Rapport de Stage

présenté à
M. Marc Parizeau

par
Stéphane Drouin

Eté 1999

Sommaire

L’analyse d’une image débute habituellement par la construction d’un modèle de la scène. Les segments de droite et les points d’intersections sont deux primitives classiques dans l’élaboration du modèle. Ce rapport présente une méthode simple pour obtenir ces primitives. L’algorithme de Canny utilise l’information du gradient pour obtenir les arêtes aux maxima locaux. Les arêtes obtenues sont regroupées en contours avec l’utilisation d’un double seuil, puis sont approximées par des segments de droite. Les points d’intersection sont calculés en utilisant ces segments.

Un ensemble de classes C++ dans lequel ces algorithmes sont implantés est aussi présenté avec quelques résultats obtenus sur des images de test. Les résultats obtenus sur les scènes très simples correspondent parfaitement aux caractéristiques réelles, mais les résultats sur des scènes complexes ou posant des problèmes d’illumination ou de bruit sont moins intéressants.

Finalement, une nouvelle classe C++ pour accéder aux fichier images est présentée. Cet outil est très simple d’utilisation et peut décoder plusieurs types de fichiers sans indication particulière de la part de l’utilisateur.

Table des matières

1.0 Introduction 1

2.0 Contours et points d'intersection 2

2.1 Définition du problème 2

2.2 Algorithmes pour la détection des arêtes 2

2.2.1 Canny 3

2.2.2 Convolution 3

2.2.3 Suppression des non-maxima 5

2.2.4 Double seuil 5

2.3 Algorithmes pour le traitement des contours 6

2.3.1 Segmentation des contours courbés 6

2.3.2 Fusion des segments de droite 7

2.3.3 Identification des points d'intersection 9

2.4 Classes définies 9

2.4.1 Hiérarchie Image 10

2.4.2 Patron de classe Image 10

2.4.3 Classe ImageRGB 10

2.4.4 Classe ImageGradient 10

2.4.5 Classe ImageGris 11

2.4.6 Classe ImageBinaire 11

2.4.7 Classe Contour 11

2.4.8 Classe Point 12

2.4.9 Classe Operateur 12

2.4.10 Intégration avec les classes FVdessin 13

2.5 Etude des résultats obtenus sur des images de test 14

2.5.1 Choix des images de test et quantification des résultats 14

2.5.2 Résultats obtenus 14

2.5.3 Etude des paramètres 16

2.5.4 Forces et faiblesses des algorithmes étudiés 16

2.6 Autres algorithmes 16

2.6.1 Détecteur de Harris 16

2.6.2 Détecteur "Laplacian of Gaussian" 17

2.6.3 Points d'intersections 17

3.0 Lecture et écriture d'images avec ImageStream 18

3.1 Aperçu général 18

3.2 Conception de la classe ImageStream 18

3.2.1 Simplicité d'utilisation 19

3.2.2 Maintenabilité 19

3.2.3 Portabilité 20

3.3 Gestion des erreurs 20

4.0 Conclusion 21

Annexe A Documentation complète des classes développées 23

Annexe B Contours: images de test et résultats 58

Liste des tableaux

Tableau 1 Conversions des classes Image vers les primitives de FVdessin. 13
Tableau 2 Identification des segments et des points d'intersection, paramètres optimaux. 15
Tableau 3 Identification des segments et des points d'intersection par Canny et LoG. 17
Tableau 4 Identification des segments selon l'utilisation des points d'intersection (pi). 17
Tableau 5 Formats des fichiers d'image supportés par ImageStream. 18
Tableau 6 Identification des segments et des intersection, paramètres pour house. 60
Tableau 7 Identification des segments et des intersection, paramètres pour Ombre0_OG4. 60
Tableau 8 Identification des segments et des intersection, paramètres pour Ombre0_OG5. 61
Tableau 9 Identification des segments et des intersection, paramètres pour Paper0_OG0. 61
Tableau 10 Identification des segments et des intersection, paramètres pour rlf1. 62
Tableau 11 Identification des segments et des intersection, paramètres pour sqr1. 62
Tableau 12 Identification des segments et des intersection, paramètres pour wdg4. 63

Liste des figures

Figure 1 Convolution d'une image et d'un opérateur. 3
Figure 2 Image miroir. 4
Figure 3 Fusion de segments: (a) le troisième critère empêche la fusion; (b) fusion réalisée. 7
Figure 4 Traitement des segments unitaires: (a) Segments unitaires; (b) résultat possible sans traitement préalable; (c) regroupement des segments unitaires; (d) résultat attendu. 7
Figure 5 Filtrage des segments: (a) Agencement initial avec un segment plus court que le critère de distance; (b) après fusion; (c) après filtrage. 9
Figure 6 Hiérarchie des classes Image. 10
Figure 7 Opérateurs de Roberts 2x2. 11
Figure 8 Représentation d'un Operateur. 13
Figure 9 Résultats normalisés moyens obtenus avec les paramètres optimaux pour des images soumises à un bruit gaussien dont la variance est entre 0 et 78. 15
Figure 10 Image house, référence et résultat. (CMU/VASC Image Database) 58
Figure 11 Image Ombre0_OG4, référence et résultat. (INRIA-Syntim ©) 58
Figure 12 Image Ombre0_OG5, référence et résultat. (INRIA-Syntim ©) 58
Figure 13 Image Paper0_OG0, référence et résultat. (INRIA-Syntim ©) 59
Figure 14 Image rlf1, référence et résultat. (HIPR Image Library) 59
Figure 15 Image sqr1, référence et résultat. (HIPR Image Library) 59
Figure 16 Image wdg4, référence et résultat. (HIPR Image Library) 59
Figure 17 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image house soumise à un bruit gaussien dont la variance est entre 0 et 78. 64
Figure 18 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image Ombre0_OG4 soumise à un bruit gaussien dont la variance est entre 0 et 78. 64
Figure 19 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image Ombre0_OG5 soumise à un bruit gaussien dont la variance est entre 0 et 78. 65
Figure 20 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image Paper0_OG0 soumise à un bruit gaussien dont la variance est entre 0 et 78. 65
Figure 21 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image rlf1 soumise à un bruit gaussien dont la variance est entre 0 et 78. 66
Figure 22 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image sqr1 soumise à un bruit gaussien dont la variance est entre 0 et 78. 66
Figure 23 Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image wdg4 soumise à un bruit gaussien dont la variance est entre 0 et 78. 67
Figure 24 Image des 6 objets. Paramètres: f=5, g=3, t=17, q=19, i=9, v=5, m=0.01. 68
Figure 25 Image 02_Mtl_G. Paramètres: f=9, g=3, t=35, q=10, i=0, v=10, m=0.1. 68
Figure 26 Image 09_Bur_1. Paramètres: f=5, g=3, t=55, q=3, i=0, v=5, m=0.1. 69
Figure 27 Image 12_Numeros. Paramètres: f=5, g=3, t=55, q=3, i=0, v=5, m=0.1 69
Figure 28 Image stéréo du corridor. Paramètres: f=0, g=2, t=80, q=15, i=4, v=2, m=0.01 69

Liste des symboles

$f(i,j)$	Image
$h(i,j)$	Image résultant d'une convolution entre un opérateur et une image
$p(i,j)$	Pixel d'une image
$g(i,j)$	Opérateur
(o_x, o_y)	Origine d'un opérateur
n_{image}	Nombre de pixels sur une image (n_{image} = lignes x colonnes)
$n_{opérateur}$	Nombre de coefficients d'un opérateur
$n_{segments}$	Nombre de segments de droite dans une image
$O()$	Complexité d'un algorithme, c'est-à-dire une limite supérieure à la relation entre la taille du probème traité et la durée du traitement.

1.0 Introduction

La manipulation d'images est une tâche courante au Laboratoire de Vision et Systèmes Numériques (LVSN). D'une part, il faut accéder à des images archivées sur disque et d'autre part, on doit leur appliquer des traitements élémentaires. Dans l'élaboration de systèmes plus complexes, ces deux tâches devraient apparaître comme des composantes de base, des acquis. C'est avec cette idée en tête qu'ont été créés deux ensembles d'outils d'usage général codés en C++. Ces outils se présentent comme des classes et hiérarchies de classes encapsulant des traitements classiques sur les images.

La hiérarchie de classes `Image` sert à représenter des images en couleur ou niveaux de gris. Chaque type d'image possède des traitements particuliers qui sont implantés dans les classes pertinentes. La recherche de contours et de points caractéristiques dans les images en niveaux de gris est le premier problème traité. Cette étape est souvent essentielle dans la construction d'un modèle du monde réel à partir d'une image. La panoplie de traitements élémentaires associés à cette tâche inclut la détection d'arêtes et le suivi de contour. Les algorithmes propres à la recherche de contours et de points d'intersection sont tout d'abord présentés, de même que les points importants de leur implantation et les classes impliquées. En particulier, l'algorithme de Canny et son implantation dans une classe représentant une image en niveaux de gris sont présentés avec les résultats obtenus sur des images de test. Les limites des algorithmes implantés sont discutées et des pistes de solution sont proposées. Finalement, certains des autres algorithmes implantés et quelques uns qu'il serait intéressant d'intégrer à la boîte à outils sont brièvement présentés.

Dans un deuxième temps, une classe en C++ pour la lecture et l'écriture de fichiers image sur le disque est présentée. La classe `ImageStream` est une spécialisation de la classe standard `fstream` faisant le lien, par les opérateur d'insertion et d'extraction, entre les objets image en mémoire et l'archivage sur disque. Les choix de conception et les détails d'utilisation de la classe `ImageStream` sont discutés. En particulier, sa simplicité d'utilisation, sa maintenabilité et les formats de fichiers supportés sont les points importants qui sont exposés.

2.0 Contours et points d'intersection

2.1 Définition du problème

Les images obtenues par les systèmes de vision ne peuvent pas être directement analysées. La séquence de traitements implique en premier lieu de retrouver l'information pertinente contenue dans l'image. Dans le cas présent, les informations recherchées sont les segments de droite contenus dans l'image et les points d'intersection entre ces segments. L'objectif des classes développées est donc le suivant: étant donné une image en entrée, produire la liste des contours et la liste des points caractéristiques la composant.

La première étape du traitement consiste à identifier, sur une image en niveaux de gris, les arêtes de la scène. En d'autres mots, on doit trouver, parmi tous les pixels de l'image, ceux qui sont situés sur la bordure d'une surface uniforme. Cette étape est fondamentale puisque tous les traitements suivants sont faits sur ces arêtes.

La deuxième étape consiste à rassembler les pixels d'arête selon leur appartenance à un segment continu. Puisqu'on s'intéresse aux segments de droite, on doit pouvoir décomposer les segments trop courbés en des segments plus courts, mais respectant la contrainte fixée. On désire aussi avoir les segments les plus longs possibles. Les segments contigus doivent être aboutés si possible.

Finalement, les points d'intersection entre les segments de droite sont recherchés. La représentation finale est un graphe où les sommets sont les points d'intersection et les arêtes sont les segments de droite.

2.2 Algorithmes pour la détection des arêtes

La détection des arêtes consiste essentiellement à repérer les changements locaux d'intensité. La dérivée, ou le gradient dans le cas des fonctions à plusieurs dimensions, est tout à fait désignée dans un tel cas. Puisqu'une image est une fonction discrète, des méthodes de différences finies sont utilisées pour le calcul du gradient. Les algorithmes de détection d'arête sont constitués de trois étapes:

- *Filtrage*: La dérivée est très sensible au bruit; une image doit donc être filtrée afin de ne pas générer trop de fausses arêtes. Cependant, le filtrage diminue aussi la réponse de l'algorithme pour les arêtes réelles. Il y a donc un compromis entre la diminution du bruit et la conservation des arêtes.
- *Mise en valeur*: Il s'agit de faire ressortir les arêtes. C'est à cette étape que le gradient est calculé.
- *Détection*: L'algorithme doit pouvoir déterminer, pour chaque pixel, si le gradient y est suffisamment grand pour être sur une arête.

2.2.1 Canny

L'algorithme choisi pour détecter les arêtes est l'algorithme de Canny. Le détecteur de Canny est la dérivée d'une gaussienne. L'algorithme s'énonce comme suit:

Algorithme 1 Détection d'arêtes avec l'opérateur de Canny. [8, p.172]

1. Lisser l'image à l'aide d'un filtre Gaussien.
2. Calculer le module et l'orientation du gradient pour chaque pixel en utilisant une approximation par différences finies pour les dérivées partielles.
3. Faire la suppression des non-maxima pour le module du gradient.
4. Utiliser l'algorithme de double seuil pour détecter et lier les arêtes.

La complexité algorithmique de Canny dépend de la complexité des algorithmes utilisés pour réaliser chacune des parties. Les étapes 1 et 2 sont réalisées par convolution (voir ci-dessous pour la discussion) de complexité $O(n_{opérateur}n_{image})$. L'étape 3 utilise un algorithme de suppression des non-maxima de complexité $O(n_{image})$ et l'étape 4 utilise un algorithme de double seuil aussi de complexité $O(n_{image})$.

La complexité de l'algorithme de Canny est $O(n_{opérateur}n_{image})$, $n_{opérateur}$ faisant référence à l'opérateur de plus grande taille utilisé dans l'algorithme (filtrage ou dérivation).

2.2.2 Convolution

Puisque les parties 1 et 2 de Canny impliquent des opérations linéaires, on peut les réaliser par la convolution de l'image d'intérêt avec un opérateur approprié. Soit f(i,j) l'image originale, g(i,j) l'opérateur de convolution de dimension *n x m* et dont l'origine est (o_x, o_y). Le résultat de la convolution, est [8, p.116]:

$$h(i,j) = f(i,j) \times g(i,j) = \sum_{k=1}^{n}\sum_{l=1}^{m} f(i+k-o_x, j+l-o_y)g(k,l) \qquad \text{(EQ 1)}$$

Par exemple pour l'image et l'opérateur de la figure suivante, on aurait:

$$h(4,3) = f(4,3)\,g(1,1) + f(4,4)\,g(1,2) + f(4,5)\,g(1,3) + f(5,3)\,g(2,1) + f(5,4)\,g(2,2) + f(5,5)\,g(2,3) \qquad \text{(EQ 2)}$$

f(1,1)	f(1,2)	f(1,3)	f(1,4)	f(1,5)
f(2,1)	f(2,2)	f(2,3)	f(2,4)	f(2,5)
f(3,1)	f(3,2)	f(3,3)	f(3,4)	f(3,5)
f(4,1)	f(4,2)	f(4,3)	f(4,4)	f(4,5)
f(5,1)	f(5,2)	f(5,3)	f(5,4)	f(5,5)
f(6,1)	f(6,2)	f(6,3)	f(6,4)	f(6,5)

g(1,1)	g(1,2)	g(1,3)
g(2,1)	g(2,2)	g(2,3)

FIGURE 1. Convolution d'une image et d'un opérateur.

Dans cet exemple, l'origine de l'opérateur, c'est à dire la coordonnée du coefficient de l'opérateur qui multiplie le pixel d'intérêt est le point (1,1). En général, on admet que l'origine peut être située n'importe où sur l'opérateur. Il est bon de noter que changer l'origine de l'opérateur est équivalent à faire une translation des pixels de l'image résultante.

Dans le cas discret de la convolution, les bordures de l'image posent un problème si on veut multiplier un coefficient d'opérateur avec un pixel qui n'existe pas a l'extérieur de l'image. Une solution possible est d'ignorer les pixels de l'image où cette condition est rencontrée et de produire une image résultante plus petite que l'image d'entrée. Dans notre exemple, on ne calculerait pas les résultats h(6,*), h(*,4) et h(*,5). L'image résultante serait de dimension 5x3 pour une image originale 6x5.

Puisque l'algorithme de Canny implique plusieurs convolutions avec des opérateurs de dimension potentiellement non négligeable, on souhaite produire un résultat pour tous les pixels de l'image. Il faut alors ajouter arbitrairement des pixels autour de l'image d'entrée afin de produire un résultat de même dimension que l'entrée. Une solution simple est de fixer ces nouveaux pixels à une valeur nulle. Cependant, une importante distorsion apparaît alors pour les pixels de bordure [4]. La solution choisie est de créer un effet 'miroir' sur les bordures. D'abord avec les bordures horizontales (haut et bas) puis dans un deuxième temps avec les bordures verticales (gauche et droite). Dans ce cas, les valeurs produites pour les pixels de bordure ne présentent pas de distorsion notable. Dans notre exemple, le résultat de l'ajout des bordures prendrait la forme suivante:

f(1,1)	f(1,2)	f(1,3)	f(1,4)	f(1,5)	f(1,4)	f(1,3)
f(2,1)	f(2,2)	f(2,3)	f(2,4)	f(2,5)	f(2,4)	f(2,3)
f(3,1)	f(3,2)	f(3,3)	f(3,4)	f(3,5)	f(3,4)	f(3,3)
f(4,1)	f(4,2)	f(4,3)	f(4,4)	f(4,5)	f(4,4)	f(4,3)
f(5,1)	f(5,2)	f(5,3)	f(5,4)	f(5,5)	f(5,4)	f(5,3)
f(6,1)	f(6,2)	f(6,3)	f(6,4)	f(6,5)	f(6,4)	f(6,3)
f(5,1)	f(5,2)	f(5,3)	f(5,4)	f(5,5)	f(5,4)	f(5,3)

FIGURE 2. Image miroir.

Algorithme 2 Convolution.

1. Produire une image temporaire contenant l'image originale et ajouter les bordures `miroir'.

2. Pour chaque pixel de l'image originale:

2.1. Placer l'origine de l'opérateur au pixel traité.

2.2. Calculer la somme des multiplications des coefficients de l'opérateur avec les pixels correspondants de l'image temporaire.

2.3. Ecrire le résultat dans le pixel traité après s'être assuré que les valeurs limites acceptées par l'image sont respectées.

Cet algorithme calcule, pour chacun des n_{image} pixels de l'image, $n_{opérateur}$ multiplications et ($n_{opérateur}$-*1*) additions. L'algorithme est donc de complexité $O(n_{opérateur}n_{image})$.

2.2.3 Suppression des non-maxima

La partie 3 de Canny implique l'utilisation d'un algorithme de suppression des non-maxima. On définit une arête comme l'endroit sur une image où le changement d'intensité est un maximum local, c'est à dire où le module du gradient est un maximum local. On obtient alors des arêtes de largeur unitaire. L'idée est de comparer le module du gradient en chaque point avec le module du gradient des pixels voisins. Encore une fois, les pixels situés sur les bords de l'image posent un problème. Dans ce cas, on ignore les pixels des lignes et colonnes formant la bordure de l'image. On décide donc arbitrairement qu'ils ne sont pas des maximum locaux. D'une part, l'opération de convolution ayant produit les résultats sur la bordure a potentiellement crée une distorsion pour ces pixels. D'autre part, on peut supposer que les arêtes d'intérêt se situent plutôt au centre de l'image et que l'information contenue sur les extrémités, de par sa nature incomplète, est négligeable.

Algorithme 3 Suppression des non-maxima. [8, p.170]

1. Fixer tous les pixels de bordure de l'image résultante à 0.

2. Pour tous les autres pixels p(i,j):

2.1. Déterminer quels sont les deux pixels voisins de part et d'autre de p(i,j), dans la direction du gradient à p(i,j) (horizontal, vertical, diagonal; voir [8, p.171] pour une représentation de la partition des orientations).

2.2. Si le module du gradient à p(i,j) n'est pas plus grand que le module du gradient aux deux pixels voisins déterminés précédemment, le pixel n'est pas à un maxima et il faut le fixer à 0.

