Compression d'images par fractales

Rémi Dupré

Juin 2016

Table des matières

1 Introduction

Stocker une image sous un format brut entraîne des coûts importants en mémoire, il est donc important de chercher à réduire la taille des informations à stocker. Le principe de la compression d'images par fractales (1988) repose sur la redondance de motifs géométriques dans nos images.

Je me suis intéressé pendant l'année à l'implémentation de ce type de compression, ainsi qu'aux mathématiques sous-jacentes.

J'ai réalisé ce TIPE en autonomie, cependant Lucas Demarne, un autre candidat de ma classe a traité le même sujet. Nous avons emprunté des directions différentes concernant l'implémentation. Ce que l'un gagne sur la rapidité de compression est perdu au moment de la décompression.

2 Systèmes itérés de fonctions

2.1 Définition d'un espace d'images

Les théorèmes qui prouvent l'efficacité de l'algorithme qui va être décrit s'appliquent dans un espace métrique complet. Il est donc nécessaire de donner une représentation mathématique raisonnable d'une image. En l'occurrence j'ai choisi de me restreindre pour cette étude à des images bicolores : ne comportant que des pixels noirs ou blancs. On peut alors considérer qu'une image n'est qu'un ensemble fini de points de $\mathbb{N}^2$.

On considère donc l'ensemble $\mathbb{I}$ des images de largeur L et hauteur H, toutes inclues dans le rectangle $R = [\![1, L]\!] \times [\![1, H]\!]$ avec :

— $\mathbb{I} = \mathscr{P}(R)$, l'ensemble des images. (Si $A \in \mathbb{I}$, alors $(x, y) \in A$ si et seulement si le pixel de coordonnées (x, y) est noir).

— R est fini. Donc $\mathbb{I}$ est fini, de cardinal 2^{LH}.

2.2 Distance de Hausdorff

On défini la distance de Hausdorff sur $\mathbb{I}$ par $d_{\mathscr{H}} : \left\{ \begin{array}{ccc} \mathbb{I}^2 & \longrightarrow & \mathbb{R}_+ \\ A, B & \longmapsto & \max(\max\limits_{a \in A} d(a, B), \max\limits_{b \in B} d(A, b)) \end{array} \right.$

avec d la distance euclidienne sur N^2 et si $B \in \mathbb{I}$ et $a \in R$, $d(a, B) = \inf_{b \in B} d(a, b)$. Cette borne inférieure est un minimum car B est fini.

On admet ici que $d_{\mathscr{H}}$ est une distance. On a maintenant un moyen formel de caractériser la proximité entre deux images.

Comme $\mathbb{I}$ est fini, toute suite d'éléments de $\mathbb{I}$ possède une suite extraite convergente et donc $\mathbb{I}$ est compact. Comme une suite de Cauchy admettant une valeur d'adhérence est convergente, on en déduit que $\mathbb{I}$ est complet.

2.3 Systèmes itérés de fonctions

2.3.1 Définition

Soient $N \in \mathbb{N}^*$, $\alpha \in [0, 1[$. On considère une famille $(f_i)_{i \in [\![1,N]\!]} \in (\mathbb{I}^{\mathbb{I}})^N$ de fonctions contractantes de rapport de contraction inférieur à α. La famille (f_i) est appelée *système itéré de fonction* (ou encore IFS [1]). Ainsi, on a : $\forall i \in [\![1, N]\!],\ \forall (A, B) \in \mathbb{I}^2,\ d_{\mathscr{H}}(f_i(A), f_i(B)) \leq \alpha \cdot d_{\mathscr{H}}(A, B)$.

On pose $F : \left\{ \begin{array}{ccc} \mathbb{I} & \longrightarrow & \mathbb{R}_+ \\ X & \longmapsto & \bigcup\limits_{i=1}^{N} f_i(X) \end{array} \right.$.

1. "Iterated Function System", je n'ai jamais croisé l'acronyme équivalent en français

2.3.2 Preuve de contractance

Soient A et B dans $\mathbb{I}$. Pour tout $x \in F(A)$, il y existe $k_x \in [\![1, N]\!]$ et $a_x \in A$ tels que $x = f_{k_x}(a_x)$ et alors :

$$\begin{aligned}
d(x, F(B)) &= d(f_{k_x}(a_x), F(B)) = d(f_{k_x}(a_x), \bigcup_{i=1}^{N} f_i(B)) \\
&\leq d(f_{k_x}(a_x), f_{k_x}(B)) && \text{car } k_x \in [\![1, N]\!] \\
&= \inf_{b \in B} d(f_{k_x}(a_x), f_{k_x}(b)) && \text{par définition de } d(a, B) \\
&\leq \inf_{b \in B} (\alpha d(a_x, b)) && \text{par contractance} \\
&= \alpha d(a_x, B)
\end{aligned}$$

On obtient de même que pour tout élément x de $F(B)$, $d(F(A), x) \leq \alpha d(A, b_x)$ pour un certain b_x de B. On en déduit alors que F est une contraction pour la distance de Hausdorff :

$$\begin{aligned}
d_{\mathscr{H}}(F(A), F(B)) &= \max(\sup_{x \in F(A)} d(x, F(B)), \sup_{x \in F(B)} d(F(A), x)) \\
&= \max(\sup_{a_x \in A} d(f_{k_x}(a_x), F(B)), \sup_{b_x \in B} d(A, f_{k'_x}(b_x))) \\
&\leq \max(\sup_{a_x \in A} \alpha d(a_x, B), \sup_{b_x \in B} \alpha d(A, b_x)) && \text{d'après ce qui précède} \\
&= \alpha \max(\sup_{a \in A} d(a, B), \sup_{b \in B} d(A, b)) \\
&= \alpha d_{\mathscr{H}}(A, B)
\end{aligned}$$

2.4 Théorème du point fixe

2.4.1 Énoncé

Soit (E, d) un espace métrique complet, soit f une application contractante de rapport de contraction α.

Alors f possède un unique point fixe a, et, $\forall x_0 \in E, \forall n \in \mathbb{N}, d(a, x_n) \leq \frac{\alpha^n}{1-\alpha} d(a, x_0)$.

2.4.2 Preuve

Soit $x_0 \in E$, pour tout $n \in \mathbb{N}^*$ on pose $x_{n+1} = f(x_n)$.

Pour tout $n \in \mathbb{N}$ on a alors : $d(f(x_n), f(x_{n+1})) \leq \alpha^n d(x_0, x_1)$. Ainsi,
$\forall n \in \mathbb{N}, \forall p \in \mathbb{N}^*$,

$$d(x_n, x_{n+p}) \leq \sum_{k=n}^{n+p-1} d(x_k, x_{k+1}) \leq \sum_{k=n}^{n+p-1} \alpha^k d(x_0, x_1) = \alpha^n \frac{1-\alpha^p}{1-\alpha} d(x_0, x_1) \leq \frac{\alpha^n}{1-\alpha} d(x_0, x_1) \underset{n \to +\infty}{\longrightarrow} 0$$

Ainsi (x_n) est une suite de Cauchy, E étant complet cette suite converge vers un $l \in E$. Enfin, par continuité de f, $l = \lim_{n \to \infty} x_n = \lim_{n \to \infty} f(x_n) = f(l)$. Donc l est bien un point fixe de f.

Enfin, si q est un point fixe de f, $d(q, l) = d(f(q), f(l)) \leq \alpha d(q, l)$ or $\alpha < 1$ donc $d(q, l) = 0$ et donc $l = q$. Ainsi l est l'unique point fixe de f.

2.5 Application à la représentation d'images

Le théorème du point fixe dans un espace complet assure donc que F possède un unique point fixe $X \in \mathbb{I}$. De plus, quel que soit A dans $\mathbb{I}$, on sait que $\forall k \geq 1, d_{\mathscr{H}}(F^k(B), X) \leq \frac{\alpha^k}{1-\alpha} d_{\mathscr{H}}(B, X)$. On peut

donc obtenir une suite d'images $(F^k(B)_{k\geq 1})$ qui converge vers X à une vitesse exponentielle. Ainsi, on peut considérer que tout IFS représente une image, le point fixe de F.

De plus, le *théorème du collage* affirme réciproquement que tout élément de $\mathbb{I}$ peut être approché par un IFS. Ainsi, pour une image $X \in \mathbb{I}$, il n'est pas assuré qu'il existe un système de fonction dont le point fixe soit X. En revanche, quel que soit $\epsilon > 0$, le théorème du collage assure qu'il existe un IFS de point fixe A tel que $d_{\mathscr{H}}(A, X) \leq \epsilon$.

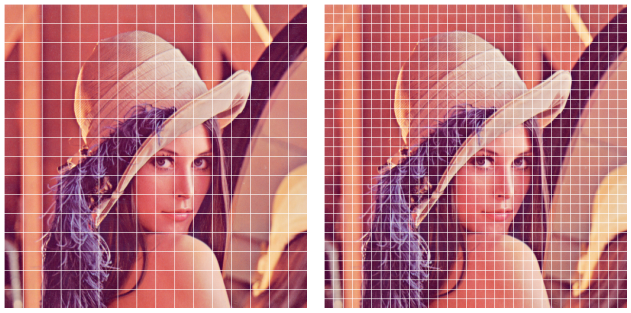
On sait donc que toute image peut être représenté grâce un système de fonctions itéré avec une précision qui peut être minimisée. Par ailleurs, du point de vue de la compression fractale, en augmentant la valeur de ϵ, on pourra trouver un IFS de poids aussi petit que souhaité pour toute image, quitte à l'altérer.

3 Principe de l'algorithme

3.1 Préliminaires

3.1.1 Découpe de l'image

En pratique les fonctions composantes de nos IFS ne vont être appliqués que localement dans l'image. On découpe l'image en petits blocs : des blocs "source" de largeur l et des blocs "destination" de largeur αl. L'objectif est que chaque bloc destination soit l'image par une application contractante d'un bloc source.



(a) Blocs sources (b) Blocs destinations

FIGURE 1 – Exemple de découpe d'une image

3.1.2 Applications

Pour un α donné, je me suis limité aux similitudes directes. Ainsi pour un bloc destination de centre m_d et un bloc source de centre m_s, je [2] cherche parmi les applications sous la forme :

$$\mathcal{T}_k : z \longmapsto m_d + \alpha e^{\frac{ik\pi}{256}}(z - m_s), k \in [\![0, 255]\!]$$

On a alors un nombre fini d'applications contractantes que l'on peut appliquer, ce qui nous permet d'énumérer tous les cas.

3.2 Algorithme de compression

L'algorithme dans sa forme primaire (cf. `compresser`) consiste à faire une recherche exhaustive pour chaque bloc destination parmi toutes les application appliquées en tous les blocs sources et conserver le couple $(source, f)$ tel que $f(source)$ soit aussi proche du bloc destination que possible.

2. L. Demarne utilise un ensemble plus restreint de fonctions.

Sous cette forme on doit donc faire une recherche exhaustive, très gourmande. En effet pour une image comportant n pixels, les nombres de blocs sources et destinations sont en $O(n)$. Ainsi, l'algorithme décrit ci-dessus demande $O(n^2)$ comparaisons (pour une taille fixée, le calcul de la distance est constant, mais la constante multiplicative peut être grande). Ainsi, pour une image carrée de coté l, l'algorithme est en $O(l^4)$.

3.3 Algorithme de décompression

L'algorithme de décompression est plus simple, il suffit d'itérer plusieurs fois notre IFS (cf. `appliquerIfs` et `decompresser`) en partant d'une image quelconque. Appliquer un ifs étant rapide je me suis permis de réitérer un nombre arbitraire de fois (10 fois en l'occurrence), au lieu par exemple de détecter une convergence en calculant la distance entre les deux derniers termes de la suite.

3.4 Implémentation

J'ai créé un programme de compression d'image en partant de l'algorithme décrit précédemment. J'ai choisi d'utiliser le langage C++, celui-ci étant compilé et permettant donc potentiellement de réduire les temps de calculs.

Les images une fois compressées sont représentées par un ifs par couche de couleur, qui est représenté de la façon suivante :

- La taille des blocs source
- La taille des blocs destination
- Pour chaque bloc destination :
  - l'indice du bloc source dont il est l'image
  - l'angle de la similitude
  - une représentation de la couleur

Ma représentation de la couleur [3] consiste à effectuer avec chaque transformation (f_i) une régression linéaire des couleurs du bloc source (c_s) vers les couleurs du bloc destination (c_d) en sorte que $c_d \approx a_i c_s + b_i$. Cependant, cette représentation est volumineuse : j'ai créé une norme de flottant 16 bits pour optimiser le stockage des a_i, ce qui ne fait que réduire le problème. Enfin, cette méthode est très coûteuse en temps de calcul car il faut effectuer une régression linéaire pendant chaque comparaison.

