unsigned short x_dest, unsigned short y_dest,
                 short largeur, short hauteur)
{
    int i, j;

    for(j=0; j<hauteur; j++)
      for(i=0; i<largeur; i++)
        rect_dest->pixel[i + x_dest + (j + y_dest) * rect_dest->largeur]
        =rect_src->pixel[i + x_src + (j +y_src) * rect_src->largeur];

} /* fin extirp_rect */


/** réduction des dimensions et de l'intensité d'un rectangle **/
/** nouveaux pixels calculés par moyenne sur 4 anciens pixels **/
void reduire_image(Rectangle *rect_src, Rectangle *rect_dest)
{
    int i, j;

    for(j=0; j<rect_dest->hauteur; j++)
      for(i=0; i<rect_dest->hauteur; i++)
      {
        /* remise à l'échelle spatiale 2 */

        rect_dest->pixel[i + j * rect_dest->largeur]=
        (rect_src->pixel[(2 * i) + (2 * j) * rect_src->largeur]+
         rect_src->pixel[(2 * i + 1) + (2 * j) * rect_src->largeur]+
         rect_src->pixel[(2 * i) + (2 * j + 1) * rect_src->largeur]+
         rect_src->pixel[(2 * i + 1) + (2 * j + 1) *rect_src->largeur]) / 4;

        /* remise à l'échelle 3/4 de l'intensité */

        rect_dest->pixel[i + j * rect_dest->largeur]=
        (rect_dest->pixel[i + j * rect_dest->largeur] * 3) / 4;
      }

} /* fin reduire_image */


/** application d'une des 8 symétries (y compris l'identité) **/
/** à un bloc carré de départ pour le transformer **/
void sym(Rectangle *bloc, Rectangle *bloc_transf, Symetrie symetrie)
{
    short i, j, x, y, tp;

    for(j=0; j<bloc->hauteur; j++)
      for(i=0; i<bloc->largeur; i++)
      {
        if(symetrie & SYM_X)
           x = (bloc->largeur - 1) - i;
        else
           x = i;

        if(symetrie & SYM_Y)
           x = (bloc->hauteur - 1) - j;
        else
           y = j;

        if(symetrie & SYM_DIAG)
        {
           tp = y;
           y = x;
```

```
            x = tp;
        }

        bloc_transf->pixel[x + y * bloc->largeur]=
        bloc->pixel[ i +j * bloc->largeur];
    }

} /* fin sym */
```

/** fichier comprime.c **/

```
/*********************************************************/
/***   Compression d'images gray-scale par fractales   ***/
/***   M. Luppi 1995                                   ***/
/*********************************************************/

/**  compilateur Power C 2.01  **/

#include <alloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "commun.h"

main(int argc, char *argv[])
{
    /***   déclarations  ***/

    FILE *f_img, *f_fractal;
    unsigned long dist, distance_min, infini=COTE*COTE*255;
    unsigned short i, range_x, range_y, domaine_x, domaine_y, shift;
    Entete entete;
    Bmp_palette palette;
    Rectangle image, image_reduite, domaine, range, range_tr;
    Symetrie symetrie;
    Pixel moy_domaine;
    Tr_affine transfo;

    /***  intro  ***/

    printf("Compression d'images gray-scale par fractales \n");
    if(argc != 3)
    {
        printf("usage : comprime image.bmp image.frc \n");
        exit(1);
    }

    /***  allocation de mémoire pour les blocs  ***/

    domaine.largeur = range.largeur = range_tr.largeur = COTE;
    domaine.hauteur = range.hauteur = range_tr.hauteur = COTE;
    domaine.longueur = range.longueur = range_tr.longueur = COTE*COTE;
    domaine.pixel = (Pixel *) calloc(COTE*COTE, sizeof(Pixel));
    range.pixel = (Pixel *) calloc(COTE*COTE, sizeof(Pixel));
    range_tr.pixel = (Pixel *) calloc(COTE*COTE, sizeof(Pixel));