Cet algorithme applique un nombre constant d'opérations sur tous les pixels de l'image traitée. Il est donc de complexité $O(n_{image})$.

2.2.4 Double seuil

La partie 4 de Canny implique l'utilisation d'un algorithme de double seuil. La méthode typique pour détecter une arête est de fixer un seuil unique. Si le module du gradient en un point est supérieur à ce seuil, l'arête est acceptée, sinon elle est supprimée. Cette méthode peut, d'une part produire des fausses arêtes parce que le seuil est trop bas (faux positifs) ou d'autre part produire des arêtes manquantes parce que le seuil est trop haut (faux négatifs).

Une méthode plus efficace consiste à utiliser deux seuils. Le seuil le plus élevé permet de sélectionner les arêtes correctement en évitant les faux positifs. Le seuil le plus bas permet de compléter les arêtes sélectionnées où le gradient est plus faible à cause d'une illumination non uniforme et corrige les faux négatifs introduits par le seuil élevé.

Algorithme 4 Double seuil pour détecter et lier les arêtes. [8, p.172]

1. Choisir deux seuils, T_1 et T_2 avec $T_2 > T_1$. ($T_2 \approx 2T_1$).

2. Produire une image avec le seuil T_1 et une autre image avec le seuil T_2.

3. Produire des contours avec les pixels actifs de T_1 et T_2. Rassembler tous les pixels voisins de l'image produite avec T_2 en contours. Si un contour se termine sur cette image, ajouter les pixels voisins produits avec le seuil T_1 jusqu'à ce que le contour se termine sur l'image produite avec T_1 ou qu'il rejoigne un pixel actif sur l'image produite avec T_2.

L'algorithme crée deux images binaires, cette étape étant de complexité $O(n_{image})$ et une étape de comparaison des pixels qui est de complexité $O(n_{image})$ (chaque pixel est comparé un maximum de huit fois avec un de ses voisins). La complexité totale de l'algorithme est donc $O(n_{image})$.

Le résultat de l'algorithme de double seuil est le résultat final de l'algorithme de Canny. Les contours produits ne sont pas nécessairement des segments de droite. Aussi, des arêtes qui sont continues dans la réalité peuvent être séparées en plusieurs segments lors des traitements précédemment décrits. C'est pourquoi on a besoin d'algorithmes pour traiter les contours obtenus.

2.3 Algorithmes pour le traitement des contours

2.3.1 Segmentation des contours courbés

Un contour est une liste de *n* points. Pour obtenir les meilleurs segments de droite possibles, on doit être en mesure de quantifier l'erreur entre les points réels du segment et la droite passant par les extrémités. Une mesure adéquate est l'erreur normalisée maximale, telle que définie dans [8, p.193]:

$$\varepsilon = \frac{\max_i |d_i|}{D} \qquad \text{(EQ 3)}$$

où d_i est la distance entre le point *i* et la courbe et *D* est la longueur de la droite:

$$d_i = \frac{r_i}{D} = \frac{x_i(y_1 - y_2) + y_i(x_2 - x_1) + y_2x_1 - y_1x_2}{D} \qquad \text{(EQ 4)}$$

$$D = \sqrt{(y_n - y_1)^2 + (x_n - x_1)^2} \qquad \text{(EQ 5)}$$

Si l'erreur sur un segment est trop grande, on le sépare en deux segments, à l'endroit où l'erreur est maximale. La distance d_i suppose que la projection du point *i* sur la droite est située sur le segment de droite. Dans le cas contraire, d_i est calculé comme la distance minimale entre le point *i* et un point d'extrémité du segment.

Algorithme 5 Séparation d'un segment courbé en segments de droite. [8, p.196]

1. Trouver le point du segment le plus éloigné de la droite approximée.

2. Si l'erreur normalisée en ce point est trop grande, séparer la liste des points en deux segments à cet endroit, sinon le segment est accepté.

3. Reprendre l'algorithme pour les deux nouveaux segments s'il y a eu séparation.

Si on suppose que l'algorithme s'applique un nombre négligeable de fois sur chaque segment avant d'atteindre le cas de base, sa complexité est $O(n_{segments})$.

2.3.2 Fusion des segments de droite

Un problème rencontré par les algorithmes de détection d'arêtes est la présence de trous entre des pixels faisant partie d'un même contour. Ces trous s'expliquent par des contrastes d'éclairage ou par le choix d'un seuil trop haut dans la phase de détection. On doit donc être en mesure de fusionner des segments après avoir déterminé leur appartenance à un seul et même contour. Cette appartenance est caractérisée par la combinaison de deux critères, soit une orientation semblable et une faible distance entre deux de leurs extrémités. Un troisième critère est utilisé pour s'assurer que le segment produit est bien l'addition des deux parties, c'est à dire qu'il a une longueur supérieure à chacune des deux parties.

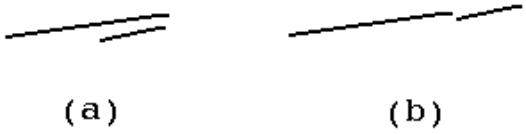

FIGURE 3. Fusion de segments: (a) le troisième critère empêche la fusion; (b) fusion réalisée.

Les segments composés d'un seul point constituent un cas particulier car ils ne possèdent pas d'orientation définie. Ils n'ont donc pas à respecter le critère d'orientation. Cependant, une série de points alignés devraient produire un segment unique. Pour cette raison et aussi pour éviter l'interférence avec les segments plus longs, on regroupe préalablement les segments unitaires en paires pour leur donner une orientation.

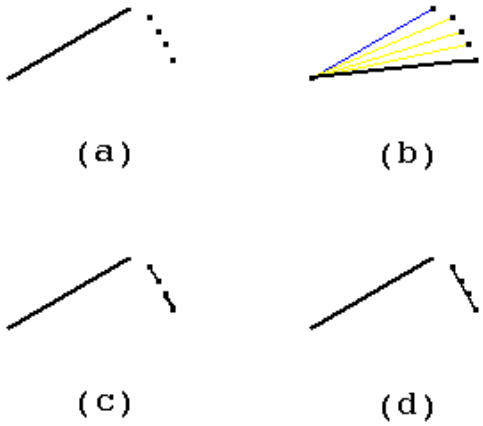

FIGURE 4. Traitement des segments unitaires: (a) Segments unitaires; (b) résultat possible sans traitement préalable; (c) regroupement des segments unitaires; (d) résultat attendu.

Algorithme 6 Regroupement par paires des segments unitaires.

1. Pour chaque segment unitaire S_i dans la liste des contours:

1.1. Trouver dans le reste de la liste le segment unitaire S_j le plus près de S_i et respectant la contrainte de distance.

1.2. Si un tel segment existe, fusionner S_i et S_j pour former un segment de deux points.

Dans le pire des cas, cet algorithme s'applique $\binom{n_{segments}}{2} = \frac{n_{segments}!}{2\,(n_{segments}-2)!}$ fois, c'est à dire pour chaque combinaison de segments. La complexité algorithmique est donc $O(n_{segments}^2)$.

Algorithme 7 Fusion de segments.

1. Regrouper en paire les segments unitaires.

2. Pour chaque segment S_i dans la liste des contours, parcourir tous les segments S_j suivants dans la liste:

2.1. Calculer l'orientation de S_j. Si le critère d'orientation n'est pas respecté, passer au segment suivant.

2.2. Calculer la distance minimale entre les extrémités de S_i et S_j correspondant au meilleur agencement ($fin_i \rightarrow origine_j$, $fin_i \rightarrow fin_j$, $origine_i \rightarrow origine_j$, $origine_i \rightarrow fin_j$). Si le critère de distance n'est pas respecté, passer au segment suivant.

2.3. Calculer la distance entre les extrémités les plus éloignées. Si cette distance est plus petite que la longueur de S_i ou de S_j, passer au segment suivant.

2.4. Fusionner les listes de points de S_i et S_j en respectant l'ordre impliqué par le meilleur agencement trouvé en 2.2.

Dans notre implantation, nous répétons l'étape 2 en incrémentant le critère de distance de 0 à la valeur maximale. On obtient ainsi de meilleures fusions quand une portion de l'image traitée comporte plusieurs segments adjacents. Comme l'algorithme de regroupement des segments unitaires, la complexité algorithmique est $O(n_{segments}^2)$.

Avec les deux algorithmes précédents et une configuration particulière de la liste des segments, il est possible de fusionner deux segments positionnés de part et d'autre d'un troisième. Normalement ce troisième segment, plus court que le critère de distance, devrait aussi être fusionné, mais il a été ignoré. Pour solutionner ce problème, on peut appliquer un filtrage sur les segments pour retirer ceux dont la longueur est inférieure au critère de distance.

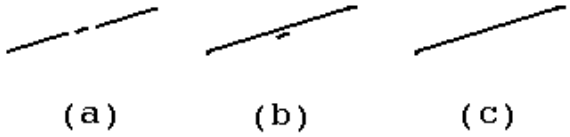

FIGURE 5. Filtrage des segments: (a) Agencement initial avec un segment plus court que le critère de distance; (b) après fusion; (c) après filtrage.

2.3.3 Identification des points d'intersection

Une méthode pour estimer les points d'intersection consiste à utiliser les équations des droites qui estiment les segments [8, p.214]. On réduit ainsi l'erreur causée par la détection des arêtes qui a une mauvaise réponse aux intersections dans l'image originale. On peut espérer une précision inférieure au pixel si l'image n'est pas affectée par du bruit et que les segments sont bien identifiés. Soient les droites D_1 et D_2, et leurs équations implicites:

$$a_1x + b_1y + c_1 = 0 \quad \text{(EQ 6)}$$

$$a_2x + b_2y + c_2 = 0 \quad \text{(EQ 7)}$$

Le point d'intersection entre les deux droites est:

$$x = \frac{c_1a_2 - c_2a_1}{a_1b_2 - a_2b_1}, y = \frac{c_2b_1 - c_1b_2}{a_1b_2 - a_2b_1} \quad \text{(EQ 8)}$$

Si a_1b_2-$a_2b_1 = 0$, les droites sont parallèles et il n'y a pas de point d'intersection unique. Autrement, si le point identifié est suffisamment près de D_1 et de D_2, le point d'intersection est accepté.

Algorithme 8 Identification des points d'intersection.

1. Pour chaque paire non ordonnée de segments (S_i, S_j), calculer le point d'intersection (x, y) entre les droites passant par les extrémités de chaque segment.

1.1. Si le point (x, y) existe et est unique et est suffisamment près de S_i et de S_j, l'identifier comme point d'intersection.

1.2. Sinon, les deux segments ne s'interceptent pas.

Cet algorithme s'applique pour chaque combinaison de segments. La complexité algorithmique est $O(n_{segments}^2)$.

2.4 Classes définies

La résolution du problème de détection de contours et d'identification des points caractéristiques est implantée dans un ensemble de classes représentant les données à traiter. La

détection des arêtes sur une image en niveaux de gris est faite dans la classe `ImageGris` et produit un objet de la classe `Contour`. Les traitements sur les contours et les points d'intersections sont faits dans cette dernière classe. Cette section présente les méthodes principales des classes développées. Une description plus détaillée des fonctions est donnée en annexe.

2.4.1 Hiérarchie `Image`

Les algorithmes s'appliquant sur les images sont implantés dans la hiérarchie Image. Cette hiérarchie prend la forme suivante:

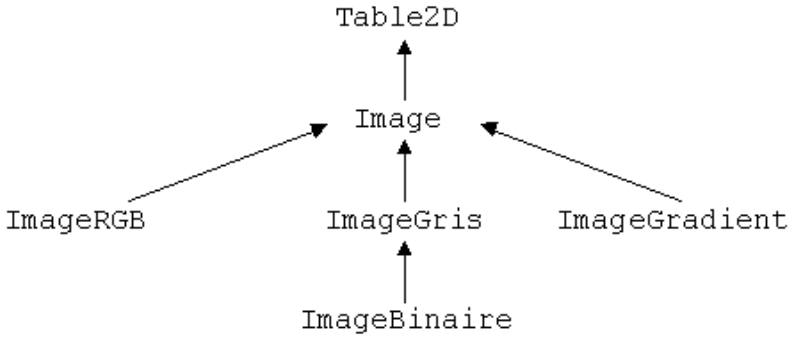

FIGURE 6. Hiérarchie des classes `Image`.

2.4.2 Patron de classe `Image`

Le patron de classe `Image` définit les traitements généraux comme la convolution avec un objet de la classe `Operateur`, les algorithmes de filtrage et les opérations mathématiques et logiques entre images et entre une image et une constante. `Image` est un patron de classe et ne spécifie donc pas le type des pixels.

2.4.3 Classe `ImageRGB`

La classe `ImageRGB` définit une représentation des images en couleurs. Le type de données utilisé est le `unsigned long`. Les classes `CouleurRGB` et `CouleurHSI` peuvent être utilisées pour accéder aux composantes rouge, vert, bleu et hue, saturation, intensité respectivement. Ces classes indépendantes sont construites de telle façon qu'il est possible de leur assigner une valeur de type `unsigned long` et d'assigner leur valeur à une variable de type `unsigned long`. La conversion d'une représentation à l'autre est également prévue. L'utilisateur peut manipuler les couleurs dans la représentation désirée sans se préoccuper du type effectif de la classe `ImageRGB`.

2.4.4 Classe `ImageGradient`

La classe `ImageGradient` définit les fonctions nécessaires pour obtenir le gradient (module et direction) d'une image en niveaux de gris. Les pixels d'une `ImageGradient` représentent le module du gradient et sont de type `int`. Un objet appartenant à la classe `ImageGradient` possède en plus une variable membre de type `Image<int>` nommée

`direction`. Cette image, de la même dimension que l'objet d'intérêt contient la direction associée au gradient pour chaque pixel, exprimée en degrés et variant entre 0 et 360. Cette séparation du module et de l'orientation a été introduite afin de faciliter l'accès au module, qui est le plus déterminant dans l'identification des arêtes et dans le but de conserver la représentation la plus simple possible.

2.4.5 Classe `ImageGris`

La classe `ImageGris` définit une représentation des images en niveaux de gris. Les pixels sont de type `unsigned char`. On peut assigner une `ImageRGB` à une `ImageGris` qui prend alors la valeur moyenne des composantes de couleur, soit l'intensité.

La méthode d'intérêt pour résoudre le problème des contours est `Canny`. Cette fonction prend en entrée les paramètres de l'algorithme de Canny [Algorithme 1], soit la taille du filtre gaussien, la taille de l'opérateur de gradient à utiliser et un seuil pour le gradient. Elle retourne la liste des segments de droite de l'image dans un objet de la classe `Contour`. L'utilisateur peut spécifier les opérateurs à utiliser pour le calcul du gradient horizontal et vertical. Par défaut, l'opérateur pour le calcul du gradient est celui de Roberts [6, p.200]:

dx = | 1 | 0 |
| 0 | -1 |

dy = | 0 | 1 |
| -1 | 0 |

FIGURE 7. Opérateurs de Roberts 2x2.

Dans notre implantation, l'opérateur de Roberts peut être défini pour différentes tailles. Un opérateur de taille *n* est un tableau *n* x *n* d'origine *((n-1)/2, (n-1)/2)* dont les cases des quatre coins ont les coefficients de l'image précédente et les autres cases ont des coefficients nuls. Un cas particulier de convolution d'une `Image` avec un opérateur de Roberts est prévu. En effet, puisque seulement deux coefficients sont non-nuls pour un tel opérateur, le nombre de multiplications à faire est réduit et la complexité de la convolution passe de $O(n_{opérateur} n_{image})$ à $O(n_{image})$.

2.4.6 Classe `ImageBinaire`

La classe `ImageBinaire` est un cas particulier d'une `ImageGris` où les pixels ne peuvent prendre que deux valeurs (`ON`, `OFF`). Dans notre cas, un pixel allumé prend la valeur d'intensité `255` et un pixel éteint prend la valeur `0`. À l'affichage, les pixels `ON` sont donc blancs sur fond noir. La classe possède des méthodes pour transformer une image en niveaux de gris en image binaire. De plus, elle peut se faire des opérations morphologiques d'érosion et de dilatation avec un élément structurant.

2.4.7 Classe `Contour`

Les algorithmes de suivi de contours et de traitement sur les segments sont placés dans la classe `Contour`. Un objet de cette classe contient tous les segments composant une image. La classe `Contour` est une liste de `ContourSegment`. Elle possède les fonctions nécessaires pour rassembler les pixels sur deux `ImageBinaire` tel que décrit dans

l'algorithme de double seuil [Algorithme 4]. Elle possède également les fonctions `SegmenterCourbes` et `Abouter` pour séparer les segments courbés en portions de droite et pour fusionner les segments de droite, respectivement. Les fonctions `Filtrer` et `TrouverIntersections` complètent l'implantation.

La méthode d'intérêt pour résoudre le problème des contours et points d'intersection est `SegmentsDroite`. Cette fonction prend en entrée les critères d'orientation et de distance de l'algorithme de fusion des segments [Algorithme 7], le critère de distance de l'algorithme de détection des intersections [Algorithme 8] et le critère d'erreur maximale de l'algorithme de séparation des segments courbés [Algorithme 5]. L'objet applique alors les algorithmes décrits sur la liste de ses segments et de ses intersections.

La classe `ContourSegment` est une liste de `Points` contigus sur une image. Elle possède la fonction `DroiteMaxErreur` pour retourner l'erreur normalisée maximale et la fonction `Separer` pour se couper en un point spécifique. En outre, elle donne accès à ses extrémités par les fonctions `Origine` et `Fin` et elle peut caractériser le segment de droite approximée avec `Orientation`, `LongueurDroite` et `EquationDroite`.

2.4.8 Classe `Point`

La classe `Point` représente une coordonnée dans un plan cartésien (deux variables `int`) ou la coordonnée d'un pixel dans une image. Le design est centré sur cette dernière application et les variables membres sont nommées `ligne` et `col`. La classe `Point` définit les opérateur relationnels et trois mesures de distance: `Chessboard(Point&)`, `CityBlock(Point&)` et `Euclidean(Point&)`. Ces mesures sont définies dans [8, p.52]:

$$d_{Euclidean}([i_1,j_1],[i_2,j_2]) = \sqrt{(i_1-i_2)^2+(j_1-j_2)^2} \quad \text{(EQ 9)}$$

$$d_{city}([i_1,j_1],[i_2,j_2]) = |i_1-i_2|+|j_1-j_2| \quad \text{(EQ 10)}$$

$$d_{chess}([i_1,j_1],[i_2,j_2]) = \max(|i_1-i_2|,|j_1-j_2|) \quad \text{(EQ 11)}$$

2.4.9 Classe `Operateur`

La classe `Operateur` représente un tableau de coefficients (`int`) utilisé pour faire une convolution avec une `Image`. En plus de ses coefficients, un `Operateur` possède un `Point` d'origine, le point où est appliqué l'`Operateur` sur l'`Image` et un facteur de division appliqué au résultat de la somme des multiplications coefficient-pixel.

Les variables membres `xg`, `xd`, `yh`, `yb` donnent accès direct au nombre de colonnes à gauche et à droite de l'origine et au nombre de lignes au-dessus et au-dessous de l'origine, respectivement, comme indiqué dans la figure suivante.

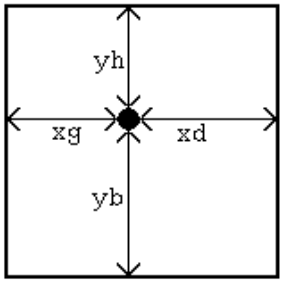

FIGURE 8. Représentation d'un `Operateur`.