4 Facteurs de qualité et de performances

4.1 Étude du programme

4.1.1 Taux de compression

Cette méthode de compression peut s'appliquer à des images qui présentent une figure fractale, et dans ce cas ci il est possible de représenter une image aussi précise que désiré. On peut ainsi obtenir un "taux de compression aussi grand que l'on veut". L'exemple donné Figure 2 est une image qui peut être décompressée avec n'importe quelle précision sans perte de qualité, et pourtant le fichier la représentant ne pèse que 54 octets. Cela montre donc qu'il est possible d'avoir des forts taux de compression pour peu d'altération de la qualité. Cependant, la compression est rarement aussi efficace dans la pratique.

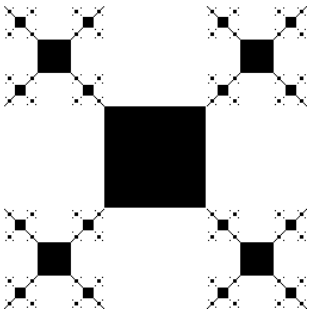


FIGURE 2

3. L. Demarne utilise une modèle plus simple.

4.1.2 Vitesse d'exécution

Le principal défaut du programme est sa lenteur : le temps d'exécution pour compresser une image est de l'ordre de l'heure. En revanche, la décompression est expérimentalement aussi rapide que prévu. La figure 3 montre que dès la troisième itération, l'image a presque convergé vers l'image finale (l'image finale est atteinte en moins de 6 itérations). Les itérations ne demandent qu'un coût linéaire en le nombre de pixels de l'image et sont donc très rapides.

(a) Image d'origine (b) Première itération (c) Deuxième itération (d) Troisième itération

FIGURE 3 – Premières itérations de la décompression de l'image

4.1.3 Comparaison à d'autres algorithmes

Pour avoir une idée de l'efficacité de la méthode de compression, je l'ai comparé au format classique de compression avec pertes : le jpeg. On remarque que sur des images comportant des surfaces lisses, l'algorithme est très efficace. Le jpeg devient bien plus efficace lorsque l'image comporte un grand nombre de détails.

Lenna C'est une photographie très classique et complète. On constate que l'image est bien plus altérée qu'avec le jpg. Mais les différentes méthodes de compression réduisent conséquemment le poids de l'image.

— Fichier brut : $3 \times 512^2 = 786$ko
— Fichier png : 476ko (61%)
— Fichiers jpg et fractal : 87ko (11%)

(a) Image complète (b) Echantillon (c) Compression en jpeg (d) Par fractales

FIGURE 4 – Exemple de compression : lenna.png

Yin Yang Mon programme est souvent plus efficace que le jpg pour compresser des images géométriques (formes droites, lises ou comportant des dégradés). Ici, on observe très peu d'artefacts de compression sur la figure 5d. Mais on remarque que l'image étant très pauvre, le format png réduit déjà très fortement la taille sans perdre aucune donnée.

— Fichier brut : 93ko

— Fichier png : 4, 2ko (4,5%)
— Fichiers jpg et fractal : 2, 1ko (2,3%)

(a) Image complète (b) Echantillon (c) Compression en jpeg (d) Par fractales

FIGURE 5 – Exemple de compression : yin-yang.png

4.2 Amélioration de la recherche

L'algorithme direct n'est clairement pas optimal, il existe des piste d'amélioration de ce dernier, que ce soit du point de vue du taux de compression ou de la vitesse d'exécution.

4.2.1 Découpe dynamisée

Lorsque j'ai implémenté l'algorithme dans sa forme primitive, j'ai vite constaté que la taille de la découpe influait beaucoup sur la qualité du résultat. Cependant, pour certaine images une certaine taille de découpe est adapté pour une partie de celle-ci mais si l'on veut un rendu correct sur une autre partie de l'image, il faudra réduire encore la taille de la découpe.

J'ai donc géré des tailles de blocs différents au sein de l'image, lorsque la précision pour un bloc destination n'est pas satisfaisante, il est subdivisé en quatre blocs de coté plus petits, pour lesquels on réitère la recherche. Cette méthode s'est révélée très concluante pour augmenter le taux de compression et réduire le temps de calculs.

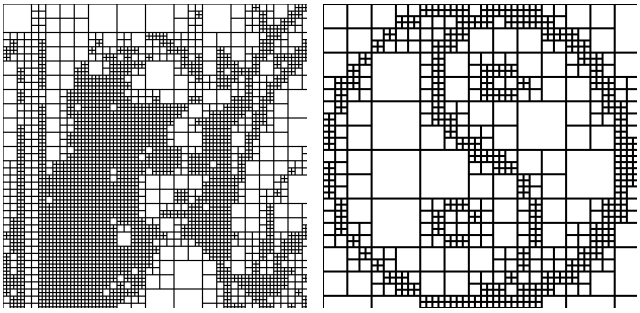


FIGURE 6 – Application d'une découpe dynamique

4.2.2 Recherche moins directe

Je me suis également essayé à trouver un algorithme permettant de réduire la complexité de la recherche d'une application parmi les blocs source. J'ai créé deux algorithme en Python qui ont une complexité en $O(n \ln n)$ au lieu de $O(n^2)$ où n est le nombre de bloc parmi lesquels rechercher. Ces algorithmes ne permettent pas de trouver le bloc le plus proche mais effectuent une approximation.

Leur implémentation en Python m'a permit d'étudier le nombre de comparaisons à effectuer pour un groupe de blocs générés aléatoirement, mais pas d'étudier la réduction de qualité qu'ils impliquent.

Chacune de ces méthodes ont conséquemment réduit le nombre de comparaisons (divisées par dix environs) mais augmentent la variance moyenne entre deux blocs d'une dizaine de pourcents. Cela pourrait dans la pratique avoir des conséquences importantes sur la qualité de l'image.

5 Conclusion

La théorie des IFS a donc permis de trouver un algorithme assez simple et plutôt efficace de compression. Sa principale faiblesse réside dans la lenteur de la compression. Cependant, l'algorithme semble pouvoir être amélioré sur de nombreux aspects et peut être prometteur, les différences de choix que L. Demarne et moi avons ont données des programmes très différents [4] qui illustrent la souplesse de l'algorithme. Et en effet, les recherches à ce sujet semblent toujours actives [5].

4. Son algorithme est plus rapide, mais il offre des taux de compression plus faibles et la décompression est beaucoup plus lente.

5. Un article a été publié au mois de mai par une université coréenne : *Enhancing fractal image compression speed using local features for reducing search space*

6 Annexes

6.1 Bibliographie

[1] *Topologie et analyse*, Georges Skandalis, édition Dunod

[2] Étude sur la qualité de compression : https://karczmarczuk.users.greyc.fr/matrs/Dess/RADI/Refs/SaHaHa96a.pdf

[3] *Development and Comparison of Image Encoders Based on Different Compression Techniques*, École de génie des télécommunications de Barcelone

[4] Étude de la nasa de ce que pourrait leur apporter cette compression (d'avant 1993) http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19930016738.pdf

6.2 Algorithmes en pseudo code

6.2.1 Compression

Données : SOURCE : liste des blocs sources (liés à l'image)
Données : DEST : liste des blocs destination (liés à l'image)
Résultat : L'IFS sous la forme de la liste des couples (source, application, destination)
pour chaque *dest* **dans** *DEST* **faire**
  `// Recherche pour le bloc DEST[i]`
  $meilleur\_src \leftarrow SOURCE[1]$
  $meilleur\_transfo \leftarrow TRANSFO[1]$
  $meilleur\_dist \leftarrow 0$
  pour chaque *source* **dans** *SOURCE* **faire**
    pour chaque *transformation* **faire**
      $bloc \leftarrow transformation(source)$
      $dist \leftarrow d_{\mathscr{H}}(bloc, dest)$
      si $dist < meilleur\_dist$ **alors**
        $meilleur\_dist \leftarrow dist$
        $meilleur\_src \leftarrow source$
        $meilleur\_transfo \leftarrow transformation$
      fin
    fin
  fin
  insérer $(dest, meilleur\_transfo, meilleur\_src)$ dans IFS
  retourner *IFS*
fin

Fonction compresser(image)

6.2.2 Décompression

Données : SOURCE : liste des blocs sources (liés à l'image)
Données : DEST : liste des blocs destination (liés à l'image)
Résultat : Une nouvelle image correspondant à IFS(image)
retour ← Image Vide
pour chaque *(source, transfo, destination)* **dans** *IFS* **faire**
  $destination(retour) \leftarrow transfo(source(image))$
fin
retourner *retour*

Fonction appliquerIfs(image, IFS)

Données : SOURCE : liste des blocs sources (liés à l'image)
Données : DEST : liste des blocs destination (liés à l'image)
Résultat : Une nouvelle image correspondant à IFS(image)
image ← Image Vide
pour $i = 1$ **à** 10 **faire**
  `// Typiquement, 10 itérations suffisent`
  $image \leftarrow appliquerIfs(image, IFS)$
fin
retourner *image*

Fonction decompresser(IFS)

6.3 Programme : recherche de bloc

comparer.py

```
# Effectue quelques tests sur les différentes méthodes de recherches qui ont été implémentées

from bloc import Bloc

NB_DESTINATIONS = 500                    # Le nombre de blocs destinations
NB_SOURCES = (NB_DESTINATIONS // 4) * 8 # 8 transformations appliquées à des blocs 8 fois plus petits

destinations = [ Bloc() for i in range(NB_DESTINATIONS) ]
sources = [ Bloc() for i in range(NB_SOURCES) ]

distances = [ Bloc.distance(d, s) for d in destinations for s in sources ]
print("La plus grande distance est : ", max(distances))
print("La plus petite distance est : ", min(distances))
print("La distance moyenne est ", sum(distances) / len(distances))

## Algorithme trivial

import trivial

Bloc.comparaisons = 0
match = trivial.match(destinations, sources)

print()
print("L'algorithme trivial demande ", Bloc.comparaisons, " comparaisons")

distances = [ Bloc.distance(destinations[d], sources[s]) for (d, s) in match ]
print("La plus grande distance est : ", max(distances))
print("La plus petite distance est : ", min(distances))
print("La distance moyenne est ", sum(distances) / len(distances), " (optimal)")

## Par découpe de l'espace

from decoupe import Zone

Bloc.comparaisons = 0
arbre = Zone(sources)

print()
print("Construire l'arbre de répartition demande ", Bloc.comparaisons, " comparaisons")

a_match = []
for i in range(len(destinations)) :
    s = arbre.chercher(destinations[i])
    a_match.append((i, s))

print("Cette méthode demande au total ", Bloc.comparaisons, " comparaisons")
```

```
47
48  distances = [ Bloc.distance(destinations[d], sources[s]) for (d, s) in a_match ]
49  print("La plus grande distance est : ", max(distances))
50  print("La plus petite distance est : ", min(distances))
51  print("La distance moyenne est ", sum(distances) / len(distances))
52
53
54  ## Construction d'un graphe
55
56  from graphe import Graphe
57
58  Bloc.comparaisons = 0
59  G = Graphe(sources)
60
61  print()
62  print("Construire le graphe demande ", Bloc.comparaisons, " comparaisons")
63
64  g_match = []
65  for i in range(len(destinations)) :
66      s = G.chercher(destinations[i])
67      g_match.append((i, s))
68
69  print("Cette méthode demande au total ", Bloc.comparaisons, " comparaisons")
70  distances = [ Bloc.distance(destinations[d], sources[s]) for (d, s) in g_match ]
71  print("La plus grande distance est : ", max(distances))
72  print("La plus petite distance est : ", min(distances))
73  print("La distance moyenne est ", sum(distances) / len(distances))
```

bloc.py

```
1   import numpy as np
2
3   class Bloc :
4       # Classe représentant un bloc dans une image
5       # Attributs :
6       #  - data : les pixels du bloc
7
8       TAILLE = 8          # La taille des blocs
9       comparaisons = 0    # Nombre decomparaisons effectuées
10
11      def __init__(self) :
12          # Le bloc a un contenu aléatoire par défaut
13          valeurs = np.random.random((Bloc.TAILLE, Bloc.TAILLE)) * 256
14          self.data = np.array(valeurs, dtype=int)
15
16      def distance(A, B) :
17          # Calcule la variance entre deux blocs
18          n = len(A.data)
19          Bloc.comparaisons += 1
20
21          D = A.data - B.data
22          return ( (np.sum(D**2)) - (np.sum(D)**2 // n**2) ) // n**2
```