    /***  image en entrée et fichier de coefficients en sortie  ***/

    if(NULL == (f_img = fopen(argv[1], "rb")))
    /* ouverture en binaire pour éviter le caractère ascii n° 26 */
    {
        printf("Impossible d'ouvrir le fichier %s. \n", argv[1]);
        exit(1);
    }
    if(NULL == (f_fractal = fopen(argv[2], "wb")))
    {
        printf("Impossible d'ouvrir le fichier %s. \n", argv[2]);
        exit(1);
    }

    /***  lecture et recopie en-tête fichier image origine  ***/

    lit_entete_bmp(f_img, f_fractal, &entete);
    printf("largeur : %d, hauteur : %d\n",entete.largeur, entete.hauteur);

    /***  allocation de mémoire pour l'image  ***/

    image.largeur = entete.largeur;
    image.hauteur = entete.hauteur;
    image.longueur = entete.hauteur; /* pour éviter l'overflow de short */
```

```
image.longueur *= entete.largeur;
image.pixel = (Pixel *) calloc(image.longueur, sizeof(Pixel));
if(NULL == image.pixel)
{
  printf("Impossible d'allouer %ld octets de mémoire pour l'image. \n",
  image.longueur);
  exit(1);
}

/***  allocation de mémoire pour la réduction de l'image  ***/

image_reduite.largeur = entete.largeur/2;
image_reduite.hauteur = entete.hauteur/2;
image_reduite.longueur = entete.hauteur/2;
image_reduite.longueur *= entete.largeur/2;
image_reduite.pixel = (Pixel *) calloc(image_reduite.longueur, sizeof(Pixel));
if(NULL==image_reduite.pixel)
{
  printf("Impossible d'allouer %ld octets de mémoire pour l'image. \n",
  image_reduite.longueur);
  exit(1);
}

/***  recopie de la palette de couleurs  ***/

if(1 != fread(&palette, sizeof(Bmp_palette), 1, f_img))
{
    printf("Erreur lecture palette BMP. \n");
    exit(1);
}
if(1 != fwrite(&palette, sizeof(Bmp_palette), 1, f_fractal))
{
    printf("Erreur d'écriture dans le fichier %s. \n", argv[2]);
    exit(1);
}

/***  remplissage de image.pixel  ***/

for (i=0; i<image.longueur; i++) image.pixel[i] = fgetc(f_img);

fclose(f_img);

/***  contraction de l'image pour les range  ***/

reduire_image(&image, &image_reduite);

/***  parcours des blocs domaine  ***/
/*-----------------------------------*/

for(domaine_y=0; domaine_y<image.hauteur; domaine_y+=COTE)
   for(domaine_x=0; domaine_x<image.largeur; domaine_x+=COTE)
   {
       printf("domx %d domy %d \n", domaine_x, domaine_y);
       distance_min = infini;

       extirp_rect(&image, domaine_x, domaine_y, &domaine, 0, 0, COTE, COTE);
       moy_domaine = moy(&domaine);

       /***  parcours des blocs portée  ***/
       /*---------------------------------*/

       for(range_y=0; range_y<=image_reduite.hauteur-COTE;
           range_y+=(COTE/2))
         for(range_x=0; range_x<=image_reduite.largeur-COTE;
             range_x+=(COTE/2))
         {
            extirp_rect(&image_reduite, range_x, range_y, &range, 0, 0, COTE, COTE);

            /***  décalage, pour que les moyennes soient égales  ***/

            shift = ((short) moy_domaine) - ((short) moy(&range));
            decal_intens(&range, shift);

            /***  parcours des symétries  ***/
            /*------------------------------*/

            for(symetrie=0; symetrie<=NB_SYM; symetrie++)
            {
               sym(&range, &range_tr, symetrie);
               dist = distance(&domaine, &range_tr);
               if(dist<distance_min)
               {
                  /* on a trouvé un bloc portée plus convenable */
```

```
                    distance_min = dist;
                    transfo.shift = shift;
                    transfo.symetrie = symetrie;
                    transfo.ech_x = range_x;
                    transfo.ech_y = range_y;

                    /* inesthétique mais économe en cas de miracle */
                    if(dist == 0) goto distance_nulle;

                } /* if */
            } /* for symétries */
        } /* for range */

        distance_nulle:

        /***  stockage en entier -> pas de compression supplémentaire  ***/
        /*
        fprintf(f_fractal, "%d,%d,%d,%d \n", transfo.ech_x,
                transfo.ech_y, transfo.symetrie, transfo.shift);
        */

        /***  stockage en AASCI -> compression supplémentaire  ***/

        fputc(transfo.ech_x, f_fractal);
        fputc(transfo.ech_y, f_fractal);
        fputc(transfo.symetrie, f_fractal);
        fputc(transfo.shift, f_fractal);

        fflush(f_fractal); /* évite les "crasses" rencontrées */

    } /* for domain */

    fclose(f_fractal);

    /***  désallocation  ***/

    free(domaine.pixel);
    free(range.pixel);
    free(range_tr.pixel);
    free(image.pixel);
    free(image_reduite.pixel);

} /* main */
```

/** fichier decomp.c **/

```
/*********************************************************/
/***   Décompression d'images gray_scale par fractales   ***/
/***   M. LUPPI 1995                                     ***/
/*********************************************************/

/**  compilateur : Power C 2.01  **/

#include <alloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "commun.h"

#define NB_ITER 15      /* par défaut si non spécifié */
#define VAL_ARBITR 127  /* couleur arbitraire d'un pixel */

void main(int argc, char *argv[])
{
    /***  déclarations  ***/

    FILE *f_img, *f_fractal;
    unsigned short i, nb_transfos, domaine_x, domaine_y,
                  iter=NB_ITER, decal=0;
    Entete entete, entete_initial;
    Bmp_palette palette;
    Rectangle image, image_reduite, range, range_tr;
    Tr_affine *pt_transfo, *pt_parcours;

/*    FILE *f_orig; */  /* remplissage avec image initiale */

    /*** intro  ***/

    printf("Décompression d'images gray_scale par fractales \n");
    if((argc < 3)||(argc > 4))
```

```
{
    printf("usage : decomp [nb_itérations] image.frc image.bmp \n");
    exit(1);
}
if(argc==4)
{
    iter = atoi(argv[1]);
    decal = 1;
}

/***  coefficients en entrée et image en sortie  ***/

if(NULL == (f_fractal = fopen(argv[decal+1], "rb")))
/* ouverture en binaire pour éviter le caractère ascii n° 26 */
{
    printf("Impossible d'ouvrir le fichier %s. \n", argv[decal+1]);
    exit(1);
}
if(NULL == (f_img = fopen(argv[decal+2], "wb")))
{
    printf("Impossible d'ouvrir le fichier %s. \n", argv[decal+2]);
    exit(1);
}

/***  lecture et recopie en-tête fichier origine  ***/

lit_entete_bmp(f_fractal, f_img, &entete);
printf("largeur : %d, hauteur : %d\n", entete.largeur, entete.hauteur);

/***  recopie de la palette de couleurs  ***/

if(1 != fread(&palette, sizeof(Bmp_palette), 1, f_fractal))
{
    printf("Erreur lecture palette BMP. \n");
    exit(1);
}
if(1 != fwrite(&palette, sizeof(Bmp_palette), 1, f_img))
{
    printf("Erreur d'écriture dans le fichier %s. \n", argv[decal+2]);
    exit(1);
}

/***  allocation de place pour les transformations  ***/

nb_transfos = entete.largeur * entete.hauteur / (COTE * COTE);
pt_transfo = (Tr_affine *) calloc(nb_transfos, sizeof(Tr_affine));

/***  allocation de place pour l'image  ***/

image.largeur = entete.largeur;
image.hauteur = entete.hauteur;
image.longueur = entete.hauteur; /* pour éviter l'overflow de short */
image.longueur *= entete.largeur;
image.pixel = (Pixel *) calloc(image.longueur, sizeof(Pixel));
if(NULL == image.pixel)
{
  printf("Impossible d'allouer %ld octets de mémoire pour l'image. \n",
  image.longueur);
  exit(1);
}