Il existe plusieurs arrangements de coefficients qu'on appelle types d'opérateurs. Les opérateurs de Sobel, Roberts et Prewitt servent à calculer la dérivée selon l'axe horizontal ou vertical d'une image. Les opérateurs de forme gaussienne et moyenne sont utilisés principalement comme filtres. Un objet de la classe `Operateur` peut être complètement spécifié par l'utilisateur ou il peut être d'un des types prédéfinis mentionnés précédemment. L'opérateur gaussien a la propriété intéressante d'être séparable en composantes verticale et horizontale. On peut obtenir un opérateur gaussien de taille 5x5 en appliquant séparément un opérateur vertical 5x1 et un opérateur horizontal 1x5. La complexité de l'opération passe donc de $O(n^2)$ à $O(n)$ si on considère n comme étant la largeur du filtre.

2.4.10 Intégration avec les classes `FVdessin`

Afin de tirer partie au maximum du code déjà existant, on a porté une attention particulière à intégrer les classes de visualisation `FVdessin` dans le design de la hiérarchie `Image` et des classes utilitaires associées. En général, l'utilisateur peut directement insérer les objets appartenant aux classes développées dans un `FVdessin`.

Les classes développées contiennent les opérateurs de conversion leur permettant d'être insérées dans un `FVdessin` en tant que primitives graphiques correspondantes. Ainsi, un objet appartenant à la classe `ImageGris` peut être inséré en tant que `FVimageGris`. De même, un objet `Point` peut être inséré en tant que `FVptcour`. Le tableau suivant présente toutes les conversions possibles.

TABLEAU 1. Conversions des classes `Image` vers les primitives de `FVdessin`.

Classe originale	Primitive insérée
`ImageGris`	`FVimageGris`
`ImageGradient`	`FVimageGris`
`ImageRGB`	`FVimageRGB`
`Point`	`FVptcour`
`ContourSegment`	`FVligne`

De plus, la classe `Contour` possède la fonction `Afficher(...)` qui permet à un objet contour de s'afficher dans un dessin (segments et points d'intersections).

2.5 Etude des résultats obtenus sur des images de test

2.5.1 Choix des images de test et quantification des résultats

Les images choisies sont de nature et de difficulté variables. Toutes les images et les résultats obtenus sont présentées en annexe. Les résultats pour chaque image sont quantifiés par la comparaison à un ensemble de résultats de référence établis par un être humain. Les points d'intersection et les segments de droite sont comparés séparément. Pour chaque élément de la liste *référence*, l'élément de la liste *résultat* se situant le plus près est déterminé. Si la distance séparant les deux éléments est à l'intérieur d'une certaine tolérance, ils sont appariés. Le résultat de la comparaison de chaque liste est calculé comme suit:

$$r = \frac{2 \times N_{Paires}}{N_{Reference} + N_{Resultat}} \qquad \text{(EQ 12)}$$

La note attribuée tient donc compte des faux positifs (dénominateur augmenté) et des faux négatifs (numérateur diminué). Puisque les points d'intersections sont calculés à partir des segments de droite, la note totale donne plus d'importance aux résultats associés à la comparaison des segments de droite:

$$r_{Tot} = 0.8 \times r_{Segments} + 0.2 \times r_{Points} \qquad \text{(EQ 13)}$$

La mesure de distance utilisée pour les points d'intersection est la distance euclidienne. Pour les segments de droite, on utilise le calcul suivant: soient les segments S et T dont les extrémités sont les points s_1 et s_2, t_1 et t_2 respectivement. On a les distances euclidiennes entre les points d'extrémités $d_1(s_1, t_1)$, $d_2(s_2, t_2)$, $d_3(s_1, t_2)$ et $d_4(s_2, t_1)$ et la différence entre l'orientation des deux segments $\Delta\theta$, évalué en degrés (0 à 90). La distance entre les deux segments est la suivante:

$$D(S, T) = \frac{\min(d_1 + d_2, d_3 + d_4)}{\frac{90 - \Delta\theta}{90} + 1} \qquad \text{(EQ 14)}$$

2.5.2 Résultats obtenus

Les paramètres donnant les meilleurs résultats pour chaque image sont recherchés par un algorithme génétique. Une population initiale de paramètres est évaluée par la méthode présentée précédemment et une nouvelle génération est engendrée. Après un nombre fixé d'itérations, l'individu ayant donné les meilleurs résultats pour chaque image est choisi pour faire les comparaisons présentées dans la suite de ce rapport. La tolérance est fixée à 2 pour l'appariement des points et à 3 pour les segments.

Le tableau qui suit contient les meilleurs résultats obtenus pour chacune des images de test et les paramètres correspondants. Une série de résultats pour différents paramètres est présentée en annexe.

TABLEAU 2. Identification des segments et des points d'intersection, paramètres optimaux.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
house	0	3	23	7	10	6	0.07	59%	89%	65%
Ombre0_OG4	5	3	40	24	10	13	0.5	55%	33%	50%
Ombre0_OG5	9	3	36	12	5	10	0.1	61%	44%	58%
Paper0_OG0	5	3	35	34	10	9	0.35	84%	77%	83%
rlf1	0	2	50	10	10	5	0.1	100%	100%	100%
sqr1	0	2	50	10	10	5	0.1	100%	100%	100%
wdg4	5	4	50	10	10	10	0.2	100%	100%	100%

Paramètres pour Canny: **f**: taille du filtre gaussien; **g**: taille de l'opérateur pour le calcul du gradient; **t**: seuil pour le gradient.

Paramètres pour les contours: **v**: distance maximale entre les segments à fusionner; θ: angle minimal pour intersections/angle maximal pour fusion; **i**: distance maximale entre point d'intersection et chaque segment intercepté; **m**: erreur normalisée maximale sur les segments de droite.

Résultats: $\mathbf{r_{Segments}}$: pour les segments; $\mathbf{r_{Points}}$: pour les points d'intersection; $\mathbf{r_{Tot}}$: résultat final.

Les meilleurs paramètres sont ensuite utilisés pour traiter des images soumises à un bruit gaussien. Les résultats ainsi obtenus sont comparés aux résultats obtenus précédemment pour quantifier la dégradation de la réponse due au bruit. En général, les algorithmes peuvent supporter un bruit gaussien de variance 10 sans trop affecter les résultats.

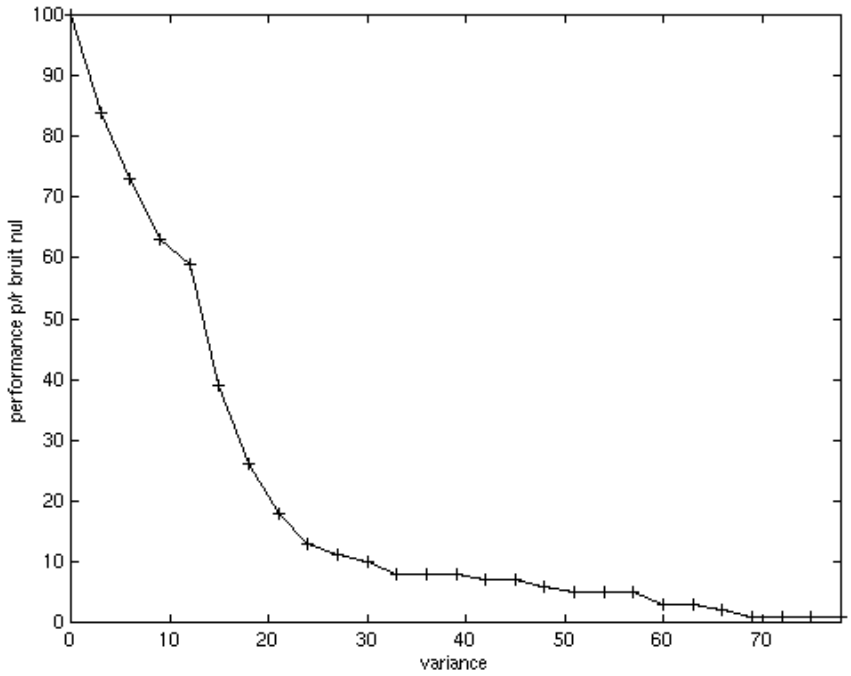

FIGURE 9. Résultats normalisés moyens obtenus avec les paramètres optimaux pour des images soumises à un bruit gaussien dont la variance est entre 0 et 78.

2.5.3 Etude des paramètres

Pour l'application de l'algorithme de Canny, le paramètre de la taille du filtre dépend évidemment du niveau de bruit de l'image. Un filtre 5 x 5 a cependant donné de bons résultats dans l'ensemble. D'autre part, une taille de 3 x 3 pour l'opérateur de calcul du gradient semble être la plus efficace dans le cas général. Pour des images dont les arêtes sont moins bien définies (wdg4), une taille supérieure peut être nécessaire. Finalement, le choix du seuil est particulier à chaque image. Les valeurs pour les images réelles (Ombre*, Paper*) se situent autour de 30.

Pour le traitement des contours, les paramètres dépendent moins de l'image originale. En général, une tolérance de 0.1 pour la courbure, une fenêtre de 10 entre les segments à fusionner, une distance maximale de 5 à 10 entre les points d'intersection et les segments donnent de bons résultats. Pour l'angle limite entre fusion et intersection, une valeur de 10 degrés donne de bons résultats dans l'ensemble.

2.5.4 Forces et faiblesses des algorithmes étudiés

On constante que les résultats pour les formes rectangulaires simples et bien définies sont excellents. Pour les images en 3D avec de l'ombre ou pour les arêtes ne présentant pas une grande variation d'intensité, les résultats sont moins intéressants. D'autre part, les détails impliquant plusieurs petits segments ne sont pas complètement représentés dans les segments résultants.

L'algorithme de Canny utilisé pour détecter les arêtes est efficaces pour détecter les changements d'intensité en échelon, les changements brusques d'une intensité à une autre. Cependant, il existe d'autres types d'arêtes, notamment les lignes pour lesquels Canny n'a pas une bonne réponse.

2.6 Autres algorithmes

2.6.1 Détecteur de Harris

Les points d'intérêt peuvent être obtenus directement de l'image. Il existe des détecteurs qui utilisent l'information locale autour d'un pixel pour déterminer son appartenance à la liste des points d'intérêt. Le détecteur de Harris [1] fait partie de cette catégorie. Pour chaque pixel, on calcule le "Average Squared Gradient", une matrice 2 x 2 obtenue à partir des composantes du gradient. La réponse au détecteur de Harris est calculée en utilisant le déterminant et la trace ce la matrice *M*:

$$R = \mathrm{Det}(M) - k\mathrm{Trace}(M)^2 \qquad \text{(EQ 15)}$$

Le détecteur de Harris est implanté pour les images en niveaux de gris et pour les images en couleur [9].

2.6.2 Détecteur “Laplacian of Gaussian”

L’algorithme de Canny utilise l’information de la première dérivée pour détecter les arêtes. Le détecteur “Laplacian of Gaussian” (LoG) [8, p.157] calcule la dérivée seconde pour identifier les maxima de la fonction image. Cependant, la grande sensibilité au bruit de la dérivée seconde peut entraîner des résultats moins intéressants que Canny:

TABLEAU 3. Identification des segments et des points d’intersection par Canny et LoG.

	house	Ombre0_OG4	Ombre0_OG5	Paper0_OG0	rlf1	sqr1	wdg4
Canny	65%	50%	58%	83%	100%	100%	100%
LoG	76%	40%	43%	63%	95%	100%	83%

Le détecteur LoG est implanté pour les images en niveaux de gris.

2.6.3 Points d’intersections

Les points d’intersections peuvent être utilisés pour estimer les extrémités des segments. Ainsi, à l’intérieur d’une certaine tolérance, chaque point d’extrémité est remplacé par le point d’intersection le plus près. Si les deux extrémités d’un segment devaient être remplacées par un même point, seule la plus proche est remplacée. Cette méthode donne de meilleurs résultats pour évaluer la position des segments:

TABLEAU 4. Identification des segments selon l’utilisation des points d’intersection (pi).

	house	Ombre0_OG4	Ombre0_OG5	Paper0_OG0	rlf1	sqr1	wdg4
Sans pi	59%	55%	61%	84%	100%	100%	100%
Avec pi	95%	58%	73%	84%	100%	100%	100%

3.0 Lecture et écriture d'images avec `ImageStream`

3.1 Aperçu général

La lecture et l'écriture d'images sur disque est une partie importante de plusieurs applications. Ces opérations peuvent sembler banales à première vue, mais il existe une multitude de formats et certains algorithmes utilisés pour la compression des données sont assez sophistiqués. D'autre part, il est tout à fait inutile de refaire le code pour chaque nouvelle application. C'est dans ce contexte qu'a été créée la classe `ImageStream`. La classe `ImageStream`, dérivée de la classe standard `fstream`, permet de faire le lien entre les images enregistrées sur le disque et les objets des classes `ImageGris` et `ImageRGB`. Elle permet, à l'aide des opérateurs d'insertion dans un flot et d'extraction d'un flot, d'écrire et de lire les images en différents formats standards. Les formats supportés sont:

- **En lecture**: SunRaster (1, 8, 24, 32 bits; RLE8), BMP (RGB 1, 4, 8, 24 bits; RLE4, RLE8), JPG, GIF, PNG, X Bitmap (Version 11), X Pixmap, PCX (1, 4, 8, 24 bits), X Window Dump, Portable Bitmap (PBM), Portable Graymap (PGM) et Portable Pixmap (PPM), ces trois derniers en format ASCII et RAW, FormatContour (format maison).
- **En écriture**: SunRaster (Niveaux de gris, RGB 24 bits), BMP (24 bits), JPG, GIF, PNG, X Bitmap (Version 11), Portable Bitmap (PBM), Portable Graymap (PGM) et Portable Pixmap (PPM), ces trois derniers en format ASCII, FormatContour (format maison).

Les formats JPEG, GIF, PNG et X Bitmap sont lus et écrits en utilisant les classes de Colosseum Builders [3]. Les autres formats utilisent du code fourni par des membres du LVSN ou écrit pour l'occasion. La classe permet la lecture et l'écriture des `ImageRGB` et `ImageGris` dans tous les formats. Pour les formats qui ne se prêtent pas à un ou l'autre des types d'images, la conversion est faite par `ImageStream` afin de se conformer au exigences du format de fichier.

TABLEAU 5. Formats des fichiers d'image supportés par `ImageStream`.

Format	Lecture	Écriture	Commentaires
SunRaster	Oui	Oui	
BMP	Oui	Oui	
JPEG	Oui	Oui	
GIF	Oui	Oui	Écriture sans compression.
PNG	Oui	Oui	
X Bitmap	Oui	Oui	Monochrome.
X Pixmap	Oui	Non	Support incomplet.
PCX	Oui	Non	
X Window Dump	Oui	Non	Support incomplet.
PBM	Oui	Oui	Monochrome; seuil à la valeur centrale sur `ImageGris`.
PGM	Oui	Oui	Niveaux de gris.
PPM	Oui	Oui	Couleur (RGB).
FormatContour	Oui	Oui	Format maison pour la classe `Contour`.

3.2 Conception de la classe `ImageStream`

La classe `ImageStream` a été développée de façon à être simple d'utilisation et facilement maintenable. La simplicité d'utilisation implique qu'on ne peut pas supposer que

l'utilisateur connaît le format d'image qu'il manipule. La maintenabilité est essentielle si on veut pouvoir intégrer facilement de nouveaux formats à la classe ou mettre à jour les formats existant.

3.2.1 Simplicité d'utilisation

La classe `ImageStream` prévoit trois mécanismes pour déterminer le format d'image en lecture. Deux de ces mécanismes sont mis en oeuvre à la création d'un objet et le troisième est appliqué lors de la lecture de l'image à partir du fichier. En écriture, seuls les deux premiers mécanismes s'appliquent.

Lors de la création d'un objet de la classe `ImageStream`, un nom de fichier et un mode d'accès sont spécifiés. En option, l'utilisateur peut préciser le format d'image avec lequel il travaille. Si ce type n'est pas spécifié, l'objet examine l'extension du nom de fichier pour obtenir une première estimation du type d'image utilisé. Si ces deux mécanismes ne peuvent pas déterminer un format, le type SunRaster est utilisé par défaut. En écriture, le format déterminé par ces mécanismes est celui utilisé par l'objet. Cependant, en lecture un troisième mécanisme plus robuste est prévu.

Dans le cas de la lecture d'une image sur le disque, `ImageStream` essaie tous les décodeurs avant de lancer une exception. Ce mécanisme permet d'éviter les erreurs dues au mauvais choix de format par l'utilisateur ou à la mauvaise extension de nom de fichier. Le premier décodeur essayé est bien entendu celui correspondant au format déterminé par la première estimation. Si ce décodeur ne peut pas reconnaître la signature correspondante, tous les autres décodeurs sont essayés jusqu'à ce la signature soit reconnue ou que tous les décodeurs aient produit une erreur.

La séquence des trois mécanismes permet à l'utilisateur de préciser le type de ses données, donc d'éviter de perdre le temps nécessaire à la détection automatique du format de fichier. D'autre part, l'utilisateur peut être totalement ignorant du type de ses données et être capable de les lire sans problèmes.

3.2.2 Maintenabilité

L'avantage de regrouper tous les formats dans une classe unique est que l'utilisateur n'a pas à se préoccuper de choisir le bon décodeur ou encodeur pour le fichier image d'intérêt. Cependant, chaque format d'image implique plusieurs dizaines de lignes de code. Il ne serait pas avisé de placer tout le code dans une même classe et espérer conserver une bonne maintenabilité. C'est pourquoi chaque format est encapsulé dans une classe indépendante.

Un objet de la classe `ImageStream` crée au besoin un objet de la classe décodeur ou encodeur pour le format désiré et tente de se traiter avec cet objet. Cette façon de faire facilite la maintenabilité du code. D'une part, chaque encodeur et décodeur est indépendant, de telle sorte qu'il peut être mis à jour sans affecter les autres. D'autre part, l'ajout d'un nouveau format de fichier n'implique pas de changements importants pour `ImageStream`. Le code est donc plus facile à maintenir.

3.2.3 Portabilité

Les classes décodeur et encodeur utilisent les types de donnée 8, 16 et 32 bits définis dans `datatypes.h`. D'autre part, les données lues sur fichier sont transformées en données système par les fonctions `LittleEndianToSystem` et `BigEndianToSystem` et les données écrites sur fichier sont transformées du système par les fonctions `SystemToLittleEndian` et `SystemToBigEndian`, selon le format s'appliquant spécifiquement au format de fichier traité. Ces fonction sont choisies à la compilation selon l'état du marqueur `BIGENDIAN_SYSTEM` défini dans `Systeme.hpp`. Le code peut donc être porté sur un système en modifiant ces deux fichiers d'entête pour répondre aux spécifications du système.

3.3 Gestion des erreurs

L'accès au disque implique des possibilités d'erreur. Ces erreurs sont signalées à l'utilisateur par le mécanisme des exceptions. Un objet de la classe `ImageStream` peut lancer certains types d'exception si des erreurs ne pouvant être corrigées sont rencontrées.

L'utilisateur peut s'attendre à recevoir deux types d'exception. Le premier, de type `ErreurImage` est lancé quand une erreur de lecture survient. Le deuxième type, `ErreurFormatImage` se divise en deux cas particuliers, selon que l'accès se fait en lecture ou en écriture. Une exception de type `ErreurFormatImageLecture` est lancée quand le format de fichier n'est pas supporté en lecture par la classe. De même, une exception de type `ErreurFormatImageEcriture` est lancée quand le format de fichier n'est pas supporté en écriture par la classe. De plus, les objets des classes propres à chaque format de fichier peuvent lancer des exceptions particulières si le fichier ne respecte pas les exigences du format.