trivial.py

```python
# Ce fichier contient les fonctions de recherche exhaustive
# Le résultat par cette méthode donne donc les meilleurs approximations possibles et une complexité maximale.

from bloc import Bloc

def match(destinations, sources) :
    # Effectue la recherche des blocs proches de façon exhaustive
    # Entrée :
    #  - destinations : liste des blocs destination (pour lesquels on cherche)
    #  - sources : liste des blocs source (parmi lequels on cherche)
    # Sortie : une liste du doublets (d, s)
    #  - où 'd' est l'indice d'un bloc destination
    #  - s est l'indice du bloc source le plus proche

    retour = []
    for d in range(len(destinations)) :
        dest = destinations[d]
        retour.append( (d, chercher(sources, dest)) )
    return retour

def chercher_min(sources, bloc, membres=None) :
    # Retourne le bloc source le plus proche
    # Entrées :
    #  - sources : liste des blocs source parmis lesquels chercher
    #  - bloc : le bloc qu'on cherche à approcher
    #  - membres (falcutatif) : les indices à considérer dans 'sources'
    # Sortie :
    #  - l'indice du bloc source proche
    #  - la distance de ce bloc
    if membres is None : membres = range(len(sources))

    distance = float('inf')
    proche = membres[0]
    for s in membres :
        nv_dist = Bloc.distance(sources[s], bloc)
        if nv_dist < distance :
            distance = nv_dist
            proche = s
    return proche, distance

def chercher(sources, bloc, membres=None) :
    # Pareil que chercher_dist mais sans retourner la distance
    r, _ = chercher_min(sources, bloc, membres)
    return r
```

graphe.py

```python
from operator import itemgetter

from bloc import Bloc
```

```python
import trivial

class Graphe :
    # Représente un ensemble de Blocs par un graphe
    # Attributs :
    #  - voisins : liste des listes d'adjacence
    #  - sources : liste des blocs source
    # La racine de l'arbre est le bloc d'indice 0

    def __init__(self, sources) :
        # Initialise le graphe
        # Entrée (sources) : la liste des blocs à représenter

        n= len(sources)
        self.sources = sources
        self.voisins = [ [] for _ in range(n) ]

        distances = [ (Bloc.distance(sources[i], sources[0]), i) for i in range(1, len(sources)) ]
        distances.sort(key=itemgetter(0)) # On ajoute les blocs par ordre croissant de distance
        for dist, i in distances :
            parent = self.chercher(sources[i], 0, dist)
            self.voisins[parent].append(i)

    def chercher(self, bloc, sommet=0, dist=None) :
        # Retourne l'indice d'un bloc proche dans le graphe
        # Entrées :
        #  - bloc : le bloc qu'on cherche à approcher
        #  - sommet : le sommet d'où part la recherche
        #  - dist : la distance du bloc à ce sommet

        if dist is None : dist = Bloc.distance(self.sources[sommet], bloc)
        if not self.voisins[sommet] :
            return sommet
        else : # On cherche si un descendant du sommet est plus proche
            fils, nv_dist = trivial.chercher_min(self.sources, bloc, self.voisins[sommet])
            if nv_dist < dist :
                return self.chercher(bloc, fils, nv_dist)
            else :
                return sommet
```

decoupe.py

```python
from operator import itemgetter

from bloc import Bloc
import trivial


TAILLE_MIN_DECOUPE = 10 # Le nombre de
```

```
8
9  class Zone :
10     # Représente le partitionement d'un ensemble de blocs
11     # Attributs :
12     #  - (list) sources : liste des blocs sources
13     #  - (list) membres : liste des membres de la zone
14     #  - (bool) feuille : vrai si la zone n'a pas été redécoupée
15     # Si ce n'est pas une feuille :
16     #  - (int) centre : l'indice du représentant de la zone
17     #  - (float) r1, r2 : les deux rayons délimitant 3 zones
18     #  - (Zone) b1, b2, b3 : les trois zones
19
20     def __init__(self, sources, membres=None, distances=None) :
21         # Crée une zone représentant la liste des membres
22         # Distance est défini pour ne pas être recalculé si le centre (premier
   ↪ élément de 'membres') est le même que celui de la zone parente
23         if membres is None : membres = range( len(sources) )
24
25         self.sources = sources
26         self.membres = membres
27         self.feuille = len(membres) < TAILLE_MIN_DECOUPE
28
29         if not self.feuille :
30             self.centre = membres[0]
31             if distances is None :
32                 distances = [ (bloc, self.eloignement(sources[bloc])) for bloc in
   ↪ membres[1:] ]
33                 distances.sort(key=itemgetter(1)) # Ordonne par ordre croissant de
   ↪ distance
34                 self.membres = [ self.centre ] + [ bloc for bloc, _ in distances ] #
   ↪ Réinjecte dans membres
35
36             p1, p2 = len(distances)//3, 2*len(distances)//3 # On coupe à la médiane
37             self.r1, self.r2 = distances[p1][1], distances[p2][1]
38
39             self.b1 = Zone(sources, self.membres[:p1], distances[:p2])
40             self.b2 = Zone(sources, self.membres[p1:p2])
41             self.b3 = Zone(sources, self.membres[p2:])
42
43     def eloignement(self, bloc) :
44         # Retourne l'éloignement du bloc au centre de la zone
45         return Bloc.distance(bloc, self.sources[self.centre])
46
47     def candidats(self, bloc) :
48         # Retourne une liste de potentiels blocs proches de 'bloc'
49         if self.feuille :
50             return list(self.membres)
51         elif self.eloignement(bloc) < self.r1 :
52             return self.b1.candidats(bloc) + self.b2.candidats(bloc)
53         elif self.eloignement(bloc) > self.r2 :
54             return self.b1.candidats(bloc) + self.b2.candidats(bloc) +
   ↪ self.b3.candidats(bloc)
```

```
55          else :
56              return self.b2.candidats(bloc) + self.b3.candidats(bloc)
57
58      def chercher(self, bloc) :
59          # Recherche un bloc source proche dans la zone, en retoure l'indice
60          return trivial.chercher(self.sources, bloc, self.candidats(bloc))
```

6.4 Programme de compression

Le code complet est également disponible sur github (https://github.com/remi100756/Compression-Fractale).

main.cpp

```
/*
 * Programme de compression d'image par fractale (Rémi Dupré & Lucas Demarne)
 * Dépendances particulières :
 *  - tclap (dans les dépots ubuntu, à intégrer manuelement avec mingw)
 *  - lpthread (nécessite d'être linké avec -lpthread)
 *
 ******************* Page d'aide affichée **************
 *
 * USAGE:
 *
 *   ./fzip-l64  [-e <int>] [--threads <int>] [-n <int>] [-b <int>] [-s
 *               <int>] [-f <string>] [-p <string>] [-x] [-z] [-t] [-c] [-q]
 *               [-v] [--] [--version] [-h]
 *
 *
 * Where:
 *
 *    -e <int>,  --examples <int>
 *      Génère un set de fichiers types, donner leur taille en argument
 *
 *    --threads <int>
 *      Nombre de threads maximum utilisés
 *
 *    -n <int>,  --nb-iterations <int>
 *      Le nombre d'itérations à la décompression
 *
 *    -b <int>,  --big <int>
 *      La taille des gros carrés (compression)
 *
 *    -s <int>,  --small <int>
 *      La taille des petits carrés (compression)
 *
 *    -f <string>,  --fractal-file <string>
 *      Le fichier .ifs
 *
 *    -p <string>,  --png-file <string>
 *      Le fichier .png
 *
 *    -x,  --extract
 *      Le fichier entré doit être extrait
 *
 *    -z,  --compress
 *      Le fichier entré doit être compressé
 *
 *    -t,  --transparence
 *      L'image doit être compressée avec transparence
```

```
47   *
48   *    -c,  --couleur
49   *      L'image doit être compressée en couleur
50   *
51   *    -q,  --quiet
52   *      Retire les affichages courants de la console
53   *
54   *    -v,  --verbose
55   *      Afficher le debugage
56   *
57   *    --,  --ignore_rest
58   *      Ignores the rest of the labeled arguments following this flag.
59   *
60   *    --version
61   *      Displays version information and exits.
62   *
63   *    -h,  --help
64   *      Displays usage information and exits.
65   *
66   *
67   *    Algorithme de compression fractal
68   */
69
70   #include "FigureFractale.h"
71
72   #include <tclap/CmdLine.h>
73   #include <cmath> // ceil
74   #include "ImageFractale.h"
75   #include "debug.h"
76
77   #include <sstream>
78   #include <string>
79
80   int main(int argc, char** argv) {
81       extern bool VERBOSE, SILENCIEUX;
82       extern int ITERATIONS_DECOMPRESSION, NB_THREADS;
83       extern int TAILLE_MIN_DECOUPE, NB_MAX_DECOUPE;
84
85           /* *************** Lecture des entrées (paramètres de la ligne de commande)
     ↪  *************** */
86
87       try {
88           TCLAP::CmdLine cmd("Algorithme de compression fractal", ' ', "0.42");
89
90           // Paramètres de compression
91           TCLAP::ValueArg<int> argTailleGros("b", "big", "La taille des gros carrés
     ↪  (compression)", false, 96, "int");
92           TCLAP::ValueArg<int> argTaillePetit("s", "small", "La taille des petits
     ↪  carrés (compression)", false, 48, "int");
93           TCLAP::ValueArg<int> argNbIterations("n", "nb-iterations", "Le nombre
     ↪  d'itérations à la décompression", false, ITERATIONS_DECOMPRESSION, "int");
```

```
94          TCLAP::ValueArg<int> argThreads("", "threads", "Nombre de threads maximum
    ↪ utilisés", false, NB_THREADS, "int");
95          // Fichiers d'entrée
96          TCLAP::ValueArg<std::string> argFractalFile("f", "fractal-file", "Le fichier
    ↪ .ifs", false, "out.ifs", "string");
97          TCLAP::ValueArg<std::string> argNormalFile("p", "png-file", "Le fichier
    ↪ .png", false, "out.png", "string");
98          // Affichage
99          TCLAP::SwitchArg argVerbose("v", "verbose", "Afficher le debugage", cmd,
    ↪ !VERBOSE);
100         TCLAP::SwitchArg argQuiet("q", "quiet", "Retire les affichages courants de
    ↪ la console", cmd, SILENCIEUX);
101         // Type d'image
102         TCLAP::SwitchArg argCouleur("c", "couleur", "L'image doit être compressée en
    ↪ couleur", cmd, false);
103         TCLAP::SwitchArg argTransparence("t", "transparence", "L'image doit être
    ↪ compressée avec transparence", cmd, false);
104         // Type de travail
105         TCLAP::SwitchArg argCompresser("z", "compress", "Le fichier entré doit être
    ↪ compressé", cmd, false);
106         TCLAP::SwitchArg argExtraire("x", "extract", "Le fichier entré doit être
    ↪ extrait", cmd, false);
107         TCLAP::ValueArg<int> argExamples("e", "examples", "Génère un set de fichiers
    ↪ types, donner leur taille en argument", false, 0, "int");
108
109         cmd.add( argNormalFile );
110         cmd.add( argFractalFile );
111         cmd.add( argTaillePetit );
112         cmd.add( argTailleGros );
113         cmd.add( argNbIterations );
114         cmd.add( argThreads );
115         cmd.add( argExamples );
116         cmd.parse( argc, argv );
117
118         VERBOSE = argVerbose.getValue();
119         SILENCIEUX = argQuiet.getValue();
120         ITERATIONS_DECOMPRESSION = argNbIterations.getValue();
121         NB_THREADS = argThreads.getValue();
122
123         const char* fractalFile = argFractalFile.getValue().c_str();
124         const char* normalFile = argNormalFile.getValue().c_str();
125         int taillePetit = argTaillePetit.getValue();
126         int tailleGros = argTailleGros.getValue();
127         bool couleur = argCouleur.getValue();
128         bool transparence = argTransparence.getValue();
129
130         TAILLE_MIN_DECOUPE = taillePetit / NB_MAX_DECOUPE;
131
132         /* ************** Mise en exécution des entrées ************** */
133
134         DEBUG << "Flotant : " << sizeof(Flotant16b) << "octets" << std::endl;
135         DEBUG << "En tete : " << sizeof(Pack_Entete) << "octets" << std::endl;
```

```
136         DEBUG << "ISF : " << sizeof(Pack_IFS) << "octets" << std::endl;
137         DEBUG << "Correspondance : " << sizeof(Pack_Correspondance) << "octets" <<
    ↪  std::endl;
138 
139         if( argCompresser.getValue() ) { // Doit encoder
140             ImageFractale imgF = ImageFractale::compresser(normalFile, taillePetit,
    ↪  tailleGros, couleur, transparence);
141             imgF.enregistrer(fractalFile);
142             imgF.debugSplit();
143             imgF.exporter("debug.png");
144         }
145 
146         if( argExtraire.getValue() ) { // Doit décoder
147             ImageFractale img( fractalFile );
148             img.debugSplit();
149             img.exporter( normalFile );
150         }
151 
152         if( argExamples.getValue() > 0 ) { // Génération de fichiers types
153             FigureFractale::generer_exemples(argExamples.getValue());
154         }
155     }
156     catch (TCLAP::ArgException &e) {
157         std::cerr << "error: " << e.error() << " for arg " << e.argId() <<
    ↪  std::endl;
158     }
159 }
```