/***  allocation de place pour l'image réduite  ***/

image_reduite.largeur = entete.largeur/2;
image_reduite.hauteur = entete.hauteur/2;
image_reduite.longueur = entete.hauteur/2;
image_reduite.longueur *= entete.largeur/2;
image_reduite.pixel = (Pixel *) calloc(image_reduite.longueur, sizeof(Pixel));
if(NULL == image_reduite.pixel)
{
  printf("Impossible d'allouer %ld octets de mémoire pour l'image. \n",
  image_reduite.longueur);
  exit(1);
}

range.largeur = COTE;
range.hauteur = COTE;
range.longueur = COTE * COTE;
range.pixel = (Pixel *) calloc(COTE*COTE, sizeof(Pixel));

range_tr.largeur = COTE;
range_tr.hauteur = COTE;
range_tr.longueur = COTE * COTE;
```

```
    range_tr.pixel = (Pixel *) calloc(COTE*COTE, sizeof(Pixel));

    /***  remplissage de image.pixel  ***/

    /** avec image de départ (attention : mêmes dimensions) **/
/*
    if(NULL == (f_orig = fopen("depart.bmp", "rb")))
    {
        printf("Impossible d'ouvrir le fichier %s.\n", "depart.bmp");
        exit(1);
    }

    for(i=1; i<=1078; i++)
       fgetc(f_orig);

    for(i=0; i<image.longueur; i++)
       image.pixel[i] = fgetc(f_orig);

    fclose(f_orig);
*/
    /** remplissage arbitraire **/

    for(i=0; i<image.longueur; i++)
       image.pixel[i] = VAL_ARBITR;

    /*** lecture des transformations ***/

    pt_parcours = pt_transfo; /* première transformation */

    for(domaine_y=0; domaine_y<image.hauteur; domaine_y+=COTE)
       for(domaine_x=0; domaine_x<image.largeur; domaine_x+=COTE)
       {
           /***  lecture comme entiers -> pas de compression supplémentaire  ***/
           /*
           fscanf(f_fractal, "%d,%d,%d,%d\n", &pt_parcours->ech_x,
                  &pt_parcours->ech_y, &pt_parcours->symetrie,
                  &pt_parcours->shift);
           */

           /***  lecture en AASCI -> compression supplémentaire  ***/

           pt_parcours->ech_x = fgetc(f_fractal);
           pt_parcours->ech_y = fgetc(f_fractal);
           pt_parcours->symetrie = fgetc(f_fractal);
           pt_parcours->shift = fgetc(f_fractal);

           pt_parcours++; /* transformation suivante */

       } /* for */

    fclose(f_fractal);

    /***  itérations des transformations  ***/
    /*-----------------------------------*/

    for(i=0; i<iter; i++)
    {
         printf("Itération : %d\n", i+1);
         reduire_image(&image, &image_reduite);
         pt_parcours = pt_transfo; /* première transformation */

         /***  parcours des blocs domaine  ***/
         /*-----------------------------------*/

         for(domaine_y=0; domaine_y<image.hauteur; domaine_y+=COTE)
            for(domaine_x=0; domaine_x<image.largeur; domaine_x+=COTE)
            {
                /***  extraction du range bloc correspondant  ***/
                /*-----------------------------------------------*/

                extirp_rect(&image_reduite, pt_parcours->ech_x,
                            pt_parcours->ech_y, &range, 0, 0, COTE, COTE);

                /***  shift d'intensité et application symétrie  ***/
                /*-----------------------------------------------*/

                decal_intens(&range, pt_parcours->shift);
                sym(&range, &range_tr, pt_parcours->symetrie);

                /***  bloc transformé mis dans l'image  ***/
                /*----------------------------------------*/

                extirp_rect(&range_tr, 0, 0, &image,domaine_x, domaine_y,
```

```
                        COTE,COTE);

                pt_parcours++; /* transformation suivante */

            } /* for domaines */
    }/* for itérations */

    /***  écriture des pixels finaux dans le fichier BMP  ***/

    if(image.longueur != fwrite(image.pixel, sizeof(Pixel),
                                image.longueur, f_img))
    {
        printf("Erreur d'écriture dans le fichier %s\n", argv[decal+2]);
        exit(1);
    }

    fclose(f_img);