Un objet de la classe `ImageStream` peut aussi lancer un autre type d'exception, qui est intercepté par l'objet lui-même. Il s'agit de `ErreurFormat` qui indique qu'un décodeur n'a pu reconnaître la signature attendue dans le fichier. L'objet essaie alors un autre décodeur jusqu'à ce qu'il les ait tous essayés. À la fin de cette séquence, une exception `ErreurFormatImageLecture` est lancée si le fichier n'a pu être décodé.

4.0 Conclusion

Les segments de droite et les points d'intersection sont deux caractéristiques de base qu'on peut extraire d'une image en niveaux de gris.

Dans un premier temps, les algorithmes utilisés ont été présentés. L'algorithme de Canny permet de repérer les points faisant partie des arêtes et de les regrouper en contours. Les algorithmes subséquents utilisés pour obtenir les segments de droite et les points d'intersection ont été décrits. Les classes C++ développées pour représenter et effectuer des traitements sur les images ont été présentés et les fonctions pertinentes, mises en évidence. Finalement, quelques résultats ont été discutés. En particulier, les segments et les points caractéristiques obtenus avec les images les plus simples et les moins affectées par le bruit correspondent parfaitement aux résultats attendus. D'autre part, les résultats obtenus sur les images complexes ou affectées par du bruit sont moins intéressants.

Dans un deuxième temps, un nouvel outil C++ pour accéder aux fichier images a été présenté. La classe `ImageStream` est facile d'utilisation et permet d'accéder aux formats de fichiers image les plus répandus.

En somme, nous croyons que les classes développées sont utilisables dans plusieurs contextes et qu'elles s'intègrent parfaitement au code déjà existant au LVSN. Leur nature modulaire permet d'ajouter et de mettre à jour les méthodes très facilement.

Bibliographie

[1] Stéphane Bres, Jean-Michel Jolion. (Page consultée le 29 juillet 1999). *Multiresolution Contrast Based Detection of Interest Points*. [En ligne]. Adresse URL: http://rfv.insa-lyon.fr/~jolion/PAPIERS/ptint/texte.html

[2] C. Wayne Brown, Barry J. Shepherd. *Graphics File Formats: reference and guide*. Prentice-Hall, 1995.

[3] Colosseum Builders. (Page consultée le 6 juillet 1999). *Image Library.* [En ligne]. Adresse URL: http://www.colosseumbuilders.com/sourcecode.htm

[4] R. Fisher, S. Perkins, A. Walker, E. Wolfart. (Page consultée le 6 juillet 1999). *Hypermedia Image Processing Reference*. J. Wiley & Sons. [En ligne]. Adresse URL: http://www.hig.no/bibliotek/hipr/html/

[5] Wolfgang Förstner. *A framework for low level feature extraction*. In ECCV, 1994.

[6] Rafael C. Gonzales, Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[7] R. Horaud, T. Skrordas, F. Veillon. *Finding geometric and relational structures in an image*. In ECCV, p. 374-384, 1990.

[8] Ramesh Jain, Rangachar Kasturi, Brian G. Schunck. *Machine Vision*. McGraw-Hill, 1995.

[9] P. Montesinos, V. Gouet, R. Deriche, D. Pelé. (Page consultée le 29 juillet 1999). *Differential Invariants for Color Images*. [En ligne]. Adresse URL: http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MONTESSINOS/cv_online.html

[10] Theodosios Palvidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, 1982.

[11] John C. Russ. *The image processing handbook*. 2nd edition. CRC Press, 1994.

[12] Cordelia Schmid, Roger Mohr, Christian Bauckhage. *Comparing and Evaluating Interest Points*. In International Conference on Computer Vision, IEEE Computer Society Press, janvier 1998.

Annexe A Documentation complète des classes développées

A.1 Image: représentation générale d'une image

Le patron `Image` permet de faire des traitements généraux sur des pixel de type quelconque (classe `T`). `Image` est dérivée publiquement du patron `Table2D` et possède deux données additionnelles (de type `T`): les valeurs minimale et maximale admises pour les pixels. Toutes les fonctions et opérateurs de `Image` font saturer les pixels à ces valeurs limites.

Le type `T` doit définir les opérations mathématiques, relationnelles et logiques, entre éléments de type `T`, entre un élément de la classe `T` et un `long`, entre un élément de la classe `T` et un `double`.

Le patron `Image` faut aussi appel aux classes `Operateur` et `Point`.

A.1.1 Constructeurs

- `Image();`

  Construire une image de taille nulle dont les pixels devront avoir une valeur entre 0 et 255.

- `Image(T min, T max);`

  Construire une image de taille nulle dont les pixels devront avoir une valeur entre `min` et `max`.

A.1.2 Opérateurs

- `const Image &operator=(const Image &rh);`

  Affecte l'image `rh` (pixels et valeurs limites) et retourne une référence constante sur l'objet courant.

- `T &operator()(const Point &pn) const;`

  Retourne une référence sur le pixel de coordonnée `pn` de l'objet courant.

- `const Image& operator*=(const Operateur &mask);`

  Convolution avec un objet de la classe `Operateur`. Retourne également une référence constante sur l'objet courant.

- `Image operator*(const Operateur &mask) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator-=(const Image &rh);`

  Valeur absolue de la différence entre les pixels de deux images. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator-(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator-=(const T &rh);`

  Valeur absolue de la différence entre les pixels de l'image courante et une constante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator-(const T &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator+=(const Image &rh);`

  Addition de la valeur des pixels de deux images. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator+(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator+=(const T &rh);`

  Addition d'une constante à chacun des pixels de l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator+(const T &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator*=(const Image &rh);`

  Multiplication pixels par pixels de deux images. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator*(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator*=(const double &rh);`

  Multiplication de chacun des pixels de l'image par une constante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator*(const double &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator/=(const Image &rh);`

  Division des pixels de l'image courante par les pixels de l'image `rh`. Une division par zéro entraîne un résultat à la valeur maximale pour le pixel concerné. Par contre, on considère que 0/0=1. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator/(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator/=(const double &rh);`

  Division des pixels de l'image courante par une constante. Lance une erreur si `rh` est nul. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator/(const double &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator&=(const Image &rh);`

  Opération ET bit à bit, pixel à pixel, pour chacun des pixels. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator&(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator&=(const T &rh);`

  Opération ET bit à bit, pour chacun des pixels avec la constante `rh`. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator&(const T &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator|=(const Image &rh);`

  Opération OU bit à bit, pixel à pixel, pour chacun des pixels. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator|(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator|=(const T &rh);`

  Opération OU bit à bit, pour chacun des pixels avec la constante `rh`. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator|(const T &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator^=(const Image &rh);`

  Opération OU EXCLUSIF bit à bit, pixel à pixel, pour chacun des pixels. Lance une erreur si l'image `rh` n'est pas de la même taille que l'image courante. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator^(const Image &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `const Image& operator^=(const T &rh);`

  Opération OU EXCLUSIF bit à bit, pour chacun des pixels avec la constante `rh`. Le résultat pour chaque pixel est conservé dans les limites admises par l'image courante. Cet opérateur retourne également une référence constante sur l'objet courant.

- `Image operator^(const T &rh) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `Image operator~() const;`

  Inversion de chaque bit de tous les pixels de l'image courante. Le résultat est conservé dans les limites admises. Retourne le résultat dans une nouvelle image, sans affecter l'objet courant.

- `operator T*() const;`

  Convertit une image en `T*` (opérateur de cast). L'objet courant n'est pas affecté directement par cet opérateur.

A.1.3 Fonctions d'interface

- `void BruitFlipBits(double prob=0.1);`

  Inverse `prob`% des bits composant l'image. Les résultats sont conservé dans les limites admises par l'objet.

- `void BruitGaussien(double sigma=5);`

  Ajoute un bruit gaussien d'écart type `sigma` à l'image.

- `void BruitImpulsion(double prob=0.1);`

  Place `prob`% des pixels de l'image à `MaxValue()`.

- `void BruitSaltPepper(double prob=0.1,double salt=0.5);`

  Place `prob*salt`% des pixels a la valeur maximale et `prob*(1-salt)` à la valeur minimale.

- `void FiltreConservateur(unsigned int _taille=3);`

  Filtre utilisé pour enlever les impulsions isolées. La valeur de chaque pixel est comparée aux valeurs dans un voisinâtes `_taille x _taille`. Si elle dépasse ou est inférieure aux extremum voisins, elle est ramenée à l'extremum approprié.

- `void FiltreGaussien(unsigned int _taille=3);`

  Appliquer un filtre gaussien de dimension `_taille x _taille`. Effectue une convolution avec un filtre horizontal *1 x _taille* puis avec un filtre vertical *_taille x 1*. Ecrit le résultat dans l'objet courant.

- `void FiltreMaxMin(unsigned int _taille=3);`

  Chaque pixel est remplace par la valeur moyenne du minimum et maximum dans son voisinage `_taille x _taille`.

- `void FiltreMediane(unsigned int _taille=3);`

  Trouve la médiane dans un voisinage `_taille x _taille` pour chaque pixel. Si le voisinage comprend un nombre pair de pixels, la moyenne des deux éléments médians est calculée. Le résultat est écrit dans l'objet courant.

- `void FiltreMoyenne(unsigned int _taille=3);`

  Calculer la moyenne dans un voisinage `_taille x _taille` pour chaque pixel et écrit le résultat dans l'objet courant.

- `void FiltreUnsharp(double k=0.5);`

  Mise en évidence des arêtes par des opérations arithmétiques. Obtient une version lissée de l'image (*Smoothed*) et calcule: *Image += k*(Image - Smoothed).* Les valeurs recommandées de `k` se situent entre 0.2 et 0.7.

- `Image & HistoEqual(double (*distrib)(double));`

  Modifie l'image pour que l'intégrale de son histogramme ait la forme de la fonction `distrib` qui reçoit un réel $0 \leq r \leq 1$ et qui retourne un réel $0 \leq s \leq 1$. `distrib` doit être croissante monotone. De plus, `distrib(1)` doit être égal à *1* pour que tous les pixels reçoivent une nouvelle valeur, sans quoi les pixels de valeur élevée sont mis à 0.

- `Table1D<unsigned long> Histogramme(unsigned long classes=0) const;`

  Evalue l'histogramme de l'objet courant. Par défaut, chaque valeur possible est représentée dans l'histogramme, à l'indice correspondant. On peut spécifier le nombre de 'classes' (valeurs de pixel) représentées, uniformément distribuées sur l'intervalle des valeurs admises par l'objet.

- `Image & HistoScale(T min, T max);`

  Modifie l'image pour que l'intensité de tous ses pixels soit située entre `min` et `max`. La fonction fait correspondre la valeur maximale originale à `max` et la valeur minimale originale à `min`. Les autres intensités sont distribuées uniformément sur l'intervalle.

- `Image RegionGet(int ligne, int col, int dligne, int dcol) const;`

  Copie une partie de l'image dans une nouvelle image retournée par la fonction. La région commence à `ligne`, `col` de l'objet courant et est large de `dcol` et haute de `dligne`. Une exception est lancée si la région commence à une coordonnée incorrecte ou si elle dépasse les limites de l'image courante.

- `void RegionPut(const Image&, int ligne, int col);`

  Copie une région dans une partie de l'image courante. La région est insérée à `ligne`, `col` et est constituée de l'image placée en argument. Une exception est lancée si la région commence à une coordonnée incorrecte ou si elle dépasse les limites de l'image courante.

- `T Min() const;`

  Valeur minimum des pixels de l'image.

- `T Max() const;`

  Valeur maximum des pixels de l'image.

- `void MaxMin(T &max, T &min) const;`

  Valeurs minimum et maximum des pixels de l'image.

- `T MinValue() const;`

  Valeur minimum permise pour les pixels de l'image.

- `T MaxValue() const;`

  Valeur maximum permise pour les pixels de l'image.

- `int Width() const;`

  Retourne la largeur, en pixels, de l'image. Même résultat que la fonction `Cols()` de `Table2D`.

- `int Height() const;`

  Retourne la hauteur, en pixels, de l'image. Même résultat que la fonction `Lignes()` de `Table2D`.

- `int Length() const;`

  Retourne le nombre total de pixels de l'image. Même résultat que la fonction `Dim()` de `Table1D`.

A.2 ImageRGB: représentation d'une image en couleur

La classe `ImageRGB` permet de représenter et de gérer des images en couleur. Elle est dérivée publiquement de `Image<unsigned long>`. Les composantes rouge, vert et bleu des pixels sont conservés dans des `unsigned long` de la manière suivante: 0x00RRGGBB.

La classe `ImageRGB` faut aussi appel aux classes `CouleurRGB`, `CouleurHSI`, `ImageGris`, `ImageGradient`, `FVimageRGB` et `BitmapImage` (Colosseum Builders) et `MonceauBin`.

A.2.1 Constructeurs

- `ImageRGB();`

  Construit une `Image` dont la valeurs des pixels peut varier de 0 à 0xFFFFFF.

A.2.2 Opérateurs

- `const ImageRGB & operator=(const ImageGris &rh);`

  Affecter une `ImageGris`. Les composantes rouge, vert et bleu de chaque pixel prennent tous la valeur d'intensité correspondante du pixel de `rh`. Retourne aussi une référence constante sur l'objet courant.

- `const ImageRGB & operator=(const BitmapImage &bi);`

  Affecter une `BitmapImage` de la librairie graphique de Colosseum Builders. Retourne aussi une référence constante sur l'objet courant.

- `operator BitmapImage() const;`

  Convertit en `BitmapImage` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur.

- `operator FVimageRGB*() const;`

  Convertit en `FVimageRGB*` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur. Permet une intégration dans un `FVdessin`.

A.2.3 Fonctions d'interface

La classe `ImageRGB` désactive les fonctions `HistoEqual(double (*distrib)(double))` et `Histogramme(unsigned long classes=0)` du patron Image.

- `Liste<Point> Harris(int np=0,double s=1,double k=0.06) const;`

Détection des points d'intérêt par le détecteur de Harris. Retourne la liste des `np` points ayant eu la réponse la plus grande (tous les points si `np` = 0). Utilise un filtre gaussien de variance `s` et calcule la réponse: Det(M) - `k` Trace(M)^2.

A.3 CouleurRGB: couleur par ses composantes RGB

La classe `CouleurRGB` permet de représenter et de gérer des couleurs par les composantes de base rouge, vert et bleu. Elle peut servir de décodeur pour les pixels d'une `ImageRGB` pour un accès direct aux composantes. La classe `CouleurRGB` possède trois données de type `unsigned char` accessibles publiquement, la valeur des composantes: `redValue`, `greenValue` et `blueValue`.

La classe `CouleurRGB` faut aussi appel à la classe `CouleurHSI`.

A.3.1 Constructeurs

- `CouleurRGB(unsigned char _r=0, unsigned char _g=0, unsigned char _b=0);`

  Construire une couleur dont les composantes rouge, vert et bleu sont `_r`, `_g` et `_b`.

A.3.2 Opérateurs

- `const CouleurRGB & operator=(const unsigned long val);`

  Affecter `val` à l'objet courant et en retourner une référence constante. On s'attend à ce que `val` soit composé de RGB de la manière suivante: 0x00RRGGBB.

- `const CouleurRGB & operator=(const CouleurHSI &);`

  Affecter un objet de la classe `CouleurHSI` à l'objet courant et en retourner une référence constante. La conversion d'une représentation à l'autre est effectuée.

- `operator unsigned long() const;`

  Convertir en `unsigned long` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur.

- `const CouleurRGB& operator-=(const CouleurRGB &);`

  Valeur absolue de la différence entre les composantes de deux couleurs. Cet opérateur retourne également une référence constante sur l'objet courant.

- `CouleurRGB operator-(const CouleurRGB &) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans un nouvel objet, sans affecter l'objet courant.

- `const CouleurRGB& operator+=(const CouleurRGB &);`

  Addition des composantes de deux couleurs. La valeur des composantes sature à 255. Cet opérateur retourne également une référence constante sur l'objet courant.

- `CouleurRGB operator+(const CouleurRGB &) const;`

  Comme l'opérateur précédent, mais retourne le résultat dans un nouvel objet, sans affecter l'objet courant.

- `friend ostream &operator<<(ostream &,const CouleurRGB &);`

  Insère une couleur dans le flot de sortie. Les valeur entière des composantes rouge. vert et bleu sont écrites dans cet ordre, séparées par des espaces.

- `friend istream &operator>>(istream &,CouleurRGB &);`

  Extrait une couleur du flot d'entrée. S'attend à recevoir les composantes rouge, vert et bleu dans cet ordre.

A.3.3 Fonctions d'interface

- `double Hue() const;`

  Retourne la valeur de hue correspondant aux valeur RGB de l'objet courant. La valeur retournée est en radiants et est située entre 0 et 2π. L'objet courant n'est pas affecté par cette fonction.

- `double Saturation() const;`

  Retourne la valeur de saturations correspondant aux valeurs RGB de l'objet courant. La valeur retournée est située entre 0 et 1. L'objet courant n'est pas affecté par cette fonction.

- `double Intensite() const;`

  Retourne la valeur d'intensité correspondant aux valeur RGB de l'objet courant. La valeur retournée est la moyenne des composantes et est située entre 0 et 255. L'objet courant n'est pas affecté par cette fonction.

- `unsigned long getInt() const;`

  Retourne un nombre entier représentant les trois composantes. La valeur retournée est située entre 0 et 0xFFFFFF. L'objet courant n'est pas affecté par cette fonction.

- `unsigned char getChar() const;`

  Retourne un nombre entier représentant l'intensité de la couleur. L'objet courant n'est pas affecté par cette fonction.

A.4 CouleurHSI: couleur par ses composantes HSI

La classe `CouleurHSI` permet de représenter et de gérer des couleurs par les composantes de base hue, saturation et intensité. Elle peut servir de décodeur pour les pixels d'une `ImageRGB` pour un accès direct aux composantes. La classe `CouleurHSI` possède trois données de type `double` accessibles publiquement, la valeur des composantes: `hue`, `saturation` et `intensite`.

La classe `CouleurHSI` faut aussi appel à la classe `CouleurRGB`.

A.4.1 Constructeurs

- `CouleurHSI(double _h=0, double _s=0, double _i=0);`

  Construire une couleur dont les composantes hue, saturation et intensité sont `_h`, `_s` et `_i` respectivement.

A.4.2 Opérateurs

- `const CouleurHSI & operator=(const unsigned long val);`

  Affecter `val` à l'objet courant et en retourner une référence constante. On s'attend à ce que `val` soit composé de RGB de la manière suivante: 0x00RRGGBB.

- `const CouleurHSI & operator=(const CouleurRGB &);`

  Affecter un objet de la classe `CouleurRGB` à l'objet courant et en retourner une référence constante. La conversion d'une représentation à l'autre est effectuée.

- `operator unsigned long() const;`

  Convertit en `unsigned long` (opérateur de cast). N'affecte pas l'objet courant.

- `friend ostream &operator<<(ostream &,const CouleurHSI &);`

  Insère une couleur dans le flot de sortie. Les valeur réelles des composantes hue. saturation et intensité sont écrites dans cet ordre, séparées par des espaces.

- `friend istream &operator>>(istream &,CouleurHSI &);`

  Extrait une couleur du flot d'entrée. S'attend à recevoir les composantes hue, saturation et intensité dans cet ordre.

A.4.3 Fonctions d'interface

- `int Valider() const;`

  Vérifie que les composantes ont une valeur à l'intérieur de leur intervalle respectif. 0 à 255 pour `intensite`, 0 à 1 pour `saturation` et 0 à 2π pour `hue`. Une erreur est lancée si un des intervalles n'est pas respecté. Si tous les intervalles sont respectés, la valeur 1 est retournée. L'objet courant n'est pas affecté par cette fonction.