parametres.cpp

```
1  /* Ce fichier contient les paramètres de compression
2   * Ces paramètes sont présent sous la forme de variables globales car ils sont
    ↪  quasi-constant, mais quand même modifiable au lancement du programme
3   */
4 
5  /* ***************** Gestion des ressources ***************** */
6 
7  // Le nombre de processus qui sont crées pour la compression
8  int NB_THREADS = 10;
9 
10 // Le nombre d'itérations effectuées pour décompresser une image
11 int ITERATIONS_DECOMPRESSION = 15;
12 
13 /* ***************** Algorithme de compression ***************** */
14 
15 // Limite d'acceptation pour les bouts lisses
16 float SEUIL_LISSAGE = 625;
17 
18 // Limite d'acceptation pour les transfos en général
19 float SEUIL_VARIANCE = 625;
20 
21 // Limite au dessus de laquelle on redécoupe la partie
```

```
22 float SEUIL_DECOUPE = 2000;
23
24 // Taille minimum de redécoupe
25 int TAILLE_MIN_DECOUPE = 4;
26
27 // Le nombre maximal de découpes qui pouront être faites, /!\ Prends le dessus sur
   ↪  TAILLE_MIN_DECOUPE
28 int NB_MAX_DECOUPE = 4;
29
30 /* ***************** Décompression ***************** */
31
32 int QUALITE_DECOMPRESSION = 5; // Ameiloration de la decompression, 1 : rien n'est
   ↪  changé
33
34 /* ***************** Débugage ***************** */
35
36 // Dossier contenant les resultats du debugage
37 const char* DOSSIER_DEBUG = "debug/";
38
39 // Si activé, désactive les affichages courrants en console
40 bool SILENCIEUX = false;
41
42 // Si activé, active les débugs en console
43 bool VERBOSE = true;
```

ImageMatricielle.h

```
1  class ImageMatricielle;
2
3  #ifndef IMAGEMATRICIELLE
4  #define IMAGEMATRICIELLE
5
6  #include <vector>
7  #include <queue>
8  #include <stack>
9  #include <list>
10 #include <ctime>
11 #include <cmath> // pow
12 #include <unistd.h> // sleep(int)
13 #include "lib/lodepng.h" // https://github.com/lvandeve/lodepng
14
15 #define timespec thread_timespec // Evite un conflict avec ctime (win)
16 #include <pthread.h>
17 #undef timespec
18
19
20 #include "ImagePart.h"
21 #include "formatIfs.h"
22 #include "multithread.h"
23 #include "debug.h"
24
25 class ImageMatricielle {
```

```
26      public :
27          ImageMatricielle(const char* fichier, int couche);
28          ImageMatricielle(unsigned int x, unsigned int y);
29          ~ImageMatricielle();
30
31          void sauvegarder(const char* fichier) const;
32          ImageMatricielle* cloner();
33
34          /* ********** Traitement ********** */
35
36          std::vector<ImagePart> decouper(int taille);
37          static std::vector<ImagePart> adapterDecoupe(std::vector<ImagePart>&, const
    ↪   std::vector<Correspondance>&);
38
39          IFS chercherIFS(int taillePetit, int tailleGros, const char* message = "");
40          ImageMatricielle appliquerIFS(const IFS& ifs);
41
42          void lisser(int n = 1);
43          void retrecir(int reduction);
44
45          /* ********** Getters & Setters ********** */
46
47          unsigned char* operator[](int i);
48          unsigned char& at(int i, int j);
49
50          int getLargeur() const;
51          int getHauteur() const;
52
53          unsigned char moyenne() const;
54          void adapterMoyenne(unsigned char val);
55          void remplir(unsigned char val);
56
57      private :
58          unsigned char **mImage; // Les pixels sont représentés par des octects
59          unsigned int mLargeur;  // Largeur en pixels de l'image
60          unsigned int mHauteur;  // Hauteur en pixels de l'image
61  };
62
63  #endif
```

ImageMatricielle.cpp

```
1   #include "ImageMatricielle.h"
2
3   /* ************** Constructeur / Destructeur ************** */
4
5   ImageMatricielle::ImageMatricielle(unsigned int x, unsigned int y) : mLargeur(x),
    ↪   mHauteur(y) {
6       /* Créé une nouvelle image de dimensions données
7        * Les pixels de l'image ne sont pas initialisés
8        */
9       mImage = new unsigned char* [mLargeur];
```

```
10      for(int i=0 ; i<mLargeur ; i++) {
11          mImage[i] = new unsigned char[mHauteur];
12      }
13  }
14
15  ImageMatricielle::ImageMatricielle(const char* fichier, int couche) {
16      /* Ouvre un fichier '.png'
17       * Entrées :
18          - fichier : l'adresse du fichier
19          - couche : la couche à lire (de 0 à 3)
20       * Sortie : si l'ouverture échoue, l'image prend les dimensions 0x0
21       */
22      extern bool VERBOSE;
23
24      bool erreur = false;
25      std::vector<unsigned char> png;
26      std::vector<unsigned char> img;
27
28      lodepng::load_file(png, fichier    );
29      unsigned error = lodepng::decode(img, mLargeur, mHauteur, png);
30
31      if(error) {
32          std::cerr << fichier << " -> png decoder error " << error << ": " <<
    ↪  lodepng_error_text(error) << std::endl;
33          mLargeur = mHauteur = 0;
34          erreur = true;
35      }
36
37      mImage = new unsigned char* [mLargeur];
38      for(int i=0 ; i<mLargeur ; i++) {
39          mImage[i] = new unsigned char[mHauteur];
40          for(int j=0 ; j<mHauteur ; j++) {
41              mImage[i][j] = img[ (i + j*mLargeur)*4 + couche ];
42          }
43      }
44
45      if(!erreur) DEBUG << "image lue : " << fichier << " (" << mLargeur << "x" <<
    ↪  mHauteur << "px) : couche " << couche << std::endl;
46  }
47
48  ImageMatricielle::~ImageMatricielle() {
49      /* Suppression de l'image */
50      for(int i=0 ; i<mLargeur ; i++) {
51          delete[] mImage[i];
52      }
53      delete[] mImage;
54  }
55
56  ImageMatricielle* ImageMatricielle::cloner() {
57      /* Copie de l'image
58       * /!\ libérer la mémoire manuelement
59       */
```

```
60      ImageMatricielle* copie = new ImageMatricielle(mLargeur, mHauteur);
61      for(int i=0 ; i < mLargeur ; i++) {
62          for(int j=0 ; j < mHauteur ; j++) {
63              (*copie)[i][j] = mImage[i][j];
64          }
65      }
66      return copie;
67  }
68
69  /* *************** Setters / Getters *************** */
70
71  unsigned char* ImageMatricielle::operator[](int i) {
72      /* Retourne la ligne de l'image correspondante */
73      return mImage[i];
74  }
75
76  unsigned char& ImageMatricielle::at(int i, int j) {
77      /* Retourne l'élément i,j de façon sécurisée */
78      return mImage[ i % mLargeur ][ j % mHauteur ];
79  }
80
81  int ImageMatricielle::getHauteur() const { return mHauteur; }
82  int ImageMatricielle::getLargeur() const { return mLargeur; }
83
84  unsigned char ImageMatricielle::moyenne() const {
85      /* Retourne la moyenne de teinte des pixels de l'image
86       */
87      int somme = 0;
88      for(int i=0 ; i<mLargeur ; i++) {
89          for(int j=0 ; j<mHauteur ; j++) {
90              somme += mImage[i][j];
91          }
92      }
93      return somme/(mLargeur * mHauteur);
94  }
95
96  void ImageMatricielle::adapterMoyenne(unsigned char val) {
97      /* Décale la moyenne de couleur des pixels pour la faire correspondre à 'val'
98       */
99      int decalage = val - moyenne();
100     for(int i=0 ; i<mLargeur ; i++) {
101         for(int j=0 ; j<mHauteur ; j++) {
102             mImage[i][j] += decalage;
103         }
104     }
105 }
106
107 void ImageMatricielle::remplir(unsigned char val) {
108     /* Remplis l'image de la couleur val
109      */
110     for(int i=0 ; i<mLargeur ; i++) {
111         for(int j=0 ; j<mHauteur ; j++) {
```

```
112             mImage[i][j] = val;
113         }
114     }
115 }
116 
117 /* ************** Compression ************** */
118 
119 std::vector<ImagePart> ImageMatricielle::decouper(int taille) {
120     /* Découpe l'image en sous-images
121      * Les sous parties sont des carrés de côté "taille", le dépassement est ignoré
122      */
123     std::vector<ImagePart> liste;
124     for(int i=0 ; i*taille<mLargeur ; i++) {
125         for(int j=0 ; j*taille<mHauteur ; j++) {
126             liste.push_back( ImagePart(this, i*taille, j*taille, taille) );
127         }
128     }
129     return liste;
130 }
131 
132 std::vector<ImagePart> ImageMatricielle::adapterDecoupe(std::vector<ImagePart>&
    ↪ decoupe, const std::vector<Correspondance>& correspondances) {
133     /* Adapte une découpe triviale à une liste de correspondances :
134      *  - les blocs à spliter vont se spliter
135      *  - la découpe de sortie sera de la même taille que la liste des
    ↪ correspondances
136      */
137     std::list<ImagePart> aTraiter;
138     for(int i=0 ; i < decoupe.size() ; i++) aTraiter.push_back(decoupe[i]); // On
    ↪ transforme l'entrée en liste
139     std::vector<ImagePart> retour;
140     int i = 0;
141 
142     while( !aTraiter.empty() ) {
143         for(int k=0 ; k < correspondances[i].spliter ; k++) { // On splite le nombre
    ↪ de fois demandé
144             std::queue<ImagePart> decoupes = aTraiter.front().spliter(); // Une file
    ↪ de 4 éléments
145             aTraiter.pop_front();
146             std::list<ImagePart>::iterator pos = aTraiter.begin(); // La position où
    ↪ on insère tout
147             while( !decoupes.empty() ) {
148                 aTraiter.insert(pos, decoupes.front()); // On verse la découpe
149                 decoupes.pop();
150             }
151         }
152         retour.push_back( aTraiter.front() );
153         aTraiter.pop_front();
154         i++;
155     }
156     return retour;
157 }
```

```
158
159 IFS ImageMatricielle::chercherIFS(int taillePetit, int tailleGros, const char*
    ↪ message) {
160     /* Recherche l'ifs pour l'image
161      * Entrées :
162      *  - taillePetit : la taille des blocs du petit pavages
163      *  - tailleGros : taille des gros blocs, doit être plus grand que taillePetit
164      * Sortie : IFS
165      *  - correspondances : la liste (respectant les indinces des blocs) des
    ↪ 'Correspondance' a appliquer
166      *  - taillePetit / tailleGros : la taille de découpe
167      */
168     extern int NB_THREADS;
169     extern int NB_MAX_DECOUPE; // Permet de savoir combien de pavages créer
170
171     int tDebut = time(0);
172     if(taillePetit > tailleGros) {
173         std::cerr << "Le pavage n'est pas de la bonne dimension" << std::endl;
174         IFS retour;
175             retour.correspondances = std::vector<Correspondance>();
176             retour.decoupeGros = tailleGros;
177             retour.decoupePetit = taillePetit;
178         return retour;
179     }
180
181     COUT << "Création des pavages ...";
182     std::vector<ImagePart> pavagePetit = decouper(taillePetit); // Les petits carrés
    ↪ dot on cherche une correspondance
183
184     std::vector< std::vector<ImagePart> > pavagesGros;
185     for(int i = 0 ; i <= NB_MAX_DECOUPE ; i++) {
186         pavagesGros.push_back( decouper(tailleGros / std::pow(2, i)) ); // Adapte la
    ↪ découpe en fonction du niveau de récursion
187     }
188     DEBUG << "\rPetits pavés : " << pavagePetit.size() << ", Gros pavés : " <<
    ↪ pavagesGros.size() << std::endl;
189
190     std::vector< std::queue<ImagePart> > taches = decouperTache(pavagePetit,
    ↪ NB_THREADS); // Découpe les tâches
191     std::vector< std::vector<Correspondance> > resultats(NB_THREADS,
    ↪ std::vector<Correspondance>() );
192     std::vector<pthread_t> threads(NB_THREADS, pthread_t());
193     std::vector<ThreadData> datas(NB_THREADS, ThreadData());
194     for(int i=0 ; i<NB_THREADS ; i++) {
195         datas[i].thread_id = i;
196         datas[i].travail = taches[i];
197         datas[i].antecedants = pavagesGros;
198         datas[i].resultat = &resultats[i];
199     }
200     for(int i=0 ; i<NB_THREADS ; i++) {
201         pthread_create(&threads[i], NULL, lancerThread, (void *)&datas[i]);
202     }
```

```
203
204     bool fini(false);
205     while(!fini) {
206         sleep(1);
207         int avancement(pavagePetit.size());
208         fini = true;
209         for(int i=0 ; i<NB_THREADS ; i++) {
210             avancement -= datas[i].travail.size();
211             fini &= datas[i].travail.empty();
212         }
213         COUT << '\r' << message << chargement(avancement, pavagePetit.size(), 20);
214     }
215
216     std::vector<Correspondance> correspondances;
217     for(int i=0 ; i<NB_THREADS ; i++) {
218         for(int j=0 ; j<resultats[i].size() ; j++) {
219             correspondances.push_back(resultats[i][j]);
220         }
221     }
222
223     COUT << '\r' << message << ": " << pavagePetit.size() << '/' <<
    ↪  pavagePetit.size() << " " << "(" << time(0) - tDebut << " secondes)" <<
    ↪  std::endl;
224     IFS retour;
225         retour.correspondances = correspondances;
226         retour.decoupeGros = tailleGros;
227         retour.decoupePetit = taillePetit;
228     debugIfs(retour);
229     return retour;
230 }
231
232 ImageMatricielle ImageMatricielle::appliquerIFS(const IFS& ifs) {
233     /* Applique l'IFS et en retourne le résultat
234      * Entrée : IFS : tout ce qui décrit une image encodée
235      * Sortie : l'image obtenue après application à cet objet
236      */
237     extern int NB_MAX_DECOUPE;
238
239     ImageMatricielle sortie(getLargeur(), getHauteur());
240     std::vector< std::vector<ImagePart> > decoupesEntree;;
241     for(int i = 0 ; i <= NB_MAX_DECOUPE ; i++) {
242         decoupesEntree.push_back( decouper(ifs.decoupeGros / std::pow(2, i)) ); //
    ↪  Adapte la découpe en fonction des redécoupes
243     }
244     std::vector<ImagePart> decoupeSortie = sortie.decouper(ifs.decoupePetit);
245     decoupeSortie = adapterDecoupe(decoupeSortie, ifs.correspondances);
246
247     std::stack<int> dureeSplit; // Chaque élément représente une couche de redécoupe
    ↪  et le nombre d'éléments à y traiter
248     for(int i=0 ; i<ifs.correspondances.size() ; i++) {
249         if( ifs.correspondances[i].spliter > 0 ) { // Au moins une redécoupe est
    ↪  nécessaire
```

```
250             if( dureeSplit.size() > 0 ) {
251                 dureeSplit.top() -= 1;
252             }
253             for(int k=0 ; k < ifs.correspondances[i].spliter ; k++)
254                 dureeSplit.push(3);
255             dureeSplit.top() += 1;
256         }
257 
258         int j = ifs.correspondances[i].bloc;
259         decoupesEntree[ dureeSplit.size() ][j].transformer(decoupeSortie[i],
    ↪  ifs.correspondances[i].transformation);
260 
261         if( dureeSplit.size() > 0 ) {
262             dureeSplit.top() -= 1;
263         }
264         while( dureeSplit.size() > 0 && dureeSplit.top() == 0) dureeSplit.pop(); //
    ↪  On est plus dans une partie réduite
265     }
266     return sortie;
267 }
268 
269 /* ************** Enregistrement ************** */
270 
271 void ImageMatricielle::sauvegarder(const char* fichier) const {
272     /* Enregistre l'image au format png au nom donné
273      *  /!\ Comme on ne connait qu'une couche, l'image est enregistrée en niveaux de
    ↪  gris
274      */
275     std::vector<unsigned char> pixels;
276     for(int j=0 ; j<mHauteur ; j++) {
277         for(int i=0 ; i<mLargeur ; i++) {
278             pixels.push_back(mImage[i][j]);
279             pixels.push_back(mImage[i][j]);
280             pixels.push_back(mImage[i][j]);
281             pixels.push_back(255);
282         }
283     }
284 
285     std::vector<unsigned char> png;
286     unsigned error = lodepng::encode(png, pixels, mLargeur, mHauteur);
287     if(!error) lodepng::save_file(png, fichier);
288 
289     if(error) std::cout << fichier << " -> encoder error " << error << ": "<<
    ↪  lodepng_error_text(error) << std::endl;
290 }
291 
292 /* ************** Traitement ************** */
293 
294 void ImageMatricielle::lisser(int n) {
295     /* Effectue un lissage d'ordre 1 de l'image */
296     ImageMatricielle* copie;
297     for(int k=0 ; k < n ; k++) {
```

```
298         ImageMatricielle* copie = cloner();
299         for(int i=1 ; i < mLargeur-1 ; i++) {
300             for(int j=1 ; j < mHauteur-1 ; j++) {
301                 mImage[i][j] = 0;
302                 mImage[i][j] += (*copie)[i+1][j] / 4;
303                 mImage[i][j] += (*copie)[i][j+1] / 4;
304                 mImage[i][j] += (*copie)[i-1][j] / 4;
305                 mImage[i][j] += (*copie)[i][j-1] / 4;
306             }
307         }
308         delete copie;
309     }
310 }
311
312 void ImageMatricielle::retrecir(int reduction) {
313     /* Réduit la taille de l'image (taille /= reduction) */
314     unsigned char **copie = mImage;
315
316     mImage = new unsigned char* [mLargeur / reduction];
317     for(int i=0 ; i < mLargeur / reduction ; i++) {
318         mImage[i] = new unsigned char[ mHauteur / reduction ];
319         for(int j=0 ; j < mHauteur / reduction ; j++) {
320             mImage[i][j] = copie[i*reduction][j*reduction];
321         }
322     }
323
324     for(int i=0 ; i < mLargeur ; i++) {
325         delete[] copie[i];
326     }
327     delete[] copie;
328
329     mLargeur /= reduction;
330     mHauteur /= reduction;
331 }
```