    /***  désallocation  ***/

    free(pt_transfo);
    free(image.pixel);
    free(image_reduite.pixel);

} /* main */
```

IV. Résultats et performances

A. Taux de compression

Le taux de compression résultant de l'application de l'algorithme proposé s'évalue rapidement par la formule suivante :

$$\text{taux} = \frac{L.H.N_p}{(L/C).(H/C).N_t},$$

où : L = longueur de l'image en pixels, H = hauteur de l'image en pixels, C = longueur d'un côté du domaine carré, N_p = nombre d'octets pour coder la couleur d'un pixel et N_t = nombre d'octets pour coder une transformation affine. Dans notre cas, on a donc :

$$\text{taux} = \frac{C^2.N_p}{N_t} = \frac{8^2.1}{4} = 16:1.$$

On a, pour cette évaluation, omis de considérer la taille de l'en-tête du fichier bitmap (1078 octets), on obtient donc, en pratique, des taux légèrement inférieurs à 16 selon la taille de l'image. Ce choix se justifie par le fait que, d'abord, la compression fractale ne concerne que les données pixels et pas l'en-tête, et, ensuite, qu'on a replacé comme en-tête du fichier fractal l'en-tête du fichier bitmap, tel quel. Cet en-tête de fichier bitmap contient, entre autres, pour le choix de la palette, 256.4 octets ; or, on sait que les niveaux de gris s'obtiennent en prenant les trois couleurs primaires (R, V, B) en égale quantité, on pourrait donc se contenter de 256 octets pour la palette, ou même s'en passer si on suppose les niveaux de gris toujours ordonnés pour toute image. Cela sans ans compter, comme mentionné dans l'annexe A, que certaines autres données de l'en-tête sont redondantes. On voit que l'on aurait pu compresser davantage le fichier en s'occupant de plus près de l'en-tête, cependant c'est la compression fractale qui nous guide et nous avons préféré ne pas toucher à cet en-tête.

On notera aussi que, par souci de clarté de code source, le nombre de bits optimum n'a pas été employé. Ainsi, par exemple, on code la symétrie sur un octet, alors qu'elle ne peut prendre que huit valeurs, trois bits auraient donc suffi.

Comme un codage statistique n'a pas été implémenté, quelques tests ont été réalisés pour comprimer les fichiers de sortie à l'aide de logiciels d'archivage classiques (ARJ, PKZIP, ...). Ils indiquent un taux de compression moyen de 1,2:1, on pourrait donc espérer un taux de compression aux alentours de 18:1, en cas d'implémentation d'un codage statistique.

B. Temps de calcul

Une demi-douzaine d'images, de taille proches de 60 Ko, ont été testées sur un Pentium 100 MHz avec le programme de compression présenté. Le temps moyen

d'exécution est de 20 minutes ; cela semble peu en comparaison des temps mentionnés dans la littérature, mais les dimensions des images sont aussi réduites comparées aux dimensions souvent rencontrées.

Au niveau de la décompression, quelques secondes suffisent.

C. Qualité

Du point de vue de la qualité de l'image restituée après décompression des codes IFS, la naïveté de l'algorithme développé ici ne permet pas d'obtenir à coup sûr une compression sans perte d'informations par rapport à l'image originale, cependant les résultats obtenus sont suffisamment significatifs : on reconnaît à chaque fois l'image d'origine.