- `unsigned char Red() const;`

  Retourne la valeur de rouge correspondant aux valeurs HSI de l'objet courant. La valeur retournée est située entre 0 et 255. N'affecte pas l'objet courant.

- `unsigned char Green() const;`

  Retourne la valeur de vert correspondant aux valeurs HSI de l'objet courant. La valeur retournée est située entre 0 et 255. L'objet courant n'est pas affecté par cette fonction

- `unsigned char Blue() const;`

  Retourne la valeur de bleu correspondant aux valeurs HSI de l'objet courant. La valeur retournée est située entre 0 et 255. L'objet courant n'est pas affecté par cette fonction

- `unsigned long getInt() const;`

  Retourne un nombre entier représentant les trois composantes. La valeur retournée est située entre 0 et 0xFFFFFF. L'objet courant n'est pas affecté par cette fonction.

- `unsigned char getChar() const;`

  Retourne un nombre entier représentant l'intensité de la couleur. L'objet courant n'est pas affecté par cette fonction.

A.5 ImageGradient: calcul et représentation du gradient d'une image

La classe `ImageGradient` permet de calculer et conserver le gradient d'une image en niveaux de gris. Elle est dérivée publiquement de `Image<int>`. Une donnée accessible publiquement, de type `Image<int>` et nommée `direction` conserve l'orientation du gradient alors que la partie héritée de `Image` conserve le module du gradient, plus souvent utilisé dans les calculs.

La classe `ImageGradient` faut aussi appel aux classes `ImageGris` et `FVimage-Gris`.

A.5.1 Constructeurs

- `ImageGradient();`

  Construit une `Image` dont la valeurs des pixels peut varier de -20000 à 20000.

A.5.2 Opérateurs

- `const ImageGradient & operator=(const ImageGris &rh);`

  Affecter une `ImageGris`. Les pixels prennent directement les valeurs d'intensité des pixel de `rh`. Retourne une référence constante sur l'objet courant.

- `const ImageGradient & operator=(const Image<int> &);`

  Affecter une `Image<int>`. Fonction incluse pour éviter la confusion entre les différents opérateurs d'affectation pour le compilateur.

- `const ImageGradient & operator=(const ImageGradient &);`

  Affecter une `ImageGradient`. Fonction incluse pour éviter la confusion entre les différents opérateurs d'affectation pour le compilateur.

- `operator FVimageGris*() const;`

  Convertit en `FVimageGris*` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur. Permet une intégration dans un `FVdessin`.

A.5.3 Fonctions d'interface

- `ImageGradient & Gradient(const ImageGris&, int _taille=2);`

  Obtention du module du gradient de l'image d'entrée en utilisant l'opérateur de Roberts de `_taille x _taille`. Le module du gradient est affecté à l'objet courant et une référence constante en est retournée. La variable membre `direction` prend une taille nulle après l'utilisation de cette fonction.

- `ImageGradient & GradientMaxima(const ImageGris &, int _taille=2);`

  Obtention des maxima locaux du module du gradient de l'image d'entrée en utilisant l'opérateur de Roberts de `_taille x _taille`. Le module du gradient aux maxima locaux est affecté à l'objet courant. La variable membre `direction` prend la valeur de l'orientation du gradient en chaque pixel.

- `ImageGradient & GradientMaxima(const ImageGris &, const Operateur& dx, const Operateur& dy);`

  Obtention des maxima locaux du module du gradient de l'image d'entrée en utilisant les opérateurs fournis par l'usager pour calculer le gradient en x et en y. Le module du gradient aux maxima locaux est affecté à l'objet courant et une référence constante en est retournée. La variable membre `direction` prend la valeur de l'orientation du gradient en chaque pixel.

- `void GradXY(const ImageGris&, ImageGradient& gx, ImageGradient& gy, int _taille) const;`

  Obtention des composantes x et y du gradient de l'image d'entrée en utilisant l'opérateur de Roberts de `_taille x _taille` pour le calcul du gradient. Le résultat pour chaque composante est placé dans `gx` et `gy` respectivement. L'objet courant n'est pas affecté par cette fonction.

- `void GradXY(const ImageGris&, ImageGradient&gx, ImageGradient&gy, const Operateur& dx, const Operateur& dy) const;`

  Obtention des composantes x et y du gradient de l'image d'entrée en utilisant les opérateurs fournis par l'usager pour le calcul du gradient. Le résultat pour chaque composante est placé dans `gx` et `gy` respectivement. L'objet courant n'est pas affecté par cette fonction.

- `void DirectionGradient(Image<int>&, const ImageGradient& gx, const ImageGradient& gy) const;`

  Obtention de la direction du gradient dont les composantes sont `gx` et `gy`. Le résultat est écrit en degrés (0..360) dans l'objet `Image<int>` passé par référence. L'objet courant n'est pas affecté par cette fonction.

- `void ModuleGradient(ImageGradient&, const ImageGradient& gx, const ImageGradient& gy) const;`

  Obtention du module du gradient dont les composantes sont `gx` et `gy`. Le résultat est écrit dans l'objet `ImageGradient` passé par référence. L'objet courant n'est pas affecté par cette fonction.

A.6 ImageGris: représentation d’une image en niveaux de gris

La classe `ImageGris` permet de représenter et de gérer des image en niveaux de gris. Elle est dérivée publiquement de `Image<unsigned char>`.

La classe `ImageGris` faut aussi appel aux classes `ImageRGB`, `ImageGradient`, `ImageBinaire`, `FVimageGris` et `BitmapImage` (Colosseum Builders) et `MonceauBin`.

A.6.1 Constructeurs

- `ImageGris();`

  Construit une `Image` dont la valeurs des pixels peut varier de 0 à 255.

A.6.2 Opérateurs

- `const ImageGris & operator=(const ImageRGB &);`

  Affecter une `ImageRGB`. Les pixels prennent les valeurs d’intensité de l’objet `ImageRGB`. Retourne aussi une référence constante sur l’objet courant.

- `const ImageGris & operator=(const ImageGradient &);`

  Affecter une `ImageGradient`. Les valeurs des pixels de l’objet `ImageGradient` sont mises à l’échelle pour respecter les valeurs permises dans l’objet courant. Retourne aussi une référence constante sur l’objet courant.

- `const ImageGris & operator=(const Table2D<unsigned> &);`

  Affecter une `Table2D<unsigned>`. Les valeurs des pixels de l’objet `Table2D<unsigned>` sont mises à l’échelle pour respecter les valeurs permises dans l’objet courant. Retourne aussi une référence constante sur l’objet courant.

- `const ImageGris & operator=(const BitmapImage &);`

  Affecter une `BitmapImage` de la librairie graphique de Colosseum Builders. Les pixels prennent les valeurs d’intensité de l’objet `BitmapImage`. Retourne aussi une référence constante sur l’objet courant.

- `const ImageGris& operator=(const ImageGris &);`

  Affecter une `ImageGris`. Fonction incluse pour éviter la confusion entre les différents opérateurs d’affectation pour le compilateur.

- `const ImageGris& operator=(const Image<unsigned char> &);`

  Affecter une Image<unsigned char>. Fonction incluse pour éviter la confusion entre les différents opérateurs d’affectation pour le compilateur.

- `operator BitmapImage() const;`

  Convertit en `BitmapImage` (opérateur de cast). N’affecte pas l'objet courant.

- `operator FVimageGris*() const;`

  Convertit en `FVimageGris*` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur. Permet une intégration dans un `FVdessin`.

A.6.3 Fonctions d'interface

- `Contour Canny(int t2=20, int g=3, int f=5) const;`

  Détection d'arêtes avec Canny. Le filtre gaussien utilisé a une taille `f` et l'opérateur de calcul de gradient est l'opérateur de Roberts de taille `g`. Les seuil inférieur pour la détection des arêtes est `t1=t2/2` et le seuil moins permissif est `t2`. Le résultat est retourné dans un objet `Contour`, créé par la fonction. L'objet courant n'est pas affecté par cette fonction.

- `Contour Canny(const Operateur &dx, const Operateur &dy, int t2=20,int f=5) const;`

  Détection d'arêtes avec Canny. Le filtre gaussien utilisé a une taille `f` et l'opérateur de calcul de gradient est définit par l'utilisateur, tant en *x* qu'en *y*. Les seuil inférieur pour la détection des arêtes est `t1=t2/2` et le seuil moins permissif est `t2`. Le résultat est retourné dans un objet `Contour`, créé par la fonction. L'objet courant n'est pas affecté par cette fonction.

- `void Canny(Contour&, int t2=20, int g=3, int f=5) const;`

  Détection d'arêtes avec Canny. Le filtre gaussien utilisé a une taille `f` et l'opérateur de calcul de gradient est l'opérateur de Roberts de taille `g`. Les seuil inférieur pour la détection des arêtes est `t1=t2/2` et le seuil moins permissif est `t2`. Le résultat est écrit dans un objet `Contour`, passé par référence. L'objet courant n'est pas affecté par cette fonction.

- `void Canny(Contour&, const Operateur& dx, const Operateur& dy, int t2=20, int f=5) const;`

  Détection d'arêtes avec Canny. Le filtre gaussien utilisé a une taille `f` et l'opérateur de calcul de gradient est définit par l'utilisateur, tant en *x* qu'en *y*. Les seuil inférieur pour la détection des arêtes est `t1=t2/2` et le seuil moins permissif est `t2`. Le résultat est écrit dans un objet `Contour`, passé par référence. L'objet courant n'est pas affecté par cette fonction.

- `Liste<Point> Harris(int np=0,double s=1,double k=0.06) const;`

  Détection des points d'intérêt par le détecteur de Harris. Retourne la liste d'au plus `np` points ayant eu la réponse la plus grande (tous les points si `np` = 0). Utilise un filtre gaussien de variance `s` et calcule la réponse: Det(M) - `k` Trace(M)^2.

- `Liste<Point> Horaud(int w=5,int e1=5,int t=20) const;`

  Détection des points d'intérêt par le détecteur de Horaud. Recherche les points situés dans une fenêtre `w` et à angle `e1` des segments trouvés par Canny avec le seuil `t`.

- `void LoG(Contour &, int f=5, int t=20) const;`

  Détection d'arêtes le "Laplacian of Gaussian". Le filtre gaussien utilisé a une taille `f`. L'opérateur pour le calcul du laplacien est donné par défaut dans la classe `Operateur`. Le gradient de l'image est calculé et comparé au seuil `t`. Seules les arêtes trouvées par les deux opérateurs sont conservées. Le résultat est écrit dans un objet `Contour`, passé par référence. L'objet courant n'est pas affecté par cette fonction.

A.7 ImageBinaire: représentation d'une image binaire

La classe `ImageBinaire` permet de représenter et de gérer des image binaires. Elle est dérivée publiquement de `ImageGris`. La classe possède aussi une énumération d'outils statistiques permettant de faire des conversion d'images à multiples valeurs possibles vers une `ImageBinaire`.

La classe `ImageBinaire` faut aussi appel aux classes `Kernel`, `Ensemble`, `TablePile`, `ListeTri` et `Equivalence`.

A.7.1 Constructeurs

La classe `ImageBinaire` ne définit pas de constructeur.

A.7.2 Opérateurs

La classe `ImageBinaire` ne définit pas d'opérateur.

A.7.3 Fonctions d'interface

- `ImageBinaire& Treshold(const ImageGradient&, int);`

  Assigne la valeur ON aux pixels de l'objet courant correspondant aux pixels de l'image d'entrée dont la valeur est supérieure ou égale au seuil global spécifié, assigne la valeur OFF autrement.

- `ImageBinaire& Treshold(const ImageGris&,unsigned char);`

  Assigne la valeur ON aux pixels de l'objet courant correspondant aux pixels de l'image d'entrée dont la valeur est supérieure ou égale au seuil global spécifié, assigne la valeur OFF autrement.

- `ImageBinaire& Treshold(const ImageGris&, const Ensemble&);`

  Assigne la valeur ON aux pixels de l'objet courant correspondant aux pixels de l'image d'entrée dont la valeur fait partie de l'ensemble spécifié, assigne la valeur OFF autrement.

- `ImageBinaire& AdaptiveTresholding(const ImageGris&, Statistique s=MOYENNE, int v=7, int cte=5);`

  Calcule un seuil local pour chaque pixel en utilisant la `Statistique s` sur le voisinage `v`. Par la suite, chaque pixel de l'image d'entrée est comparé à son seuil (statistique - `cte`) et les pixels de l'objet courant prennent la valeur ON si le seuil est égalé ou dépassé, OFF autrement.

- `ImageBinaire& IterativeTresholding(const ImageGris&, unsigned char _t=128);`

  Recherche itérative d'un seuil global telle que décrite dans [Machine Vision, p. 83]. Le seuil est ensuite appliqué et le résultat écrit dans l'objet courant.

- `ImageBinaire& DoubleTresholding(const ImageGris&, unsigned char _t1, unsigned char _t2);`

  Application un double seuil telle que décrite dans [Machine Vision, p.85]. Le résultat est écrit dans l'objet courant.

- `ImageBinaire& ZeroCrossing(const ImageGradient&);`

  Recherche la présence d'un passage par zéro sur une `ImageGradient` dont on a calculé le Laplacien. Le passage par zéro est détecté par la présence d'un pixel à zéro ou par la présence de deux pixels contigus de signe opposé. Les pixels où un passage par zéro est détecté sont inscrit à ON dans l'objet courant, les autres sont placés à OFF.

- `void Erode(const Kernel &kernel=Kernel(Kernel::CROIX));`

  Applique une opération d'érosion par un élément structurant `kernel` sur l'objet courant.

- `void Dilate(const Kernel &kernel=Kernel(Kernel::CROIX));`

  Applique une opération de dilatation par un élément structurant `kernel` sur l'objet courant.

- `void Open(const Kernel &kernel=Kernel(Kernel::CROIX));`

  Opération d'ouverture: érosion suivie de dilatation par l'élément structurant `kernel`.

- `void Close(const Kernel &kernel=Kernel(Kernel::CROIX));`

  Opération de fermeture: dilatation suivie de érosion par l'élément structurant `kernel`.

- `void Expand(unsigned int k=1);`

  Agrandir les région actives `k` fois. A chaque passage, les pixels inactifs qui ont un (4-) voisin à ON sont aussi placé à ON.

- `void Shrink(unsigned int k=1);`

  Diminuer les région actives `k` fois. A chaque passage, les pixels actifs qui ont un (4-) voisin à OFF sont aussi placé à OFF.

- `void Skeleton();`

  Transforme l'objet courant en squelette des régions actives.

- `void Boundary();`

  Transforme l'objet courant en frontière des régions actives (place les pixels intérieurs à OFF).

- `Table2D<unsigned int> ComponentLabeling() const;`

  Assigne une étiquette à chaque région (les étiquettes varient de 1 à n, où n est le nombre total de régions distinctes). Le résultat est inscrit dans un tableau en deux dimensions où chaque case représente un pixel de l'image.

A.8 Contour: contours et points d'intersection

La classe `Contour` permet de représenter et de gérer les contours d'une image. Elle est dérivée publiquement de `Liste<ContourSegment>`. Une donnée accessible publiquement, de type `Liste<Point>` et nommée `intersections` conserve les points d'intersection entre les segments de droite. La classe `Contour` définit également la classe `HoraudRelation` qui est une liste de `ContourSegment*`. Cette classe est utilisée pour construire le graphe InterLevel du détecteur de Horaud et peut représenter une jonction, une courbe ou un ruban.

La classe `Contour` faut aussi appel aux classes `FVdessin`, `ListeTri`, `Table1D`, `ContourSegment`, `Point` et `ImageBinaire`.

A.8.1 Constructeurs

- `Contour();`

  Construire une liste de contours et de points d'intersections vide.

A.8.2 Opérateurs

- `friend ostream &operator<<(ostream &out,const Contour &c);`

  Insère le contour `c` dans le flot de sortie `out`. Tous les points formant chaque segment sont écrits suivis de la liste des points d'intersection. L'objet courant n'est pas affecté par cet opérateur.

- `friend istream &operator>>(istream &in,Contour &c);`

  Extrait le contour `c` du flot d'entrée `in`. On s'attend à recevoir une liste de `ContourSegment` suivie de la `Liste<Point>` des points d'intersection.

A.8.3 Fonctions d'interface

- `Contour &EdgeLinking(ImageBinaire &t1, ImageBinaire &t2);`

  Lier les arêtes en contours à partir de deux seuils. On suppose que l'image `t1` contient le résultat d'un seuil plus tolérant et que `t2` contient le résultat d'un seuil plus élevé.

- `Contour &EdgeLinking(ImageBinaire &Tim);`

  Lier les pixels actifs et se touchant de `Tim` en contours. On suppose que l'image d'entrée contient une représentation binaire d'arêtes.

- `void Abouter(const unsigned int angle=10,const double dist=10);`

  Fusionner les segments dont la direction diffère d'au plus `angle` et dont les deux extrémités les plus rapprochées sont séparées d'au plus `dist`. Les segment fusionnés doivent également former un segment plus long que chacun des segments originaux. L'algorithme de fusion des segments est implanté en incrémentant le critère de distance de 0 à `dist`. Quand des segments sont fusionnés, on ne conserve que les extrémités du segment résultant.

- `void Filtrer(const double len=10);`

  Retirer les segments dont la distance entre les points d'extrémité est inférieure à `len`.

- `void RegrouperUnitaires(double maxdist=10);`

  Regrouper les segments unitaires (formés d'un seul point) en groupes de deux s'ils sont séparés d'une distance `maxdist` ou moins.

- `void SegmenterCourbes(const double maxerreur=0.1);`

  Séparer les segments potentiellement courbés en segments dont le point réel le plus éloigné est situé est à `maxerreur` (normalisée par la longueur de du segment) de la droite qui passe par les points d'origine et de fin.

- `void TrouverIntersections(double tol=5,int ang=10,double distmin=5);`

  Rechercher les intersections entre les segments de droite. Les points sont calculés en utilisant l'équation de la droite passant par les extrémités de chaque segment. Les points d'intersection acceptés sont situés à une distance de `tol` ou moins de chacun des segment se croisant à un angle d'au moins `ang`. Les points d'intersection doivent être séparés d'au moins `distmin`. Les résultats sont écrits dans la variable membre `intersections`.

- `void SegmentsDroite(unsigned int maxangle=10, double maxdist=10, double tolinter=5, double tolcourbe=0.1);`

  Effectue tous les traitements nécessaires pour obtenir des segments droits (erreur normalisée maximale inférieure ou égale à `tolcourbe`) et les intersections correspondantes (situées au maximum à une distance `tolinter` des segment se croisant à un angle d'au moins `maxangle` et séparées d'au moins `tolinter`). Les segments fusionnés ont une orientation qui diffère d'au plus `maxangle` degrés et sont séparés d'une longueur inférieure ou égale à `maxdist`.

- `void AjusterSegments(double tol=5);`

  Remplace, dans les limites de `tol`, chaque extrémité des segments par le point d'intersection le plus près. Cette fonction assume que la liste des intersections a déjà été produite par `TrouverIntersections` ou `SegmentsDroite`. La liste des intersection n'est pas modifiée par cette fonction.