ImagePart.h

```
1  class ImagePart;
2
3  #ifndef IMAGEPART
4  #define IMAGEPART
5
6  #include <cmath> // pow
7  #include <queue>
8  #include "ImageMatricielle.h"
9  #include "formatIfs.h"
10
11 #define RAD(x) (x*3.14159265/180) // degrés -> randians
12
13 class ImagePart {
14     /* Représente un bout d'image carré
15      * C'est là-dessus que sont effectuées les transformations
```

```
16      */
17
18      public :
19          ImagePart(ImageMatricielle* maman, int x, int y, int taille);
20          ImagePart(int taille);
21          ~ImagePart();
22
23          void sauvegarder(const char* fichier) const;
24
25          /* ********** Getters & Setters ********** */
26
27          void set(int x, int y, unsigned char valeur);
28          void remplir(unsigned char couleur);
29          void encadrer();
30
31          unsigned char at(int i, int j) const;
32          int getTaille() const;
33
34          unsigned char couleurMoyenne() const;
35          float moyenneDifference( const ImagePart& partie, LinReg *decalage = NULL,
    ↪  bool regression = true ) const;
36
37          /* ********** Traitement ********** */
38
39          LinReg chercherLinReg(const ImagePart& partie) const;
40          void appliquerLinReg(const LinReg& droite);
41
42          void transformer(ImagePart& sortie, const Transformation& transfo) const;
43          float chercherTransformation(const ImagePart& origine, Transformation&
    ↪  resultat) const;
44          bool chercherMeilleur(const std::vector<ImagePart>& parties, Correspondance&
    ↪  meilleurCorrespondance) const;
45
46          std::queue<ImagePart> spliter() const;
47
48      private :
49          ImageMatricielle* mImage;   // L'image dont c'est une partie
50          int mTaille;                // La taille du coté de la partie
51          int mX, mY;                 // La position du pixel haut-gauche dans l'image
    ↪  source
52          bool mVirtuel;              // Si oui, la source a été créée uniquement pour
    ↪  simuler un bout d'image
53  };
54
55  #endif
```