Une remarque est à faire concernant le rapport entre la qualité de la restitution et la taille de l'image traitée. Les premiers tests, réalisés sur des images de taille très réduite (~ 40 x 40), en raison des temps de calcul, étaient très peu encourageants tant la qualité de l'image restituée était médiocre. En fait, cette qualité d'image croît avec les dimensions de celle-ci. La raison en est simple, augmenter les dimensions revient à accroître le nombre de blocs portée possibles pour chaque bloc domaine, c'est-à-dire augmenter la probabilité de trouver un bloc portée de plus en plus convenable. Donc, plus l'image est grande, plus sa qualité de restitution après la phase de compression / décompression sera meilleure.

Voyons maintenant quelques exemples : les images, de taille avant compression aux alentours de 60Ko (~ 200 x 300), sont sorties sur imprimante à bulles d'encre en 360 dpi.

original : rose.bmp

fractal : rosed.bmp

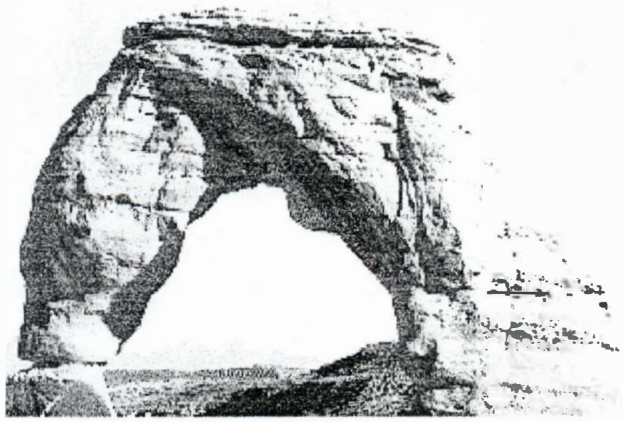
original : visage.bmp

fractal : visaged.bmp

original : idefix.bmp

fractal : idefixd.bmp

La dernière série d'illustrations montre les étapes successives de décompression à partir d'une image quelconque.



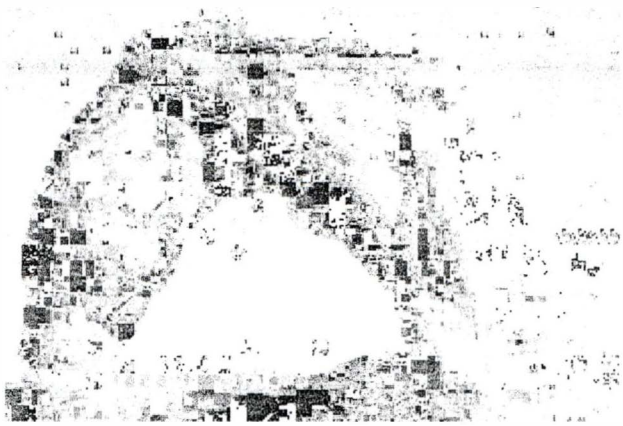
original : pic.bmp

Annexes : exemple de compression d'images par fractales

fractal itération 0 : picd0.bmp

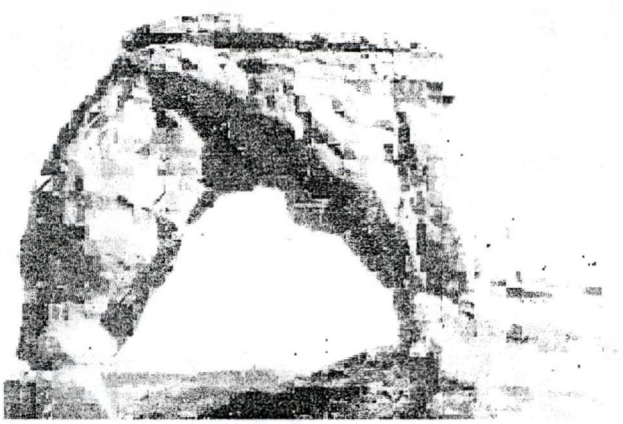
fractal itération 1 : picd1.bmp

fractal itération 2 : picd2.bmp



fractal itération 5 : picd5.bmp

fractal itération 10 : picd10.bmp



fractal itération 20 : picd20.bmp

Même si la compression se fait avec pertes d'informations, on notera que le résultat sur des petites images n'est déjà pas si mauvais. On remarquera aussi que le « motif de fond » se perçoit dès les premières itérations. Des itérations supplémentaires aux vingt premières, à peu près, n'apportent plus de modification au résultat : on a atteint l'attracteur de l'IFS.

V. Plus loin...

Le listing proposé avec ce travail fait office de programme de démonstration, et n'est absolument pas adapté pour une tâche réelle, de par ses performances réduites. On va essayer d'imaginer quelles caractéristiques devrait posséder une application performante de compression fractale.

- Tout d'abord, le choix des blocs domaine ne doit pas s'effectuer de façon statique : construits à l'avance quelle que soit l'image, comme celui présenté ici. L'idéal est un nombre minimal de blocs domaine, car c'est essentiellement ce nombre qui va déterminer le taux de compression : une position de bloc portée et une série de coefficients de transformation afffine par bloc domaine. D'où l'idée, pour minimiser ce nombre de blocs, d'une découpe successive de l'image en quadr-arbre, de manière à chercher d'abord à « couvrir » des blocs domaine de taille la plus grande possible par des blocs portée « convenables », avant de passer à des blocs de taille plus petite en cas d'échec. Si conceptuellement l'idée est simple, l'implémentation, elle, l'est nettement moins. En effet, supposons qu'une des dimensions initiales de l'image soit de (2n+1) pixels, n entier, on doit alors diviser un nombre impair en deux : (2n+1) = n + (n+1) ; de n et n+1, l'un est aussi forcément impair. Donc, lors d'une étape du partitionnement en quadr-arbre, dès qu'on arrive à une