- `Liste<Point> HoraudJunctionDetection(Liste<HoraudRelation>&, int e1=15, int w=5, double max_o=5) const;`

  Recherche les jonctions telles que définies par le détecteur de Horaud. Retourne aussi la liste des *i* points d'intersection (Angles) correspondant aux *i* premières jonctions retournées. Une différence d'orientation supérieur à `e1` entraîne un angle, sinon une colinéarité. Pour chaque extrémité de segment, on examine un voisinage `w` x `w`. On permet une superposition de longueur `max_o` sur les segment colinéaires. L'objet courant n'est pas affecté par cette fonction. Cependant, la liste des jonction est constituée de pointeurs sur les segments de l'objet courant. Tout changement dans l'objet courant rendra invalide l'objet `Liste<HoraudRelation>` obtenu par cette fonction.

- `void HoraudCurveDetection(const Liste<HoraudRelation>&, Liste<HoraudRelation>&,int e2=0,int e3=90) const;`

  Recherche les courbes telles que définies par le détecteur de Horaud. L'objet courant n'est pas affecté par cette fonction. Cependant, la liste des courbes est constituée de pointeurs sur les segments de l'objet courant. Tout changement dans l'objet courant rendra invalide l'objet `Liste<HoraudRelation>` obtenu par cette fonction.

- `void Vider();`

  Vider la liste des points d'intersection et la liste des segments.

- `void EcrireSegments(ostream & out) const;`

  Ecrire les segments de contour dans le flot `out`, en n'écrivant que les points d'extrémités de chaque segment. Les points d'intersection ne sont pas écrits. L'objet courant n'est pas affecté par cette fonction.

- `FVdessin & Afficher(FVdessin &des, FVcouleur arcol=FVrouge, FVcouleur socol=FVbleu) const;`

  Insérer les contours de l'objet courant dans `des`. Les segments sont de couleur `arcol` et les points d'intersection sont de couleur `socol`. L'objet courant n'est pas affecté par cette fonction.

A.9 ContourSegment: liste de points d'un même segment

La classe `ContourSegment` permet de représenter et de gérer des segment. Elle est dérivée publiquement de `Liste<Point>`.

La classe `ContourSegment` faut aussi appel aux classes `FVligne` et `Point`.

A.9.1 Constructeurs

La classe `ContourSegment` ne définit pas de constructeur.

A.9.2 Opérateurs

- `operator FVligne*() const`

  Convertit en `FVligne*` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur. Permet une intégration dans un `FVdessin`. On suppose qu'un point courant à l'origine du segment a déjà été inséré dans l'objet `FVdessin`.

- `friend ostream &operator<<(ostream &out,const ContourSegment &sc);`

  Insère la liste des points du segment `sc` dans le flot `out`. N'affecte pas l'objet courant.

- `friend istream &operator>>(istream &in,ContourSegment &sc);`

  Extrait la liste des points formant le segment `sc` du flot d'entrée `in`.

A.9.3 Fonctions d'interface

- `Point & Origine() const;`

  Retourne une référence au point d'origine du segment (le premier à avoir été inséré dans la liste). L'objet courant n'est pas affecté par cette fonction.

- `Point & Fin() const;`

  Retourne une référence au point de fin du segment (le dernier à avoir été inséré dans la liste). L'objet courant n'est pas affecté par cette fonction.

- `double DroiteMaxErreur(Point &pme) const;`

  Calcule l'erreur maximale entre les points réels du segment et le segment de droite joignant l'origine et la fin. Le point où l'erreur est maximale est retourné dans `pme`, passé par référence. L'objet courant n'est pas affecté par cette fonction.

- `ContourSegment Separer(const Point &ps);`

  Séparer le segment courant en deux à partir du point `ps`. Les points situés entre `ps` et le point de fin sont retirés du segment courant et retournés dans un nouveau segment.

- `int Orientation() const;`

  Calcule l'orientation du segment de droite joignant les points d'extrémité. L'orientation est retournée est entière et est située entre 0 et 180 degrés. N'affecte pas l'objet courant.

- `void EquationDroite(int &a, int &b, int &c) const;`

  Retourne l'équation implicite de la droite passant par les extrémités du segment courant. Les coefficients `a`, `b` et `c` de l'équation $ax+by+c=0$ sont écrits dans les variables passées par référence. L'objet courant n'est pas affecté par cette fonction.

- `double LongueurDroite() const;`

  Retourne la longueur du segment de droite joignant les extrémités du segment courant. L'objet courant n'est pas affecté par cette fonction.

- `int DiffOrientation(const ContourSegment &) const;`

  Retourne la différence entre l'orientation de l'objet courant et d'un autre segment. La valeur retournée est située entre 0 et 180 degrés. L'objet courant n'est pas affecté par cette fonction.

- `Point Intersection(const ContourSegment &) const;`

  Retourne le point d'intersection entre l'objet courant et un autre segment. La fonction lance une exception s'il n'existe pas de point d'intersection. L'objet courant n'est pas affecté par cette fonction.

- `int Intersection(const ContourSegment &, Point &, double tol=0) const;`

  Point d'intersection entre deux segment droits, situé a une distance maximale `tol` de chacun des segments. A la sortie, le point passé par référence contient la coordonnée du de l'intersection si elle existe et la fonction retourne la validité de ce point par rapport à la tolérance. Aucun des deux segments n'est affecté par cette fonction.

- `double Distance(const Point&) const;`

  Retourne la distance minimale entre un point et le segment de droite passant par les extrémités du segment courant. L'objet courant n'est pas affecté par cette fonction.

- `void EcrireExtremites(ostream &out) const;`

  Ecrit les extrémités du point courant dans le flot de sortie `out`. La sortie prend la forme "`Origine()` -> `Fin()`". L'objet courant n'est pas affecté par cette fonction.

A.10 Point: coordonnée d'un point dans un plan

La classe `Point` permet de représenter et de gérer la coordonnée d'un point dans un plan ou d'un pixel dans une image. Elle possède deux variables membre accessibles publiquement de type `int` nommées `ligne` et `col`.

La classe `Point` faut aussi appel aux classes `FVptcour` et `Chaine`.

A.10.1 Constructeurs

- `Point(int _l=0, int _c=0);`

  Construire un point situé à la ligne `_l` et à la colonne `_c`.

A.10.2 Opérateurs

- `int operator< (const Point &p) const;`

  Retourne la valeur logique à savoir si le point courant est inférieur au point `p`. C'est le cas si $y_{cour} < y_p$ ou ($y_{cour} = y_p$ et $x_{cour} < x_p$). N'affecte pas l'objet courant.

- `int operator> (const Point &p) const;`

  Retourne la valeur logique correspondant au fait que le point courant est supérieur ou non au point `p`. L'objet courant n'est pas affecté par cet opérateur.

- `int operator<=(const Point &p) const;`

  Retourne la valeur logique à savoir si le point courant est inférieur ou égal au point `p`. L'objet courant n'est pas affecté par cet opérateur.

- `int operator>=(const Point &p) const;`

  Retourne la valeur logique à savoir si le point courant est supérieur ou égal au point `p`. L'objet courant n'est pas affecté par cet opérateur.

- `int operator==(const Point &p) const;`

  Retourne la valeur logique à savoir si le point courant est égal au point `p`. L'objet courant n'est pas affecté par cet opérateur.

- `int operator!=(const Point &p) const;`

  Retourne la valeur logique à savoir si le point courant est différent du point `p`. L'objet courant n'est pas affecté par cet opérateur.

- `friend ostream &operator<<(ostream &out,const Point &p);`

  Insère le point `p` dans le flot de sortie `out`. La sortie prend la forme "(`col`,`ligne`)" L'objet courant n'est pas affecté par cet opérateur.

- `friend istream &operator>>(istream &in,Point &p);`

  Extrait le point `p` du flot d'entrée `in`. On s'attend à recevoir la coordonnée en x (`col`) en premier suivie de la coordonnée en y (`ligne`). Les coordonnées peuvent être entourées de parenthèses ou non et être séparées d'une virgule ou non. Les espaces avant et entre les coordonnées sont aussi ignorés.

- `operator FVptcour*() const;`

  Convertit en `FVptcour*` (opérateur de cast). L'objet courant n'est pas affecté par cet opérateur. Permet une intégration dans un `FVdessin`.

A.10.3 Fonctions d'interface

- `int Chessboard(const Point &p) const;`

  Retourne la distance "chessboard" entre le point courant et le point `p`. L'objet courant n'est pas affecté par cette fonction.

- `int CityBlock(const Point &p) const;`

  Retourne la distance "CityBlock" entre le point courant et le point `p`. L'objet courant n'est pas affecté par cette fonction.

- `double Euclidean(const Point &p) const;`

  Retourne la distance euclidienne entre le point courant et le point `p`. L'objet courant n'est pas affecté par cette fonction.

- `int X() const;`

  Retourne la coordonnée en x (`col`) de l'objet courant. L'objet courant n'est pas affecté par cette fonction.

- `int Y() const;`

  Retourne la coordonnée en y (`ligne`) de l'objet courant. L'objet courant n'est pas affecté par cette fonction.

- `void X(int _x);`

  Assigne la coordonnée en x (`col`) de l'objet courant à `_x`.

- `void Y(int _y);`

  Assigne la coordonnée en y (`ligne`) de l'objet courant à `_y`.

A.11 Operateur: opérateur de convolution

La classe `Operateur` permet de représenter et de gérer des opérateurs de convolution. Elle est dérivée publiquement de `Table2D<int>`. La classe possède une énumération de types définis d'opérateurs. Elle possède quatre variables (`int`) accessibles publiquement décrivant la position de l'origine. Elle possède également une variable membre de type `Point` donnant la position de l'origine, une de type `int` donnant le facteur de division à appliquer lors de la convolution et une variable donnant le type de l'opérateur.

La classe `Operateur` faut aussi appel à la classe `Point`.

A.11.1 Constructeurs

- `Operateur(OperateurType _type=IDENTITE);`
  Construire un opérateur de type `_type`.
- `Operateur(OperateurType _type, unsigned int _a);`
  Construire un opérateur de type `_type` et de taille `_a`.

A.11.2 Opérateurs

- `operator int*() const;`
  Convertit en `int*` (opérateur de cast). Cet opérateur n'affecte pas l'objet courant.

A.11.3 Fonctions d'interface

- `void defType(OperateurType _type);`
  Définir le type de l'opérateur courant comme étant `_type`.
- `void defType(OperateurType _type, unsigned int _a);`
  Définir le type de l'opérateur courant comme étant `_type` de taille `_a`.
- `void defIdentite();`
  L'opérateur courant prend le type identité.
- `void defSobelH();`
  Définir l'objet courant comme l'opérateur de Sobel pour arêtes horizontales.
- `void defSobelV();`
  Définir l'objet courant comme l'opérateur de Sobel pour arêtes verticales.
- `void defRobertsH();`
  Définir l'objet courant comme l'opérateur de Roberts pour arêtes horizontales.
- `void defRobertsV();`
  Définir l'objet courant comme l'opérateur de Roberts pour arêtes verticales.
- `void defRobertsH(unsigned int _a);`
  Opérateur de Roberts pour arêtes horizontales, modifié pour avoir une taille `_a` x `_a`.

- `void defRobertsV(unsigned int _a);`

  Opérateur de Roberts pour arêtes verticales, modifié pour avoir une taille `_a` x `_a`.
- `void defPrewittH();`

  Définir l'objet courant comme l'opérateur de Prewitt pour arêtes horizontales.
- `void defPrewittV();`

  Définir l'objet courant comme l'opérateur de Prewitt pour arêtes verticales.
- `void defGaussienne(unsigned int _a=5);`

  Opérateur pour un filtre gaussien sur un voisinage `_a` x `_a`. La variance est `_a`/5.
- `void defGaussienneH(unsigned int _a=5);`

  Opérateur pour un filtre gaussien sur un voisinage `1` x `_a`. La variance est `_a`/5.
- `void defGaussienneV(unsigned int _a=5);`

  Opérateur pour un filtre gaussien sur un voisinage `_a` x `1`. La variance est `_a`/5.
- `void defMoyenne(unsigned int _a=5);`

  Opérateur pour le calcul de la moyenne sur un voisinage `_a` x `_a`.
- `void defLaplacien();`

  Définir l'objet courant comme l'opérateur pour le calcul du Laplacien.
- `const int Facteur() const;`

  Retourne le facteur de division de l'opérateur courant. N'affecte pas l'objet courant.
- `OperateurType Type() const;`

  Retourne le type de l'opérateur courant. N'affecte pas l'objet courant.
- `const Point Origine() const;`

  Retourne le point correspondant à l'origine de l'opérateur. L'objet courant n'est pas affecté par cette fonction.
- `void Facteur(int _facteur);`

  Définir le facteur de division de l'opérateur à `_facteur`. Lance une exception si ce facteur est nul.
- `void Origine(Point _org);`

  Définir le point d'origine de l'opérateur à `_org`. Lance une exception si ce point n'est pas une coordonnée valide du tableau en deux dimension de l'objet courant.
- `int ValiderPosition() const;`

  Déterminer si les valeurs de position de l'origine (`xg`, `xd`, `yh`, `yb`) sont valides. L'objet courant n'est pas affecté par cette fonction.
- `void CalculerPosition();`

  Calculer la position de l'origine et écrire les résultats dans les variables membres (`xg`, `xd`, `yh`, `yb`).

A.12 Kernel: élément structurant

La classe `Kernel` permet de représenter et de gérer des éléments structurants. Elle est dérivée publiquement de `Operateur`. La classe possède aussi une énumération de types définis d'éléments structurants et une énumérations des valeurs possibles pour les pixels binaires. Elle possède également une variable membre donnant le type de l'élément structurant.

A.12.1 Constructeurs

- `Kernel(KernelType _type=T);`
  Construire un élément structurant de type `_type`.
- `Kernel(KernelType _type, unsigned int _d);`
  Construire un élément structurant de type `_type`, possédant `_d` lignes et `_d` colonnes.
- `Kernel(KernelType _type, unsigned int l, unsigned int c);`
  Construire un élément structurant de type `_type`, possédant `l` lignes et `c` colonnes.

A.12.2 Opérateurs

La classe `Kernel` ne définit pas d'opérateurs.

A.12.3 Fonctions d'interface

- `void defType(KernelType _type);`
  Définir le type `_type` pour l'élément structurant courant.
- `void defType(KernelType _type, unsigned int _a);`
  Construire un élément structurant de type `_type`, possédant `_a` lignes et `_a` colonnes.
- `void defType(KernelType _type, int _l, int _c);`
  Définir le type `_type` pour l'élément structurant courant et lui donner `_l` lignes et `_c` colonnes.
- `KernelType Type() const;`
  Retourne le type de l'objet courant. L'objet courant n'est pas affecté par cette fonction.
- `void defT(unsigned int _l=2, unsigned int _c=3);`
  Définir le type T à `_l` lignes et `_c` colonnes pour l'objet courant.
- `void defCroix(unsigned int _a=3);`
  Définir le type Croix de taille `_a` x `_a` pour l'objet courant.
- `void defCarre(unsigned int _a=3);`
  Définir l'objet courant comme un carré de taille `_a` x `_a`.

A.13 Ensemble: une liste d'intervalles

La classe `Ensemble` permet de représenter et de gérer un sous-ensemble des nombres entiers. Elle est dérivée publiquement de `Liste<Intervalle>`.

La classe `Ensemble` faut aussi appel à la classe `Intervalle`.

A.13.1 Constructeurs

- `Ensemble();`

  Construire un ensemble vide.
- `Ensemble(long _min);`

  Construire un ensemble contenant les nombre entiers supérieurs ou égaux à `_min`.

A.13.2 Opérateurs

La classe `Ensemble` ne définit pas d'opérateur.

A.13.3 Fonctions d'interface

- `int Appartient(const long &nb) const;`

  Détermine si l'élément `nb` appartient ou non à l'ensemble courant. L'objet courant n'est pas affecté par cette fonction.

A.14 Intervalle: un ensemble contigu des entiers

La classe `Intervalle` permet de représenter et de gérer un sous-ensemble contigu des nombres entiers. Elle possède deux variables accessibles publiquement de type `long` conservant les valeurs limites du sous-ensemble.

A.14.1 Constructeurs

- `Intervalle(long _min=0);`

  Construire un ensemble contenant les nombre entiers supérieurs ou égaux à `_min`.

- `Intervalle(long _min, long _max);`

  Construire un ensemble contenant les nombre entiers supérieurs ou égaux à `_min` et inférieurs ou égaux à `_max`.

A.14.2 Opérateurs

La classe `Intervalle` ne définit pas d'opérateur.

A.14.3 Fonctions d'interface

- `void PlusPetit(long _max);`

  Définit l'intervalle courant comme étant les nombre entiers inférieurs ou égaux à `_max`.

- `void PlusGrand(long _min);`

  Définit l'intervalle courant comme étant les nombre entiers supérieurs ou égaux à `_min`.

- `void Entre(long _min, long _max);`

  Définit l'intervalle courant comme étant les nombre entiers supérieurs ou égaux à `_min` et inférieurs ou égaux à `_max`.

- `void Est(long _min);`

  Définit l'intervalle courant comme étant les entiers égaux à `_min`.

- `int Appartient(const long &nb);`

  Détermine si l'élément `nb` appartient ou non à l'intervalle courant. L'objet courant n'est pas affecté par cette fonction.

A.15 ListeTri: liste triée

Le patron `ListeTri` permet de gérer une `Liste` qui est conservée triée en ordre croissant de ses éléments. Le patron est dérivé publiquement de `Liste<T>`.

A.15.1 Constructeurs

Le patron `ListeTri` ne définit pas de constructeur.

A.15.2 Opérateurs

Le patron `ListeTri` ne définit pas d'opérateur.

A.15.3 Fonctions d'interface

- `void Trier();`
  Effectuer le tri par insertion sur les éléments de la liste.
- `void Inserer(const T &_e);`
  Insérer l'élément `_e` et maintenir la liste triée.

A.16 Equivalence: table d'équivalence

Le patron `Equivalence` permet de gérer une table d'équivalence dans une table à adressage dispersé. La table est composée d'alias entre un élément effectif et l'élément correspondant désiré. Le patron est dérivé publiquement de `TableHO<IndiceAlias<T> >`.

Le patron `Equivalence` faut aussi appel au patron de classe `IndiceAlias`.

A.16.1 Constructeurs

Le patron `Equivalence` ne définit pas de constructeur.

A.16.2 Opérateurs

Le patron `Equivalence` ne définit pas d'opérateur.

A.16.3 Fonctions d'interface

- `void Inserer(const T &effectif, const T &desire);`

  Insérer une équivalence dans l'objet courant. L'objet `effectif` est la valeur courante et l'objet `desire` est la valeur équivalente désirée pour `effectif`.

- `T Chercher(const T &effectif);`

  Cherche l'équivalent désiré pour l'objet `effectif`. Lance une exception si `effectif` n'est pas dans la table. La fonction retourne la valeur équivalente.

A.17 IndiceAlias: équivalence entre deux éléments

Le patron `IndiceAlias` permet de gérer une équivalence entre deux objets de classes quelconques (classe `T`). Deux variables membres de type `T` sont accessibles publiquement, soit `eff` et `des`, les deux éléments équivalents.

A.17.1 Constructeurs

- `IndiceAlias();`

  Construire une équivalence entre deux éléments à spécifier plus tard.

- `IndiceAlias(const T &_e);`

  Construire une équivalence entre `_e` et un élément à spécifier plus tard.

- `IndiceAlias(const T &_e, const T &_d);`

  Construire une équivalence entre `_e` et `_d`.

A.17.2 Opérateurs

- `int operator==(const IndiceAlias &c) const`

  Retourne la valeur logique à savoir si l'objet courant est équivalent à l'objet `c`. Il y a équivalence lorsque les variables membres `eff` des deux objets ont la même valeur. L'objet courant n'est pas affecté par cet opérateur.