ImagePart.cpp

```
1   #include "ImagePart.h"
2
3   /* ************** Constructeur / Destructeur ************** */
4
```

```
5  ImagePart::ImagePart(ImageMatricielle* maman, int x, int y, int taille) :
6      mImage(maman), mTaille(taille), mX(x), mY(y) {
7      /* Pointe une partie d'image.
8       * Entrées :
9       *   - maman : un pointeur sur l'image source
10      *   - x,y : les coordonnées du point haut-gauche du carré
11      *   - taille : la cote du carré
12      */
13     mVirtuel = false;
14 }
15
16 ImagePart::ImagePart(int taille) : mTaille(taille) {
17     /* Crée un "faux" bout d'image. (il n'appartient à aucune image plus grande)
18      * Entrées :
19      *   - taille : la cote du carré
20      */
21     mVirtuel = true;
22     mX = mY = 0;
23     mImage = new ImageMatricielle(mTaille, mTaille);
24 }
25
26 ImagePart::~ImagePart() {
27     if(mVirtuel) { // L'image source a été créée dans le constructeur
28         delete mImage;
29     }
30 }
31
32 /* ************** Setters / Getters ************** */
33
34 void ImagePart::set(int i, int j, unsigned char val) {
35     /* Modifie la valeur d'un élément dans la parties
36      * /!\ Interdit les modifications de l'extérieur
37      * Entrées :
38      *   - i,j : les coordonnées du pixel à modifier, dans [| 0, taille [|
39      *   - val : la valeur à lui attribuer
40      */
41     if( ( i>=0 && i<mTaille ) &&  ( j>=0 && j<mTaille ) ) // Ca ne dépasse pas de la
    ↪  parcelle
42     if( i+mX < mImage->getLargeur() && j+mY < mImage->getHauteur() ) // ca ne
    ↪  dépasse pas de l'image
43         (*mImage)[i+mX][j+mY] = val;
44 }
45
46 unsigned char ImagePart::at(int i, int j) const {
47     /* Retourne la valeur aux coordonnées données (i,j)
48      * Si les coordonnées dépassent du blocs mais restent dans l'image ça marche
    ↪  quand même
49      */
50     int ix = i+mX;
51     int jy = j+mY;
52
53     if(ix < 0) ix = 0; // Si ca dépasse on se projete sur le bord
```

```
54      else if(ix >= mImage->getLargeur()) ix = mImage->getLargeur() - 1;
55
56      if(jy < 0) jy = 0; // Si ca dépasse on se projete sur le bord
57      else if(jy >= mImage->getHauteur()) jy = mImage->getHauteur() - 1;
58
59      return (*mImage)[ix][jy];
60  }
61
62  int ImagePart::getTaille() const { return mTaille; } // La côte du carré
63
64  unsigned char ImagePart::couleurMoyenne() const {
65      /* La moyenne des couleurs représentées sur le bout d'image */
66      unsigned int somme = 0;
67      for(int i=0 ; i<mTaille ; i++) {
68          for(int j=0 ; j<mTaille ; j++) {
69              somme += at(i, j);
70          }
71      }
72      return somme/(mTaille*mTaille);
73  }
74
75  void ImagePart::remplir(unsigned char couleur) {
76      /* Remplis le bout d'image avec une couleur uniforme */
77      for( int i = 0 ; i < mTaille ; i++ ) {
78      for( int j = 0 ; j < mTaille ; j++ ) {
79              set(i, j, couleur);
80          }
81      }
82  }
83
84  void ImagePart::encadrer() {
85      /* Crée un bord noir et un milieu blanc
86      */
87      for( int i = 0 ; i < mTaille ; i++ ) {
88          bool bordH = ( i == 0 ) || ( i == mTaille - 1 );
89          for( int j = 0 ; j < mTaille ; j++ ) {
90              bool bordV = ( j == 0 ) || ( j == mTaille - 1 );
91              int couleur = ( bordH || bordV ) ? 0 : 255;
92              set(i, j, couleur);
93          }
94      }
95  }
96
97  /* ************** Régression linéaire ************** */
98
99  LinReg ImagePart::chercherLinReg(const ImagePart& X) const {
100     /* On cherche une fonction affine f telle que f(X) ~= Y
101      */
102     const ImagePart &Y = *this;
103     float sumX, sumY, sumXY, sumXX; // On a besoins de calculer 4 grosses sommes
104     sumX = sumY = sumXY = sumXX = 1; // On évite les divisions par 0
105     for(int i=0 ;  i<mTaille ; i++) {
```

```
106         for(int j=0 ; j<mTaille ; j++) {
107             sumX += X.at(i, j);
108             sumY += Y.at(i, j);
109             sumXY += X.at(i, j) * Y.at(i, j);
110             sumXX += std::pow(X.at(i, j), 2);
111         }
112     }
113     float n = mTaille*mTaille;
114     LinReg retour;
115         retour.a = ( (sumX*sumY/n) - sumXY ) / ( (sumX*sumX/n) - sumXX );
116         retour.a = decode16bFloat( Flotant16b( retour.a ) );
117         retour.b = ( sumY - (retour.a*sumX) ) / n;
118     return retour;
119 }
120 
121 void ImagePart::appliquerLinReg(const LinReg& droite) {
122     /* Applique la fonction affine droite à tous les pixels de la parcelle */
123     for(int i = 0 ; i < mTaille ; i++) {
124         for(int j=0 ; j<mTaille ; j++) {
125             set(i, j, couleurLinReg(droite, at(i, j)));
126         }
127     }
128 }
129 
130 /* *************** Transformations *************** */
131 
132 void ImagePart::transformer(ImagePart& imgSortie, const Transformation& transfo)
    ↪ const {
133     /* Applique une transformation linéaire sur la partie d'image
134      * Entrées :
135      *   - imgSortie : le bout d'image dans lequel on enregistre le résultat de la
    ↪ transformation
136      *   - transfo : un type de transformation
137      * /!\ Il vaut mieux accompagner cette fonction d'un brouillon de calculs
138      */
139     int a = imgSortie.getTaille();
140     float grandissement = float(mTaille) / a;
141     float rapportx = cos(RAD(transfo.rotation)); // r*e^(i*theta)
142     float rapporty = sin(RAD(transfo.rotation));
143     float centre = a / 2; // Centre de la rotation, en x et en y
144 
145     int is, js;
146     for(is=0 ; is<a ; is++) {
147         for(js=0 ; js<a ; js++) {
148             int i = rint( grandissement* (rapportx*(is-centre) -
    ↪ rapporty*(js-centre) + centre) ); // Calculs des parties imaginaires et réelles
149             int j = rint( grandissement* (rapporty*(is-centre) +
    ↪ rapportx*(js-centre) + centre) );
150             imgSortie.set(is, js, couleurLinReg(transfo.droite, at(i, j))); // On a
    ↪ trouvé le point correspondant, on rajoute le décalage de couleur
151         }
152     }
```

```
153 }
154
155 float ImagePart::chercherTransformation(const ImagePart& origine, Transformation&
    ↪ min) const {
156     /* Cherche la meilleur transformation de origine pour correspondre à cet objet
157      * Entrées :
158      *   - origine : l'image qui subis les transformations
159      *   - min : un type Transformation
160      * Sorties :
161      *   - retourne la variance correspondant à la transformation trouvée
162      *   - modifie min, la transformation trouvée
163      */
164     extern float SEUIL_LISSAGE, SEUIL_VARIANCE;
165
166     Transformation max = ROTATION(360); // max sert juste de borne mais ne peut pas
    ↪ être la valeur de retour
167     Transformation mid = ROTATION(0);
168
169     min.rotation = 0;
170     min.droite.a = 0;                    // Vérifie la fonction constante
171     min.droite.b = couleurMoyenne();    // Donne la couleur exacte sur les bouts
    ↪ lisses, crée de la redondanec
172
173     ImagePart img(mTaille);
174
175     origine.transformer(img, max);
176     float varmax = moyenneDifference(img); // max sert juste de borne
177
178     origine.transformer(img, min);
179     float varmin = moyenneDifference(img, &min.droite, false); // Il faut donner une
    ↪ valeur à min.droite au cas où il est retourné
180
181     /* Application de la dichotomie :
182      * La variance en fonction de l'angle n'est pas (/rarement) monotone,
    ↪ l'algorithme tend vers un minimum local
183      * A chaque itération on choisis l'angle moitié, puis mon garde l'angle de
    ↪ variance la plus faible et ce nouvel angle
184      *  -> min < max gardent leur ordre mais le qualificatif n'est relatif qu'à leur
    ↪ angle
185      *  -> de toutes facons, à la fin varmin ~= varmax
186      */
187     LinReg droite;
188     if(varmin > SEUIL_LISSAGE) {
189         while(max.rotation - min.rotation > 5 && varmin > SEUIL_VARIANCE) {
190             mid.rotation = (max.rotation + min.rotation) / 2; // On prend le milieu
    ↪ et on calcul la transformation
191             origine.transformer(img, mid);
192             float variance = moyenneDifference(img, &droite);
193             if(varmin < varmax) {
194                 max.rotation = mid.rotation;
195                 max.droite = droite;
196                 varmax = variance;
```

```
197             }
198             else {
199                 min.rotation = mid.rotation;
200                 min.droite = droite;
201                 varmin = variance;
202             }
203         }
204     }
205     return varmin; // On retourne la variance obtenue pour éviter de refaire le
    ↪  calcul
206 }
207
208 bool ImagePart::chercherMeilleur(const std::vector<ImagePart>& parties,
    ↪  Correspondance& meilleurCorrespondance) const {
209     /* Cherche la meilleur image d'origine pour une transformation
210      * Entrée :
211      *   - parties : un tableau de bouts d'images
212      *   - meilleurCorrespondance : un type Correspondance :
213      *       - bloc : l'indice du bout d'image choisis dans "parties"
214      *       - transformation : la transformation optimale pour ce bloc
215      * Sortie : true si la variance est considérée comme correcte
216      */
217     extern float SEUIL_LISSAGE, SEUIL_VARIANCE;
218     extern float SEUIL_DECOUPE;
219
220     int n = parties.size();
221     Transformation transfo;
222     float variance, varianceMin = chercherTransformation(parties[0], transfo); //
    ↪  Donne une valeur initiale à varianceMin
223
224     meilleurCorrespondance.transformation = transfo;
225     meilleurCorrespondance.bloc = 0;
226     meilleurCorrespondance.spliter  = 0;
227
228     // On fait une recherche de minimum sur la variance
229     for(int i=1 ; i<n && (varianceMin>SEUIL_LISSAGE ||
    ↪  meilleurCorrespondance.transformation.droite.a != 0) &&
    ↪  varianceMin>SEUIL_VARIANCE ; i++) {
230         variance = chercherTransformation(parties[i], transfo);
231         if(varianceMin > variance) {
232             meilleurCorrespondance.bloc = i;
233             meilleurCorrespondance.transformation = transfo;
234             varianceMin = variance;
235         }
236     }
237
238     return varianceMin < SEUIL_DECOUPE;
239 }
240
241 float ImagePart::moyenneDifference(const ImagePart& B, LinReg *droite, bool
    ↪  regression) const {
242     /* Compare deux images :
```

```
243         * Etudie la moyenne des "distances" entre les pixels
244         * La moyenne de chaque image est ajustée par régression linéaire
245         * Entrées :
246         *  - B : l'image à laquelle this est comparé
247         *  - decalage : un pointeur sur une variable qui prendra la valeur de la
    ↪ régression linéaire appliquée
248         *     -> decalage doit être appliqué à B pour qu'il ressemble à l'objet
    ↪ courrant
249         *   - regression : doit faire une regression linéaire ?
250         *     -> sinon il utilise celle passée en argument (ne doit pas être NULL)
251         */
252        bool droiteToDestroy = false; // true s'il faut désalouer la valeur de "droite"
253        if(droite == NULL) {
254            droite = new LinReg();
255            droiteToDestroy = true;
256        }
257
258        if(regression)
259            *droite = chercherLinReg(B); // On cherche une transfo linéaire
260
261        float sumCarre = 0;
262        for(int i=0 ; i<mTaille ; i++) {
263            for(int j=0 ; j<mTaille ; j++) {
264                int ecart = couleurLinReg(*droite, B.at(i, j)) - at(i, j); // On utilise
    ↪ la régression linéaire
265                sumCarre += std::pow(ecart, 4);
266            }
267        }
268
269        if( droiteToDestroy ) delete droite; // Droite valait NULL
270        float n = mTaille*mTaille;
271        return sumCarre / n;
272    }
273
274    std::queue<ImagePart> ImagePart::spliter() const {
275        /* Découpe la partie d'image en 4 sous-parties
276         * /!\ Pour une entrée de taille impaire le pixel du milieu sera pris en compte
    ↪ deux fois
277         * Si la découpe échoue, la valeur de retour est cet objet
278         */
279        int nvlleTaille = mTaille / 2;
280        int midX = mX + nvlleTaille;
281        int midY = mY + nvlleTaille;
282        std::queue<ImagePart> retour;
283            if(mTaille == 1) {
284                DEBUG << "Tentative de découpe de taille 1" << std::endl;
285                retour.push(*this);
286                return retour;
287            }
288            retour.push(ImagePart(mImage, mX, mY, nvlleTaille));
289            retour.push(ImagePart(mImage, midX, mY, mTaille - nvlleTaille));
290            retour.push(ImagePart(mImage, mX, midY, mTaille - nvlleTaille));
```

```
291             retour.push(ImagePart(mImage, midX, midY, nvlleTaille));
292         return retour;
293     }
294
295     void ImagePart::sauvegarder(const char* fichier) const {
296         /* Enregistre le bout d'image au format png */
297         ImageMatricielle image(mTaille, mTaille);
298         for(int i=0 ; i<mTaille ; i++) {
299             for(int j=0 ; j<mTaille ; j++) {
300                 image[i][j] = at(i, j);
301             }
302         }
303         image.sauvegarder(fichier);
304     }
```

ImageFractale.h

```
1   class ImageFractale;
2
3   #ifndef IMAGE_FRACTALE
4   #define IMAGE_FRACTALE
5
6   #include <fstream>
7   #include <vector>
8   #include <cmath> // ceil
9
10  #include "debug.h"
11  #include "formatIfs.h"
12  #include "formatFichier.h"
13  #include "ImageMatricielle.h"
14
15
16  class ImageFractale {
17      /* Décrit une image fractale
18       * Toutes les informations nécessaires au décodage sont contenuent dans la
    ↪   classe
19       */
20
21      friend class FigureFractale;
22
23      public :
24          ImageFractale();
25          // Constructeur à partir d'un fichier .ifs
26          ImageFractale(const char* fichier);
27
28          // Enregistre au format .png
29          void exporter(const char* fichier);
30
31          // Importe le format .png
32          static ImageFractale compresser(const char* fichier, int precisionPetit, int
    ↪   precisionGros, bool couleur = false, bool transparence = false);
33
```

```
34          // Enregistre l'image au format ifs
35          void enregistrer(const char* fichier) const;
36
37          // Crée une image schématisant la découpe
38          void debugSplit() const;
39
40          void grandir(int grandissement);
41          void retrecir(int reduction);
42
43          // Getters
44          int getLargeur() const;
45          int getHauteur() const;
46          bool isCouleur() const;
47          bool isTransparent() const;
48
49      protected :
50          int mLargeur, mHauteur;                 // Les dimensions de l'image
51          bool mCouleur, mTransparence;         // L'image est en couleur ?
    ↪  Transparente ?
52          std::vector<unsigned char> mMoyenne;// La moyenne de teinte des couches :
    ↪  gris/RGB puis alpha
53          std::vector<IFS> mIfs;                // Les ifs de chaque couche
54  };
55
56  #endif
```