dimension d'un nombre impair de pixels, on doit segmenter en des rectangles qui n'ont pas obligatoirement les mêmes dimensions. Dans le programme proposé dans cette annexe, la situation était simple, à chaque bloc domaine carré correspond un bloc portée dont les dimensions sont doubles, et à chaque pixel du bloc domaine correspondent quatre pixels du bloc portée. Dans le cas général, cela se complique : on pourrait rencontrer, par exemple, un bloc portée de dimensions 2m x 2n qui devrait se transformer en un bloc domaine de m x n, ou un bloc de (m+1) x n, ou ... Cela impose des techniques autres qu'une simple moyenne arithmétique sur quatre pixels pour générer les points du bloc portée transformé. On ajoute à cette complexité le fait que si les blocs sont carrés, certaines opérations de symétrie peuvent s'effectuer, mais pas sur des blocs rectangulaires quelconques : rotation de 90°, symétrie par rapport à la diagonale, ... Les transformations affines « en continu » entre deux rectangles (ici au sens géomérique) quelconques ne sont pas toujours facilement transposables en transformations affines « discrètes » sur des zones rectangulaires de pixels.

- Une deuxième idée, puisqu'on vient de prononcer le mot « convenable » pour les blocs portée, est justement de paramétrer l'acceptabilité d'un bloc portée pour un bloc domaine donné. cela permettra d'optimiser l'algorithme en compression ou en temps. Si on impose, par exemple, que la distance entre le bloc portée ayant subi une transformation affine et le bloc domaine qu'il est censé recouvrir soit nulle, on opte alors pour une compression sans perte, avec le désavantage que cela nécessite un temps de parcours de blocs portée plus important (ainsi qu'une plus grande profondeur du quadr-arbre pour les domaines) et diminue donc les performances en terme de temps de l'algorithme. A l'inverse, on peut tolérer un couple bloc portée - transformation affine si la distance qui sépare le bloc transformé et le bloc domaine correspondant est inférieure à un paramètre de tolérance. Ceci nécessite un moins grand temps de calcul, mais donne lieu à une dégradation de l'image originale.

- Enfin, un parcours systématique « bête et méchant » de blocs portée (et des transformations associées), candidats pour un bloc domaine, allonge considérablement le

temps de recherche, alors qu'un simple coup d'oeil sur l'image nous ferait souvent affirmer qu'on ne peut associer tel bloc domaine avec tel bloc portée, quelle que soit la transformation affine possible. Par exemple, on voit mal comment on pourrait faire correspondre un fond d'image (ciel, végétation, papier mural, ..) assez uniforme et une zone frontière entre deux entités de teintes différentes (étagère-tapis, rocher-ciel, visage-cheveux, ...). On opte alors, comme déjà mentionné dans le dernier chapitre, pour un partitionnement des blocs portée possibles en ensembles de blocs présentant des traits communs. Cela impose un gros travail de recherche en début de procédure, mais on diminue drastiquement le temps de recherche d'un bloc portée convenable par la suite. Ce genre de procédé nécessite tout un arsenal de techniques de traitement d'images et d'intelligence artificielle dont l'exposé n'était pas le but.