- `friend ostream &operator<<(ostream &out,const IndiceAlias<T> &ia);`

  Insère l'équivalence `ia` dans le flot de sortie `out`. L'objet courant n'est pas affecté par cet opérateur.

- `friend istream &operator>>(istream &in,IndiceAlias<T> &ia);`

  Extrait l'équivalence `ia` du flot d'entrée `in`.

A.18 ImageStream: lecture et écriture d'images sur le disque

La classe `ImageStream` permet de gérer l'accès aux fichiers image. Elle est dérivée publiquement de `fstream`. Elle possède une énumération des formats d'image supportés et une variable membre indiquant le format d'image lu par l'objet courant. La classe possède une variable membre de type `streampos` indiquant la position dans le fichier au moment du début de la lecture, une variable de type `int` indiquant le mode d'accès et une variable de type `int` conservant la valeur logique de l'affichage des informations de déverminage.

La classe `ImageStream` faut aussi appel aux classes `ErreurFormat`, `ErreurImage`, `ErreurFormatImage`, `ErreurFormatImageLecture`, `ErreurFormatImageEcriture` qui sont dérivées de la classe `Exception`. Ces classes ne définissent que leur constructeur qui donne une valeur à leur variable membre `message`.

`ImageStream` fait aussi appel aux classes `Contour`, `ImageGris, ImageRGB`, et `BitmapImage` (Colosseum Builders) et à la classe `Chaine`.

`ImageStream` fait aussi appel aux classes `FormatSunRaster`, `FormatBMP`, `FormatPCX`, `FormatXWD`, `FormatPGM`, `FormatPBM`, `FormatPPM`, `FormatContour`, `FormatXPM`. Ces classes sont toutes dérivées de `FormatImage`, la seule décrite dans le présent document. `ImageStream` fait également appel aux classes de Colosseum Builders `JpegDecoder`, `JpegEncoder`, `GifDecoder`, `GifEncoder`, `PngDecoder`, `PngEncoder`, `XbmDecoder`, `XbmEncoder.`

Pour assurer le bon fonctionnement, on doit voir à ce que la définition de `BIGENDIAN_SYSTEM` (Systeme.hpp) soit la bonne avant de compiler le code pour l'accès aux fichiers. La valeur doit être 0 pour un système Little-Endian (Intel) et 1 pour un système Big-Endian (SUN, Motorola).

A.18.1 Constructeurs

- `ImageStream(Chaine _ndf, int _mode, imgFormat _format=NbFormats);`

  Construire un flot pour accéder au fichier `_ndf` en mode d'accès `_mode`, avec le format de fichier `_format`.

A.18.2 Opérateurs

- `ImageStream &operator<<(const ImageGris &img);`

  Ecrire l'image en niveaux de gris `img` dans le flot courant.

- `ImageStream &operator>>(ImageGris &img);`

  Lire l'image du flot courant et l'écrire dans l'objet image en niveaux de gris `img`.

- `ImageStream &operator<<(const ImageRGB &img);`

  Ecrire l'image en couleur `img` dans le flot courant.

- `ImageStream &operator>>(ImageRGB &img);`

  Lire l'image du flot courant et l'écrire dans l'objet image en couleur `img`.
- `ImageStream &operator<<(const Contour &ctr);`

  Ecrire le contour `ctr` dans le flot courant.
- `ImageStream &operator>>(Contour &ctr);`

  Lire l'image du flot courant et l'écrire dans l'objet contour `ctr`.

A.18.3 Fonctions d'interface

- `int GetVerbose() const;`

  Retourner la valeur logique de l'affichage des informations de déverminage. L'objet courant n'est pas affecté par cette fonction.
- `void SetVerbose(int _v=1);`

  Donner la valeur logique `_v` à l'affichage des informations de déverminage.

A.19 FormatImage: coder et décoder un format de fichier image

La classe `FormatImage` est une classe virtuelle pure permettant de gérer le codage et le décodage des fichiers image d'un type particulier. Elle possède une variable membre indiquant si l'objet courant doit afficher des informations lors de la manipulation des fichiers (pour le déverminage).

La classe `FormatImage` fait appel aux classe `ImageRGB` et aux classes `ErreurFormat`, `ErreurImage`.

A.19.1 Constructeurs

- `FormatImage();`
  Créer un encodeur/décodeur qui n'affiche pas d'information de déverminage.
- `virtual ~FormatImage();`
  Destructeur.

A.19.2 Opérateurs

La classe `FormatImage` ne définit aucun opérateur.

A.19.3 Fonctions d'interface

- `virtual void ReadImage(istream &, ImageRGB &) = 0;`
  Décoder l'image du flot d'entrée et l'écrire dans une image en couleur.
- `virtual void WriteImage(ostream &, const ImageRGB &) = 0;`
  Coder une image en couleur et l'écrire dans le flot de sortie.
- `void SetVerbose(int _v);`
  Donner la valeur logique `_v` à l'affichage des informations de déverminage.
- `int GetVerbose() const;`
  Retourner la valeur logique de l'affichage des informations de déverminage. L'objet courant n'est pas affecté par cette fonction.

Annexe B Contours: images de test et résultats

CMU/VASC Image Database: http://www.ius.cs.cmu.edu/idb/
INRIA-Syntim: http://www-syntim.inria.fr/syntim/analyse/images-eng.html
HIPR Image Library: http://www.hig.no/bibliotek/hipr/html/library.htm

B.1 Images, contours de références et contours résultants

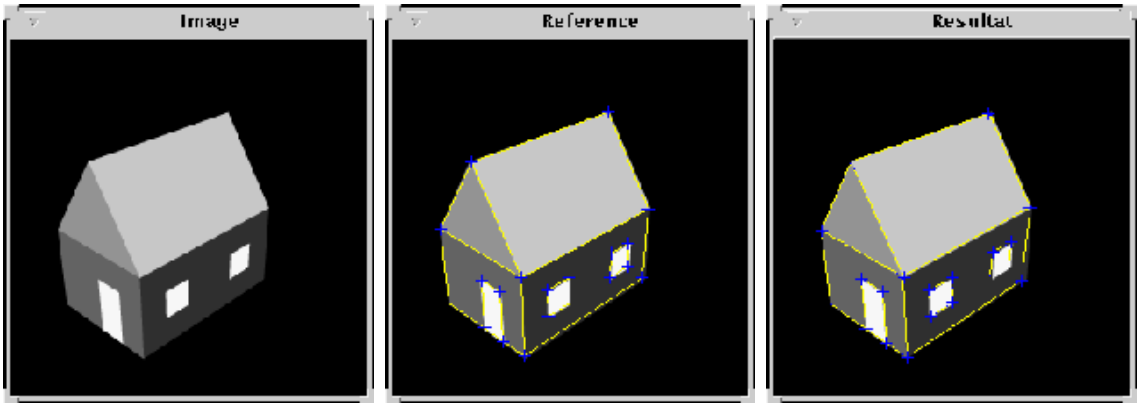

FIGURE 10. Image `house`, référence et résultat. (CMU/VASC Image Database)

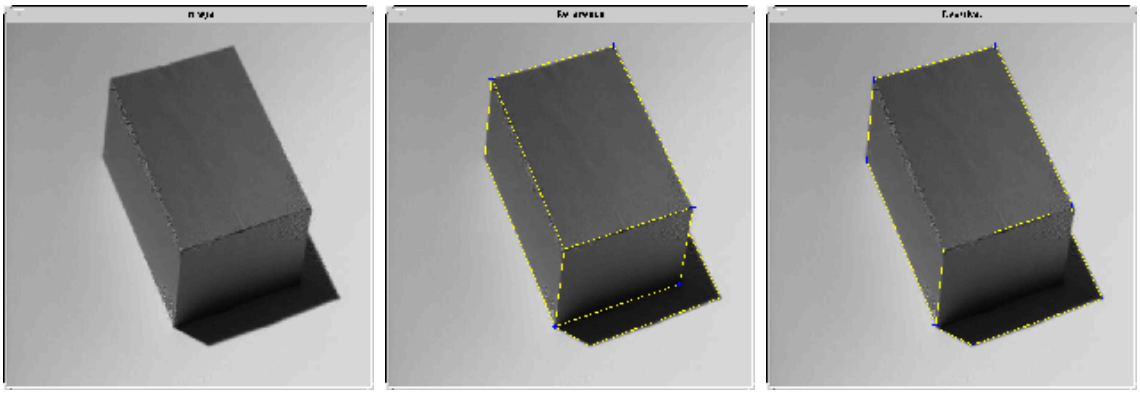

FIGURE 11. Image `Ombre0_OG4`, référence et résultat. (INRIA-Syntim ©)

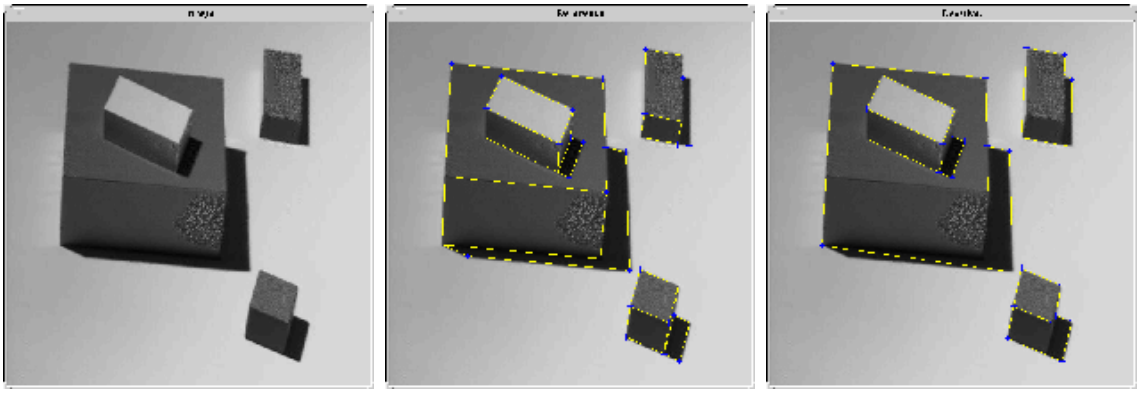

FIGURE 12. Image `Ombre0_OG5`, référence et résultat. (INRIA-Syntim ©)

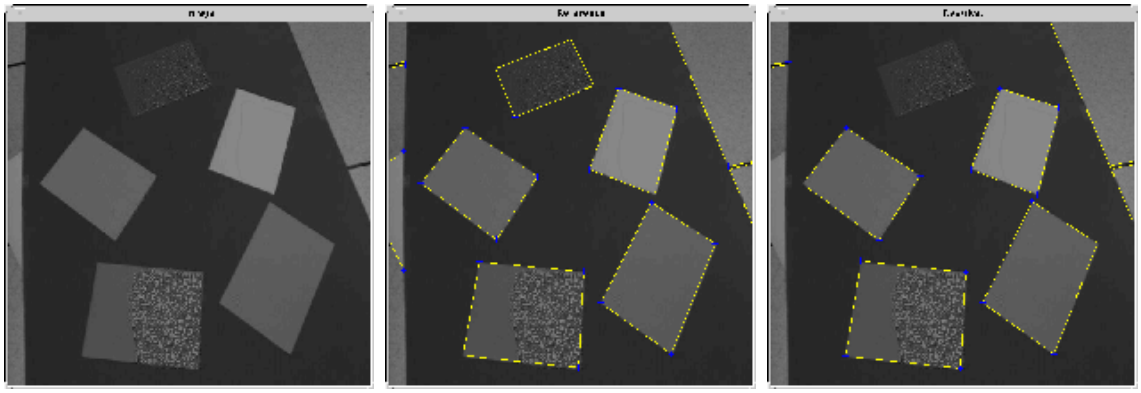

FIGURE 13. Image `Paper0_OG0`, référence et résultat. (INRIA-Syntim ©)

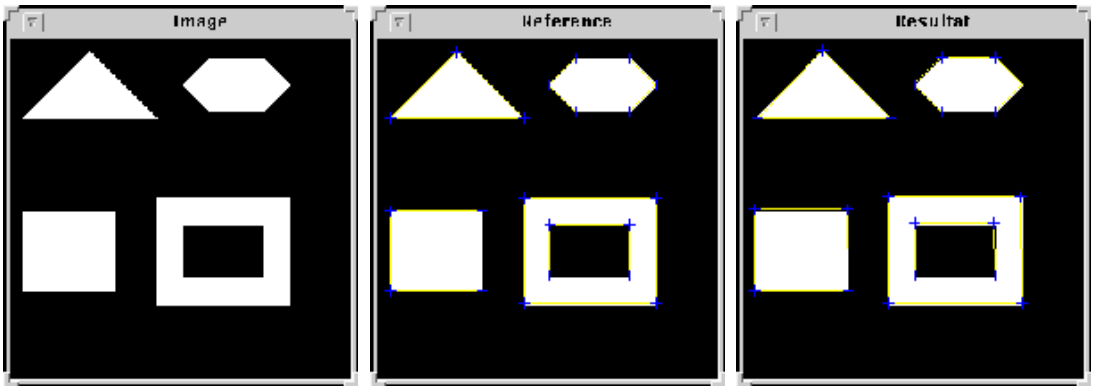

FIGURE 14. Image `rlf1`, référence et résultat. (HIPR Image Library)

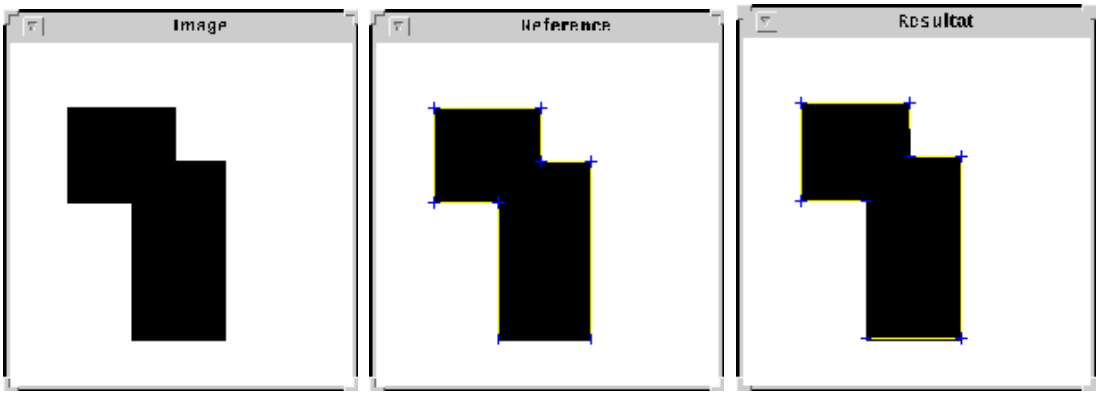

FIGURE 15. Image `sqr1`, référence et résultat. (HIPR Image Library)

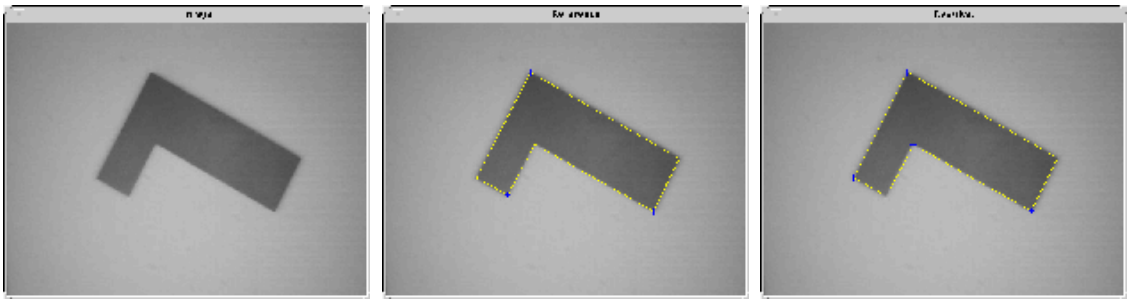

FIGURE 16. Image `wdg4`, référence et résultat. (HIPR Image Library)

B.2 Résultats numériques pour chaque image de test

TABLEAU 6. Identification des segments et des intersection, paramètres pour `house`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
house	0	3	23	7	10	6	0.07	59%	89%	65%
	5							51%	67%	54%
	10							26%	52%	31%
		2						55%	82%	60%
		6						8%	17%	10%
			50					60%	83%	65%
			100					59%	86%	64%
				0				22%	72%	32%
				22				65%	88%	70%
				45				54%	72%	57%
					20			59%	65%	60%
					40			59%	50%	57%
						0		5%	56%	15%
						10		59%	69%	61%
							0	37%	87%	47%
							0.25	47%	65%	50%
							1	37%	56%	41%

TABLEAU 7. Identification des segments et des intersection, paramètres pour `Ombre0_OG4`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
Ombre0_OG4	5	3	40	24	10	13	0.5	55%	33%	50%
	0							3%	26%	8%
	10							27%	44%	31%
		2						17%	35%	21%
		6						8%	21%	11%
			25					52%	32%	48%
			75					38%	22%	35%
				0				12%	12%	12%
				45				11%	14%	11%
					20			55%	33%	50%
					40			55%	35%	51%
						10		46%	33%	44%
						20		43%	33%	41%
							0.25	45%	33%	43%
							1	55%	33%	50%

TABLEAU 8. Identification des segments et des intersection, paramètres pour `Ombre0_OG5`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
Ombre0_OG5	9	3	36	12	5	10	0.1	61%	44%	58%
	5							51%	51%	51%
	15							34%	30%	33%
		2						34%	42%	35%
		6						46%	31%	43%
			25					47%	52%	48%
			50					47%	46%	47%
				0				40%	45%	41%
				67				59%	46%	56%
					10			61%	44%	58%
					40			61%	25%	54%
						20		49%	41%	48%
						40		30%	39%	31%
							0	30%	49%	34%
							0.25	59%	43%	56%
							1	59%	43%	56%

TABLEAU 9. Identification des segments et des intersection, paramètres pour `Paper0_OG0`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
Paper0_OG0	5	3	35	34	10	9	0.35	84%	77%	83%
	0							13%	18%	14%
	10							29%	25%	28%
		2						37%	25%	35%
		6						68%	27%	60%
			25					85%	76%	83%
			50					60%	59%	60%
				22				77%	77%	77%
				45				84%	73%	82%
					20			84%	79%	83%
					40			84%	70%	81%
						20		69%	74%	70%
						40		47%	55%	48%
							0	54%	64%	56%
							1	84%	77%	83%

TABLEAU 10. Identification des segments et des intersection, paramètres pour `rlf1`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
rlf1	0	2	50	10	10	5	0.1	100%	100%	100%
	5							100%	90%	98%
	15							62%	44%	59%
		6						24%	5%	20%
			0					0%	0%	0%
			25					100%	100%	100%
			100					100%	100%	100%
				0				100%	100%	100%
				22				100%	100%	100%
				45				85%	95%	87%
					30			100%	63%	93%
					40			100%	55%	91%
						0		91%	100%	93%
						20		100%	100%	100%
						40		50%	39%	48%
							0	100%	100%	100%
							0.25	83%	93%	85%
							0.5	54%	59%	55%

TABLEAU 11. Identification des segments et des intersection, paramètres pour `sqr1`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
sqr1	0	2	50	10	10	5	0.1	100%	100%	100%
	5							100%	100%	100%
	10							88%	50%	80%
		6						25%	0%	20%
			25					100%	100%	100%
			100					100%	100%	100%
				0				100%	100%	100%
				67				100%	100%	100%
				90				0%	0%	0%
					0			100%	0%	80%
					30			100%	100%	100%
					40			100%	93%	99%
						0		100%	100%	100%
						30		100%	100%	100%
						40		71%	86%	74%
							0	100%	100%	100%
							0.5	31%	46%	34%

TABLEAU 12. Identification des segments et des intersection, paramètres pour `wdg4`.