ImageFractale.cpp

```
1   #include "ImageFractale.h"
2
3   /* *************** Ouverture *************** */
4
5   ImageFractale::ImageFractale() {
6       mLargeur = mHauteur = 0;
7       mCouleur = mTransparence = false;
8       mMoyenne = std::vector<unsigned char>();
9       mIfs = std::vector<IFS>();
10  }
11
12  ImageFractale::ImageFractale(const char* fichier) : mMoyenne(std::vector<unsigned
    ↪  char>()), mIfs(std::vector<IFS>()) {
13      std::ifstream f(fichier, std::ios::in | std::ios::binary);
14
15      Pack_Entete entete;
16      f.read((char*)&entete, sizeof(Pack_Entete));
17          mLargeur = entete.largeur;
18          mHauteur = entete.hauteur;
19          mCouleur = entete.couleur;
20          mTransparence = entete.transparence;
21
22      int nbCouches = mCouleur ? 3 : 1; // Calcul du nombre de couches
23      if(mTransparence) nbCouches += 1;
```

```
24
25      for(int i=0 ; i<nbCouches ; i++) {
26          Pack_IFS ifs;
27          f.read((char*)&ifs, sizeof(Pack_IFS));
28          mIfs.push_back( unpack_IFS(ifs) );
29          mMoyenne.push_back( unpack_moyenne(ifs) );
30
31          int nbBlocs = std::ceil((float)mLargeur/ifs.decoupePetit) *
    ↪  std::ceil((float)mHauteur/ifs.decoupePetit);
32          lireCorrespondancesFichier(f, nbBlocs, mIfs[i].correspondances);
33      }
34
35      f.close();
36  }
37
38  ImageFractale ImageFractale::compresser(const char* fichier, int precisionPetit, int
    ↪  precisionGros, bool couleur, bool transparence) {
39      /* Retourne la compression d'un fichier au format ImageFractale
40       * Entrées :
41       *  - fichier : l'adresse du fichier (.png) à lire
42       *  - precisionPetit : la taille des petits blocs
43       *      | Réduire cette grandeur augmente la qualité
44       *      | Réduire cette grandeur augmentera la taille du fichier final
45       *  - precisionGros : la taille des gros blocs
46       *      | /!\ Doit être plus grand que 'taillePetit' (recommandation : 150%)
47       *      | Réduire cette grandeur augmente la qualitéen général (cf.
    ↪  recommandation)
48       *      | Réduire cette grandeur augmente la durée de calculs
49       *  - couleur : true si l'image est en couleur (temps de calculs x3)
50       *  - transparence : true s'il y a une couche alpha
51       */
52      ImageMatricielle image(fichier, 0);
53      ImageFractale retour;
54          retour.mLargeur = image.getLargeur();
55          retour.mHauteur = image.getHauteur();
56          retour.mCouleur = couleur;
57          retour.mTransparence = transparence;
58
59      COUT << "Dimension de l'image : " << retour.mLargeur << "x" << retour.mHauteur
    ↪  << std::endl;
60
61      if(couleur) {
62          const char* message[] = {" - Couche rouge ", " - Couche verte ", " - Couche
    ↪  bleue "};
63          for(int i=0 ; i<3 ; i++) {
64              ImageMatricielle imageTr(fichier, i);
65              retour.mIfs.push_back( imageTr.chercherIFS(precisionPetit,
    ↪  precisionGros, message[i]) );
66              retour.mMoyenne.push_back( imageTr.moyenne() );
67          }
68      }
69      else {
```

```
70          retour.mIfs.push_back( image.chercherIFS(precisionPetit, precisionGros, " -
    ↪  Couche NVDG  ") );
71          retour.mMoyenne.push_back( image.moyenne() );
72      }
73      if(transparence) {
74          ImageMatricielle imageTr(fichier, 3);
75          retour.mIfs.push_back( imageTr.chercherIFS(precisionPetit, precisionGros, "
    ↪  - Couche alpha") );
76          retour.mMoyenne.push_back( imageTr.moyenne() );
77      }
78
79      return retour;
80  }
81
82  /* ************** Enregistrement ************** */
83
84  void ImageFractale::enregistrer(const char* fichier) const {
85      std::ofstream f(fichier, std::ios::trunc | std::ios::binary);
86      if (!f.is_open()) std::cout << "Impossible d'ouvrir le fichier '" << fichier <<
    ↪  "'" << std::endl;
87
88      Pack_Entete entete = packer_entete(*this); // Récupère les données binaires de
    ↪  l'en-tête
89      f.write((char*)&entete, sizeof(Pack_Entete));
90
91      for(int i=0 ; i < mIfs.size() ; i++) {
92          Pack_IFS ifs = packer_ifs(mIfs[i], mMoyenne[i]);
93          f.write((char*)&ifs, sizeof(Pack_IFS));
94
95          const std::vector<Correspondance> &mCorrespondances =
    ↪  mIfs[i].correspondances;
96          std::cout << mCorrespondances.size() << std::endl;
97          for(int j=0 ; j < mCorrespondances.size() ; j++) {
98              Pack_Correspondance correspondance =
    ↪  packer_correspondance(mCorrespondances[j]);
99              f.write((char*)&correspondance, SIZEOF_PACK_CORRESPONDANCE);
100         }
101     }
102
103     f.close();
104 }
105
106 void ImageFractale::exporter(const char* fichier) {
107     /* Crée un rendu de l'image et l'exporte au format (.png)
108     */
109     extern int ITERATIONS_DECOMPRESSION, QUALITE_DECOMPRESSION;
110
111     grandir(QUALITE_DECOMPRESSION);
112
113     std::vector<ImageMatricielle*> couche(mIfs.size(), NULL);
114     for(int i=0 ; i < mIfs.size() ; i++) {
115         couche[i] = new ImageMatricielle(mLargeur, mHauteur);
```

```
116             couche[i]->remplir(mMoyenne[i]);
117             for(int k=0 ; k < ITERATIONS_DECOMPRESSION ; k++) {
118                 ImageMatricielle *nouveau =  new
    ↪  ImageMatricielle(couche[i]->appliquerIFS(mIfs[i]));
119                 delete couche[i]; // On désaloue pour ne pas créer de fuite de mémoire
120                 couche[i] = nouveau;
121             }
122             couche[i]->lisser(QUALITE_DECOMPRESSION-1);
123             couche[i]->retrecir(QUALITE_DECOMPRESSION);
124         }
125 
126         retrecir(QUALITE_DECOMPRESSION);
127 
128         std::vector<unsigned char> pixels;
129         for(int j=0 ; j<mHauteur ; j++) {
130             for(int i=0 ; i<mLargeur ; i++) {
131                 if(mCouleur) {
132                     for(int n=0 ; n<3 ; n++) // Ajoute les 3 couleurs
133                         pixels.push_back( (unsigned char)( (*couche[n])[i][j] ) );
134                 }
135                 else{
136                     for(int n=0 ; n<3 ; n++) // Pose 3 fois la même couleur
137                         pixels.push_back( (unsigned char)( (*couche[0])[i][j] ) );
138                 }
139                 if(mTransparence) pixels.push_back( (unsigned char)(
    ↪  (*couche[couche.size()-1])[i][j] ) );
140                 else pixels.push_back(255);
141             }
142         }
143 
144         std::vector<unsigned char> png;
145         unsigned error = lodepng::encode(png, pixels, mLargeur, mHauteur);
146         if(!error) lodepng::save_file(png, fichier);
147 
148         if(error) std::cout << fichier << " -> png encoder error " << error << ": "<<
    ↪  lodepng_error_text(error) << std::endl;
149     }
150 
151     void ImageFractale::debugSplit() const {
152         extern const char* DOSSIER_DEBUG;
153         for( int i = 0 ; i < mIfs.size() ; i++ ) {
154             ImageMatricielle img(mLargeur, mHauteur);
155             std::vector<ImagePart> parties = img.decouper( mIfs[i].decoupePetit );
156             parties = ImageMatricielle::adapterDecoupe( parties, mIfs[i].correspondances
    ↪  );
157             for( int j = 0 ; j < parties.size() ; j++ ) {
158                 parties[j].encadrer();
159             }
160 
161             std::stringstream nom;
162             nom << DOSSIER_DEBUG << "grille-" << i << ".png";
163             img.sauvegarder( nom.str().c_str() );
```

```
164         }
165     }
166
167     void ImageFractale::grandir(int grandissement) {
168         /* Augmente la taille de l'image */
169         mHauteur *= grandissement;
170         mLargeur *= grandissement;
171         for(int i=0 ; i < mIfs.size() ; i++) {
172             mIfs[i].decoupeGros *= grandissement;
173             mIfs[i].decoupePetit *= grandissement;
174         }
175     }
176
177     void ImageFractale::retrecir(int reduction) {
178         /* Réduit la taille de l'image */
179         mHauteur /= reduction;
180         mLargeur /= reduction;
181         for(int i=0 ; i < mIfs.size() ; i++) {
182             mIfs[i].decoupeGros /= reduction;
183             mIfs[i].decoupePetit /= reduction;
184         }
185     }
186
187     /* ************** Getters / Setters ************** */
188
189     int ImageFractale::getLargeur() const { return mLargeur; }
190     int ImageFractale::getHauteur() const { return mHauteur; }
191     bool ImageFractale::isCouleur() const { return mCouleur; }
192     bool ImageFractale::isTransparent() const { return mTransparence; }
```

formatIfs.h

```
1   /*
2    * Contient les structures nécéssaires à la représentation d'une IFS
3    */
4
5   struct LinReg;
6   struct Transformation;
7   struct Correspondance;
8   struct IFS;
9
10  #ifndef FORMAT
11  #define FORMAT
12
13  #include <iostream>
14  #include <vector>
15  #include "ImageMatricielle.h"
16  #include "ImageFractale.h"
17
18  /* ************** Définition de structures ************** */
19
20  typedef struct LinReg LinReg;
```

```
21  struct LinReg {
22      /* Représente une fonction affine
23       * f(x) = a*x + b
24       */
25      float a, b;
26  };
27
28  typedef struct Transformation Transformation;
29  struct Transformation {
30      /* Décrit une transformation affine appliquée à un bloc
31       *  - translation :
32       *  - rotation : en degrés
33       *  - decalage : le décalage de couleur à ajouter
34       */
35      unsigned short int rotation;   // Rotation par rapport au centre de la parcelle
    ↪  en  (0 à 360)
36      LinReg droite;                 // La correction à appliquer à la couleur
37  };
38  #define ROTATION(rot) {rot, {1,0}} // Initialisation d'une transformation de type
    ↪  rotation
39
40  typedef struct Correspondance Correspondance;
41  struct Correspondance {
42      /* Décrit un couple bloc/transformation */
43      int spliter; // Le nombre de fois qu'il faut découper le bloc avant d'y
    ↪  appliquer la transformation
44      int bloc; // l'indice du bloc à choisir dans l'image
45      Transformation transformation; // le type de transformation à y appliquer
46  };
47
48  typedef struct IFS IFS;
49  struct IFS {
50      /* Décrit tout ce qu'il faut savoir sur un ifs */
51      int decoupePetit; // Taille de découpe pour les images
52      int decoupeGros;  // Taille de découpe pour les antécédants
53      std::vector<Correspondance> correspondances; // L'ifs : la liste de la
    ↪  transformation/antécédant de chaque bloc
54  };
55
56  /* ************** Fonctions ************** */
57
58  unsigned char couleurLinReg(const LinReg& droite, unsigned char couleur);
59
60  #endif
```

formatIfs.cpp

```
1  #include "formatIfs.h"
2
3  unsigned char couleurLinReg(const LinReg& droite, unsigned char couleur) {
4      /* Donne l'image d'une couleur par une fonction affine
5       *  - droite : la fonction affine
```

```
6         *  - couleur : la couleur
7         */
8       return std::min(255, std::max(0, int(droite.a*couleur + droite.b)));
9   }
```

formatFichier.h

```
1   /*
2    * Définition des structures et des fonctions utilisées pour enregistrer dans un
     ↪  fichier
3    */
4
5   struct Flotant16b;
6   struct Pack_Entete;
7   struct Pack_IFS;
8   struct Pack_Correspondance;
9
10  #ifndef FORMAT_FICHIER
11  #define FORMAT_FICHIER
12
13  #include <cmath> // pow, frexp
14  #include "ImageFractale.h"
15
16  /* ************** Structures ************** */
17
18  #define DECALAGE_EXPOSANT 2 // Plus précis pour les petites valeurs
19  #define TAILLE_MANTISSE 1024
20
21  struct Flotant16b {
22      /* Représente un flotant sur 16 bits */
23      short signed int exp       : 5;
24      short signed int mantisse  : 11; // log2(TAILLE_MANTISSE-1) (signé)
25
26      Flotant16b() { mantisse = exp = 0; }    // Un constructeur élémentaire
27      Flotant16b(float x) {                    // Constructeur à partir d'un float 32
         ↪  bits
28          int e;
29          mantisse = TAILLE_MANTISSE * std::frexp(x, &e);
30          exp = e - DECALAGE_EXPOSANT;
31      }
32  };
33
34  float decode16bFloat(Flotant16b x);
35
36  struct Pack_Entete {
37      /* L'en-tête du document
38       *  sur 32 bits
39       */
40      unsigned int largeur    :15;
41      unsigned int hauteur    :15;
42      bool couleur            :1;
43      bool transparence       :1;
```

```
};

struct Pack_IFS {
    /* L'en-tête d'une couche
     *  sur 32 bits
     */
    unsigned int decoupeGros    :12;
    unsigned int decoupePetit   :12;
    unsigned char moyenne        :8;
};

struct Pack_Correspondance {
    /* Une correspondance
     *  + 32 bits
     *  + a (float16b) -> 48 bits
     */
    Flotant16b a;              //:16;    // Flotant 16 bits
    signed short int b           :16;    // -512 à 512
    unsigned int bloc            :20;    // Limité à 1M de gros blocs
    unsigned int spliter         :3;     // Limité à 7 splitages consécutifs
    unsigned int rotation        :9;     // 0 à 255, proportionel à l'angle
};

/* *************** Fonctions de mise en paquets *************** */

#define WARNING_PACKING true // Doit t'on afficher un message dans la console si une
 ↪  entrée semble dangereuse

Pack_Entete packer_entete(const ImageFractale&);
Pack_IFS packer_ifs(const IFS& ifs, unsigned char moyenne = 0);
Pack_Correspondance packer_correspondance(const Correspondance&);

unsigned char unpack_moyenne(const Pack_IFS&);
IFS unpack_IFS(const Pack_IFS&);
Correspondance unpack_correspondance(const Pack_Correspondance&);

/* *************** Fonctions de lecture *************** */

void lireCorrespondancesFichier(std::ifstream& fichier, int nombre,
 ↪  std::vector<Correspondance>& sortie);

#endif
```