Image	f	g	t	θ	i	v	m	$r_{Segments}$	r_{Points}	r_{Tot}
wdg4	5	4	50	10	10	10	0.2	100%	100%	100%
	0							50%	29%	46%
	10							67%	67%	67%
		2						0%	0%	0%
		6						67%	50%	63%
			25					100%	100%	100%
			75					73%	80%	74%
				0				71%	100%	77%
				67				100%	100%	100%
					20			100%	100%	100%
					40			100%	100%	100%
						0		62%	100%	70%
						20		100%	100%	100%
						40		100%	100%	100%
							0	32%	55%	36%
							1	100%	100%	100%

Paramètres pour Canny: **f**: taille du filtre gaussien; **g**: taille de l'opérateur pour le calcul du gradient; **t**: seuil pour le gradient.

Paramètres pour les contours: **v**: distance maximale entre les segments à fusionner; θ: angle minimal pour intersections/angle maximal pour fusion; **i**: distance maximale entre point d'intersection et chaque segment intercepté; **m**: erreur normalisée maximale sur les segments de droite

Résultats: $\mathbf{r}_{Segments}$: pour les segments; $\mathbf{r}_{Points}$: pour les points d'intersection; $\mathbf{r}_{Tot}$: résultat final.

B.3 Tolérance au bruit pour chaque image de test

Chaque image est soumise à un bruit gaussien et trois essais sont faits à chaque valeur de variance. Le résultat attribué est la moyenne des réponses obtenues trois essais, normalisée par rapport à la réponse de l'image originale.

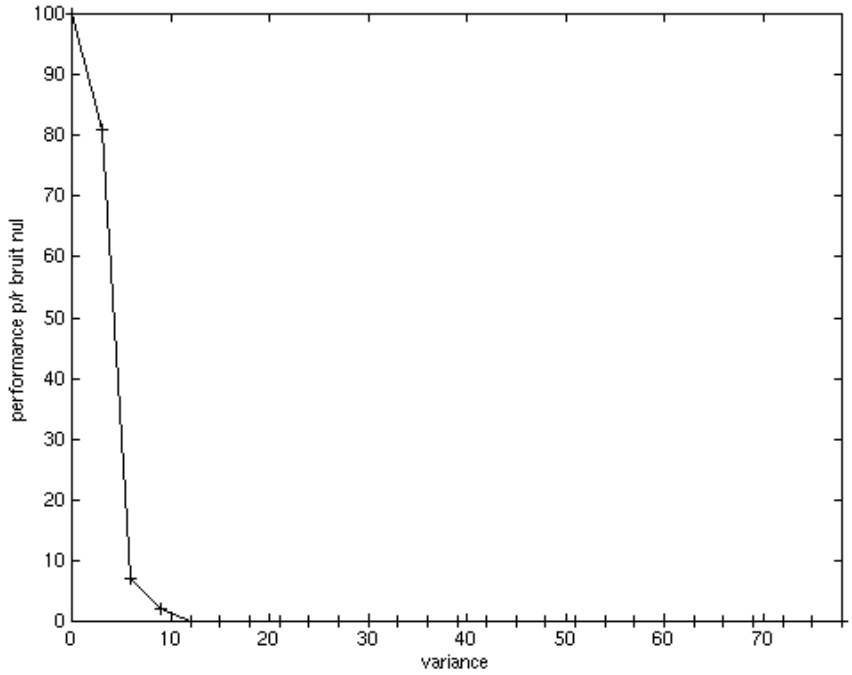

FIGURE 17. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image house soumise à un bruit gaussien dont la variance est entre 0 et 78.

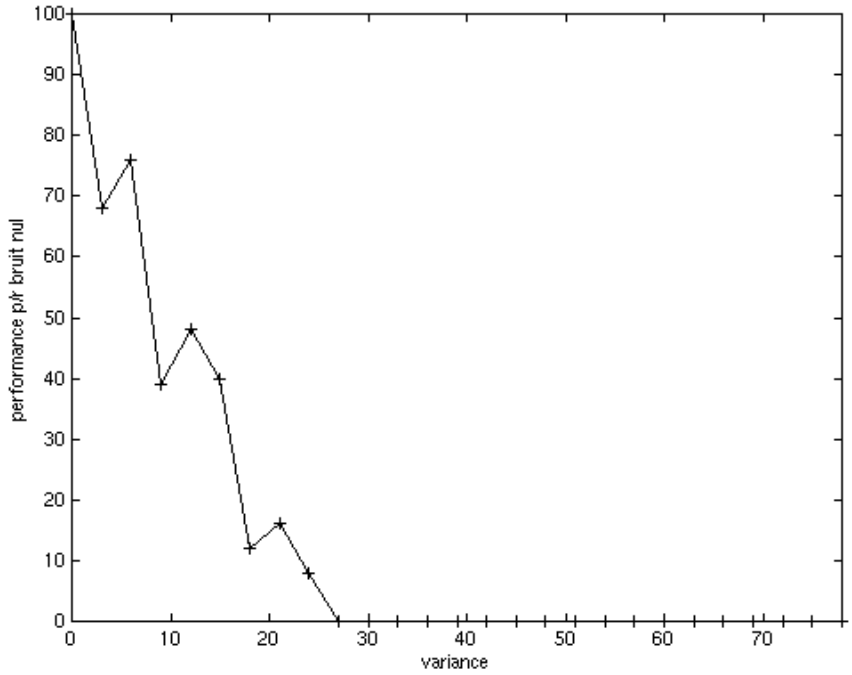

FIGURE 18. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image Ombre0_OG4 soumise à un bruit gaussien dont la variance est entre 0 et 78.

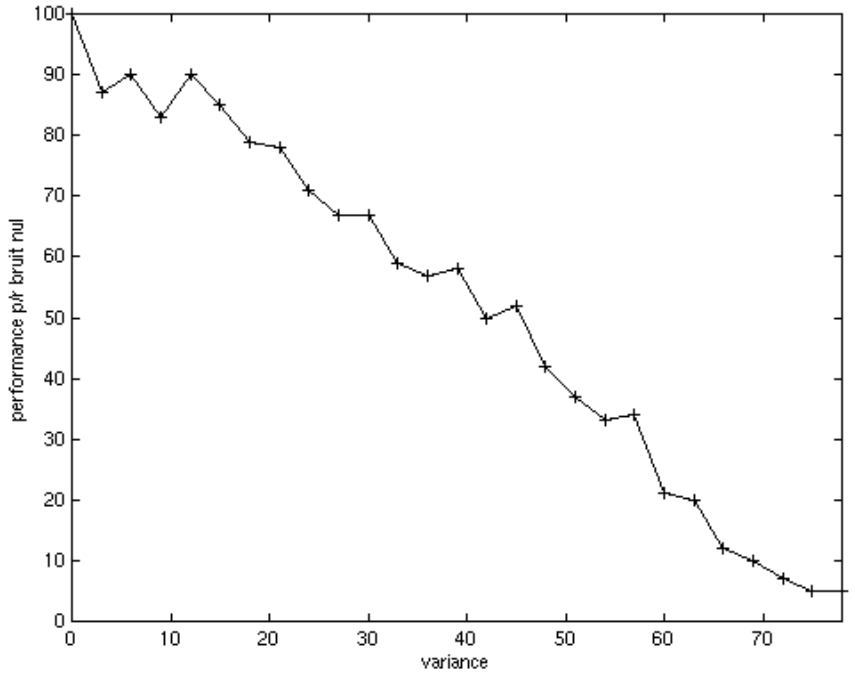

FIGURE 19. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image Ombre0_OG5 soumise à un bruit gaussien dont la variance est entre 0 et 78.

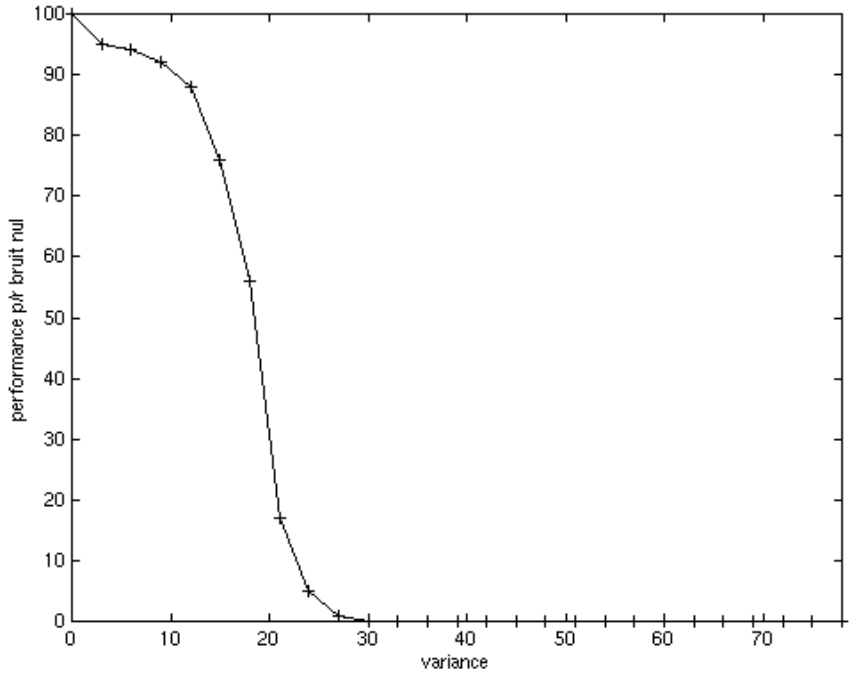

FIGURE 20. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image Paper0_OG0 soumise à un bruit gaussien dont la variance est entre 0 et 78.

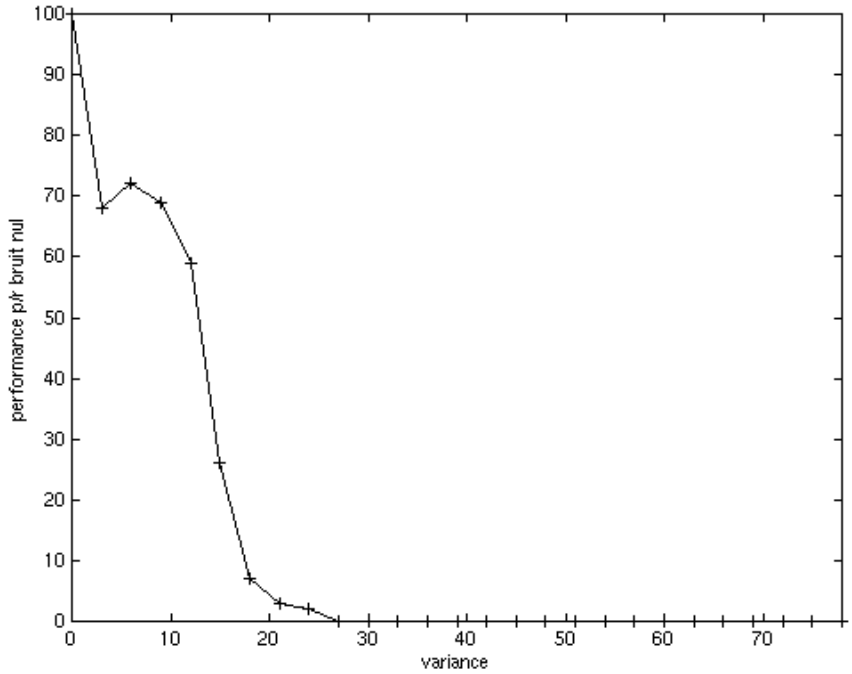

FIGURE 21. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image rlf1 soumise à un bruit gaussien dont la variance est entre 0 et 78.

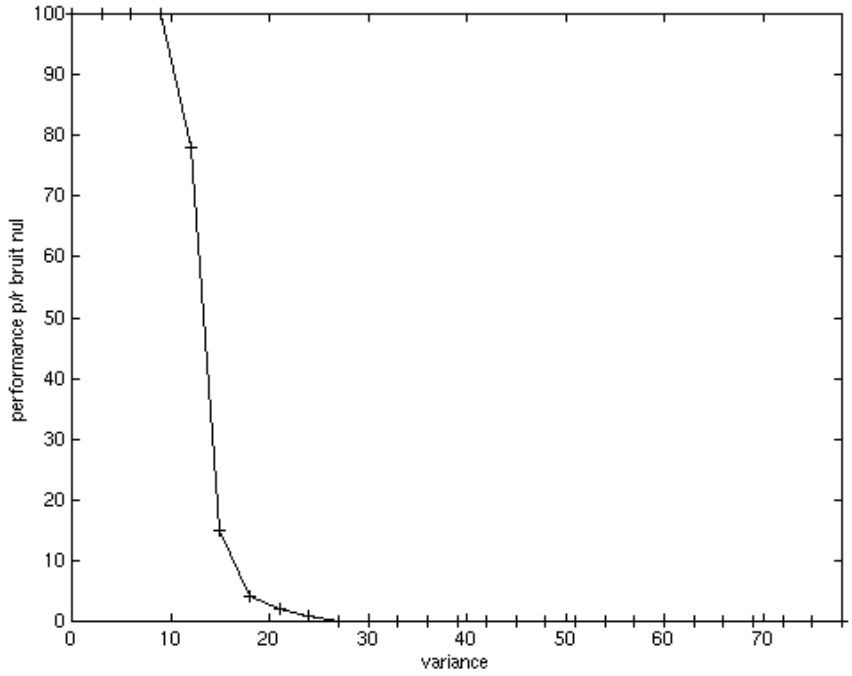

FIGURE 22. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image sqr1 soumise à un bruit gaussien dont la variance est entre 0 et 78.

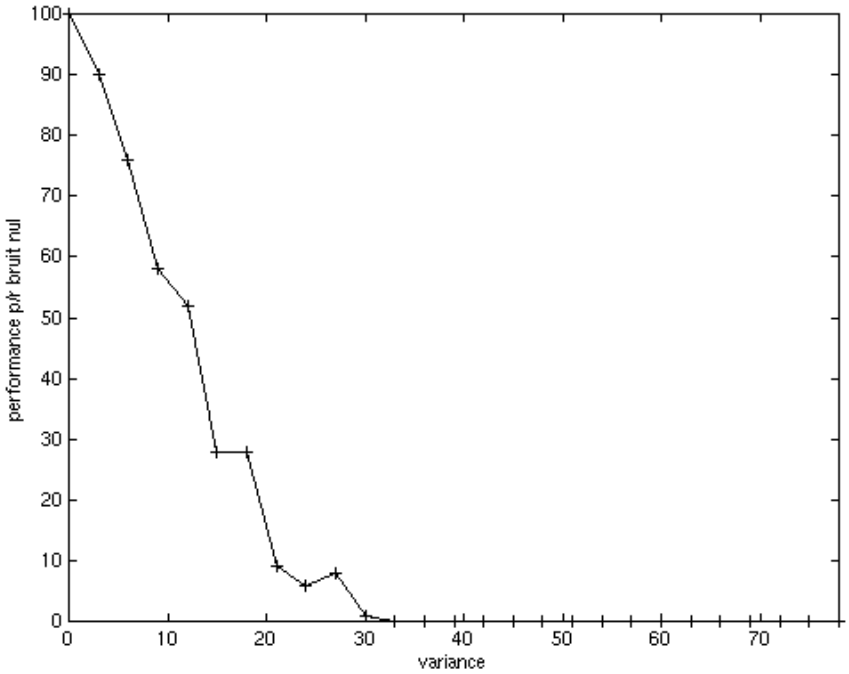

FIGURE 23. Résultat normalisé moyen obtenus avec les paramètres optimaux pour l'image wdg4 soumise à un bruit gaussien dont la variance est entre 0 et 78.

B.4 Résultats sur d'autres images

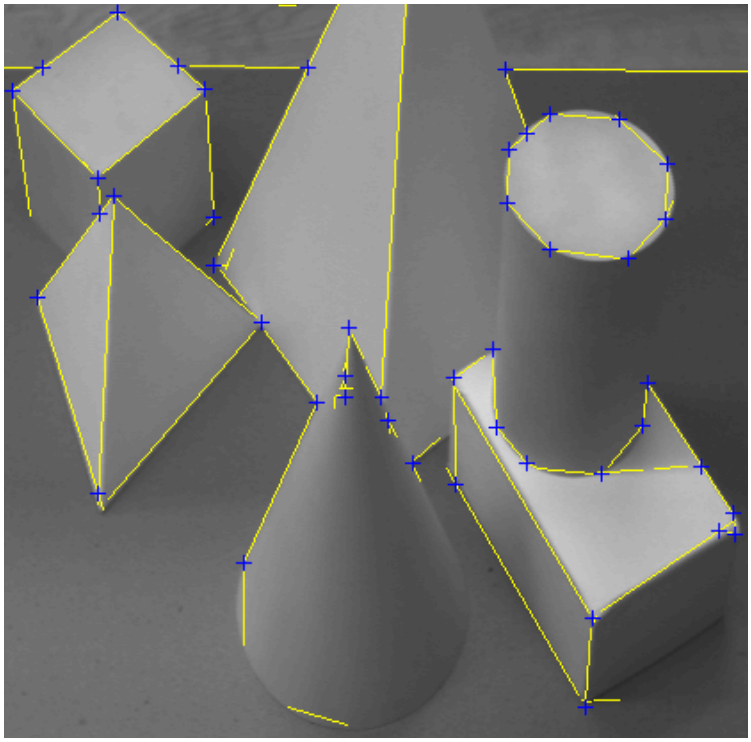

FIGURE 24. Image des 6 objets. Paramètres: f=5, g=3, t=17, θ=19, i=9, v=5, m=0.01.

FIGURE 25. Image 02_Mtl_G. Paramètres: f=9, g=3, t=35, θ=10, i=0, v=10, m=0.1.

FIGURE 26. Image 09_Bur_1. Paramètres: f=5, g=3, t=55, θ=3, i=0, v=5, m=0.1.

FIGURE 27. Image 12_Numeros. Paramètres: f=5, g=3, t=55, θ=3, i=0, v=5, m=0.1

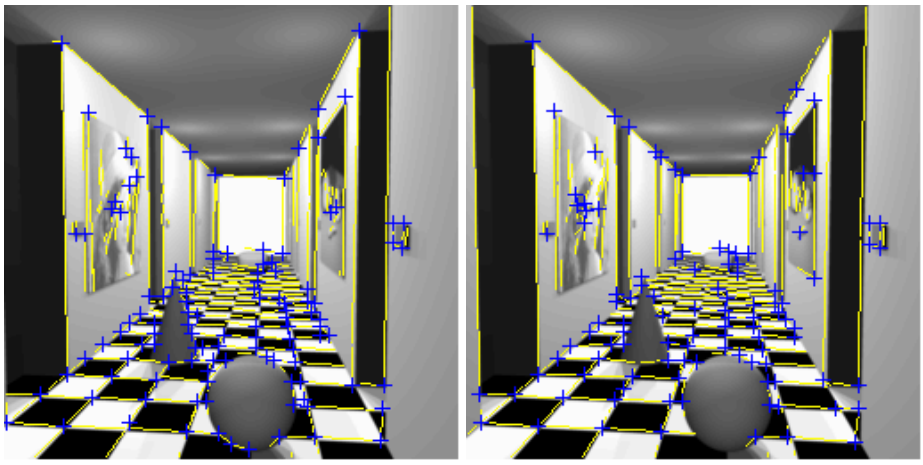

FIGURE 28. Image stéréo du corridor. Paramètres: f=0, g=2, t=80, θ=15, i=4, v=2, m=0.01.