formatFichier.cpp

```
#include "formatFichier.h"

/* *************** Flotant 16 bits *************** */

float decode16bFloat(Flotant16b x) {
    /* Retourne un flotant 32 bits (natif) à partir d'un flotant 16 bits */
    return x.mantisse * std::pow(2, x.exp + DECALAGE_EXPOSANT) / TAILLE_MANTISSE;
```

```
}

/* *************** Fonctions de mise en paquets ************** */
/* Toutes ces fonctions ont pour rôle de convertir les structures de données dans un
 ↪  format optimal en occupation mémoire */

Pack_Entete packer_entete(const ImageFractale& image) {
    Pack_Entete retour;
        retour.largeur = image.getLargeur();
        retour.hauteur = image.getHauteur();
        retour.couleur = image.isCouleur();
        retour.transparence = image.isTransparent();
    return retour;
}

Pack_IFS packer_ifs(const IFS& ifs, unsigned char moyenne) {
    Pack_IFS retour;
        retour.decoupeGros = ifs.decoupeGros;
        retour.decoupePetit = ifs.decoupePetit;
        retour.moyenne = moyenne;
    return retour;
}

Pack_Correspondance packer_correspondance(const Correspondance& correspondance) {
    Pack_Correspondance retour;
        retour.bloc = correspondance.bloc;
        retour.spliter = correspondance.spliter;
        retour.rotation = correspondance.transformation.rotation;
        retour.a = correspondance.transformation.droite.a;
        retour.b = correspondance.transformation.droite.b;
    return retour;
}

/* *************** Fonctions d'extraction des paquets *************** */

unsigned char unpack_moyenne(const Pack_IFS& paquet) {
    return paquet.moyenne;
}

IFS unpack_IFS(const Pack_IFS& paquet) {
    IFS retour;
        retour.decoupeGros = paquet.decoupeGros;
        retour.decoupePetit = paquet.decoupePetit;
    return retour;
}

Correspondance unpack_correspondance(const Pack_Correspondance& paquet) {
    Correspondance retour;
        retour.bloc = paquet.bloc;
        retour.spliter = paquet.spliter;
        retour.transformation.rotation = paquet.rotation;
        retour.transformation.droite.b = paquet.b;
```

```
59          retour.transformation.droite.a = decode16bFloat(paquet.a);
60      return retour;
61  }
62
63  /* ************** Fonctions de lecture ************** */
64
65  void lireCorrespondancesFichier(std::ifstream& fichier, int nombre,
    ↪  std::vector<Correspondance>& sortie) {
66      /* Lis des correspondances dans un fichier en prenant en compte les splitages
67       * Entrées :
68       *  - fichier : le fichier dans lequel lire
69       *  - nombre : le nombre de correspondances -de bases- à lire
70       *  - sortie : le vecteur dans lequel verser les résultat
71       */
72      for(int i=0 ; i < nombre ; i++) {
73          Pack_Correspondance correspondance;
74          fichier.read((char*)&correspondance, SIZEOF_PACK_CORRESPONDANCE);
75          sortie.push_back( unpack_correspondance(correspondance) );
76          if( sortie.back().spliter > 0 ) {
77              lireCorrespondancesFichier(fichier, 3*sortie.back().spliter , sortie);
    ↪  // Le splitage a engendré des nouveaux blocs
78          }
79      }
80  }
```

multithread.h

```
1   /*
2    * Les fonctions qui sont utilisées pour découper le travail en plusieurs threads
3    */
4
5   struct ThreadData;
6
7   #ifndef MULTITHREAD
8   #define MULTITHREAD
9
10  #include "formatIfs.h"
11  #include "ImagePart.h"
12  #include <queue>
13
14  typedef struct ThreadData ThreadData;
15  struct ThreadData { //TODO: Passer par des références
16      /* Les infos qui sont utilisée par le thread
17      */
18      int thread_id; // Un identifiant unique
19      std::queue<ImagePart> travail; // La liste des bout d'images pour lesquel le
    ↪  thread doit trouver une correspondance
20      std::vector< std::vector<ImagePart> > antecedants; // Là où sont cherchées les
    ↪  correspondances, en fonction de la redécoupe
21      std::vector<Correspondance>* resultat; // Le tableau qui contiendra les
    ↪  résultats  /!\ doit être envoyé vide
22  };
```

```
23
24  void *lancerThread(void *data);
25  std::vector<Correspondance> chercherCorrespondances(std::queue<ImagePart>& travail,
    ↪  const std::vector< std::vector<ImagePart> >& antecedants, int redecoupe = 0);
26  std::vector< std::queue<ImagePart> > decouperTache(const std::vector<ImagePart>&
    ↪  tache, int nombre);
27
28  #endif //MULTITHREAD
```

multithread.cpp

```
1   #include "multithread.h"
2
3   void *lancerThread(void *t_data) {
4       /* Lance un nouveau thread
5        * Le thread calcul les sources pour chaque élément de t_data->travail puis
    ↪  s'arrête
6        * Entrée :
7        *  - t_data : pointeur sur un élément de ThreadData
8        * Sortie : dans t_data->resultat
9        */
10      ThreadData *data = (ThreadData *) t_data;
11      *data->resultat = chercherCorrespondances(data->travail, data->antecedants);
12      DEBUG << "Thread " << data->thread_id << " terminé" << std::endl;
13      pthread_exit(NULL);
14  }
15
16  std::vector<Correspondance> chercherCorrespondances(std::queue<ImagePart>& travail,
    ↪  const std::vector< std::vector<ImagePart> >& antecedants, int redecoupe ) {
17      /* Cherche les correspondances pour un travail donné
18       * Entrées :
19       *  - antecedants : où chercher la correspondance en fonction du nombre de
    ↪  redécoupes
20       *  - redecoupe : le nombre de splitages réalisés pour donner 'travail'
21       *  - références sur travail
22       * Sortie :
23       *  - les correspondances sont retournées
24       *  - 'travail' est régulièrement élagué
25       */
26      extern int TAILLE_MIN_DECOUPE;
27      std::vector<Correspondance> retour;
28      Correspondance correspondanceTrouvee;
29      while(!travail.empty()) {
30          bool satisfaisant = travail.front().chercherMeilleur(antecedants[redecoupe],
    ↪  correspondanceTrouvee);
31          if( satisfaisant || TAILLE_MIN_DECOUPE > travail.front().getTaille() ) {
32              retour.push_back( correspondanceTrouvee ); // On est satisfait, on
    ↪  conserve le resultat
33          }
34          else {
35              std::queue<ImagePart> decoupe = travail.front().spliter(); // On
    ↪  récupère les nouvelles parties
```

```
36              std::vector<Correspondance> insertion = chercherCorrespondances(decoupe,
    ↪  antecedants, redecoupe+1); // On cherche les correspondances de ces bouts
37              insertion[0].spliter++; // On rappel qu'on a dût spliter une fois
38              for(int i=0 ; i < insertion.size() ; i++) {
39                  retour.push_back( insertion[i] ); // On rajoute les nouvelles
    ↪  correspondances
40              }
41          }
42          travail.pop();
43      }
44      return retour;
45  }
46
47  std::vector< std::queue<ImagePart> > decouperTache(const std::vector<ImagePart>&
    ↪  tache, int nombre) {
48      /* Découpe un vector en plusieurs parties, en respectant l'ordre
49       * Entrées :
50       *  - tache : un vector
51       *  - nombre : le nombre de vectors souhaités en sortie
52       * Sortie : /!\ les tableaux en sortie ne sont pas tous nécéssairement de la
    ↪  même taille
53       */
54      int taille = (tache.size() / nombre) + 1; // On se fait pas chier a arrondir
55      std::vector< std::queue<ImagePart> > sortie(nombre, std::queue<ImagePart>());
56      for(int i=0 ; i<nombre ; i++) {
57          for(int j=0 ; j < taille && j+(taille*i) < tache.size() ; j++) {
58              sortie[i].push( tache[j+(taille*i)] );
59          }
60      }
61      return sortie;
62  }
```

FigureFractale.cpp

```
1   #include "FigureFractale.h"
2
3   Transformation zone_noire = {0, {0, 0}};
4   Transformation zone_blanche = {0, {0, 255}};
5
6   LinReg meme_couleur = {1, 0};
7
8   ImageFractale FigureFractale::carre(int n) {
9       /* Une fractale représentant un carré au centre qui se répète dans les coins
10       * Entrée : n : la taille de l'image
11       */
12      n /= 3;
13
14      ImageFractale image = ImageFractale();
15      image.mIfs.push_back({n, 3*n, std::vector<Correspondance>()});
16      image.mMoyenne.push_back(0);
17      image.mLargeur = image.mHauteur = 3*n;
18
```

```
    Correspondance blanc = {0, 0, zone_blanche};
    Correspondance noir = {0, 0, zone_noire};
    Correspondance copie = {0, 0, {0, {1, 0}}};
    Correspondance rot1 = {0, 0, {90, {1, 0}}};
    Correspondance rot2 = {0, 0, {360-90, {1, 0}}};


        image.mIfs[0].correspondances.push_back(copie);
        image.mIfs[0].correspondances.push_back(noir);
        image.mIfs[0].correspondances.push_back(copie);

        image.mIfs[0].correspondances.push_back(noir);
        image.mIfs[0].correspondances.push_back(blanc);
        image.mIfs[0].correspondances.push_back(noir);

        image.mIfs[0].correspondances.push_back(copie);
        image.mIfs[0].correspondances.push_back(noir);
        image.mIfs[0].correspondances.push_back(copie);

    return image;
}

void FigureFractale::generer_exemples(int n) {
    #ifdef OS_WINDOWS
    #else
    int r = system("mkdir -p examples");
    #endif

    FigureFractale::carre(n).enregistrer("examples/carre.ifs");
    FigureFractale::carre(n).exporter("examples/carre.png");
}
```

FigureFractale.h

```
class FigureFractale;

#ifndef FIGURE_FRACTALE
#define FIGURE_FRACTALE

#include "ImageFractale.h"

class FigureFractale {
    /* Pour générer des fichiers d'exemple de fractales */
    public :
        static ImageFractale carre(int dimension);

        static void generer_exemples(int dimension);
};

#endif
```

debug.h

```
/*
 * Des fonctions destinées au débuggage d'informations
 */

#ifndef DEBUG_
#define DEBUG_

#include <iostream>
#include <stdlib.h>
#include <sstream>
#include <string>

#include "ImagePart.h"

#ifdef __MINGW32__
#define EFFACER() system("cls")
#elif __linux__
#define EFFACER() int retour_system = system("clear") ; std::cout << std::endl;
#endif

#define COUT extern bool SILENCIEUX ; if(!SILENCIEUX) std::cout
#define DEBUG extern bool VERBOSE ; if(VERBOSE) std::cout << "(dbg) "

std::string chargement(int actuel, int total, int taille=30);
std::string sourceToString(Correspondance source);
void debugIfs(const IFS&);

#endif
```

debug.cpp

```
#include "debug.h"

std::string chargement(int actuel, int total, int taille) {
    /* Retourne une barre de chargement ascii
     *  - actuel : l'état actuel
     *  - total : l'état correspondant à la fin
     *  - taille : la largeur de la barre
     */
    std::stringstream retour;
    retour << "[";
    for(int i=1 ; i<=taille ; i++) {
        if( int(i*total/taille) <= actuel ) retour << "#";
        else retour << "-";
    }
    retour << "] (" << actuel << "/" << total << ")";
    return retour.str();
}

std::string sourceToString(Correspondance src) {
    std::stringstream retour;
```

```
21      retour << src.bloc << " -> ";
22      retour << "rotation:" << src.transformation.rotation;
23      retour << " decalage:" << src.transformation.droite.a << "X+(" <<
    ↪   src.transformation.droite.b << ")";
24      return retour.str();
25  }
26
27  void debugIfs(const IFS& ifs) {
28      std::vector<int> decoupes(20, 0);
29      for(int i=0 ; i < ifs.correspondances.size() ; i++)
30          decoupes[ ifs.correspondances[i].spliter ] ++;
31      for(int i=0 ; i < decoupes.size() ; i++) {
32          if( decoupes[i] != 0 ) {
33              DEBUG << " - splitage " << i << " : " << decoupes[i] << std::endl;
34          }
35      }
36  }
```