GREYC

Equipe IMAGE

Pandore V5

Manuel de programmation

Regis Clouard
Abderrahim Elmoataz
François Angot
Alexandre Duret-Lutz
Olivier Lezoray

Rapport de recherche - août 2004 - 72 pages

GREYC - IMAGE
6, boulevard Maréchal Juin
F-14050 Caen cedex, France

Sommaire

Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen (UMRCNRS 6072)
GREYC-IMAGE, 6, boulevard du Maréchal Juin, 14050 Caen cedex (France)

Télephone : +33 (0)2 31 45 27 20 - Fax : +33 (0)2 31 45 26 98
http://www.greyc.ismra.fr/EquipeImage

REGIS.CLOUARD@GREYC.ISMRA.FR

REGIS.CLOUARD@GREYC.ISMRA.FR

1 La programmation sous Pandore

1.1 Conventions

1.1.1 Parcours des images

Dans la programmation des opérateurs, l'ordre des dimensions est:

- [profondeur][ligne][colonne]
- *Ex: image[z][y][x]*

Par contre, dans la ligne de commande des opérateurs l'ordre est:

- colonne, ligne, profondeur.
- *Ex: USAGE: valeurpixel x y z [fichier_image|-]*

1.1.2 Codage de Freeman

Le codage de Freeman attribue un numéro à chacun des voisins immédiats d'un pixel ou d'un voxel. Le codage dépend de la dimension de l'image et de la connexité choisie.

En 2D, le codage retenu est celui donné Fig. 1. Ce codage respecte ainsi les deux propriétés suivantes (cf. le parcours causal (ou balayage vidéo) d'une image):

1. Les pixels v_i pour $i \in [0;1]$ en 4-connexité (resp. $[0;3]$ en 8-connexité) sont vus avant le pixel central dans le balayage 2D de l'image et les pixels v_i pour $i \in [2;3]$ (resp. $[4;7]$) sont vus après.
2. En 4-connexité (resp. 8-connexité), le voxel v_i et le voxel v_{2+i} (resp. v_{4+i}) sont

REGIS.CLOUARD@GREYC.ISMRA.FR

symétriques par rapport au voxel central.

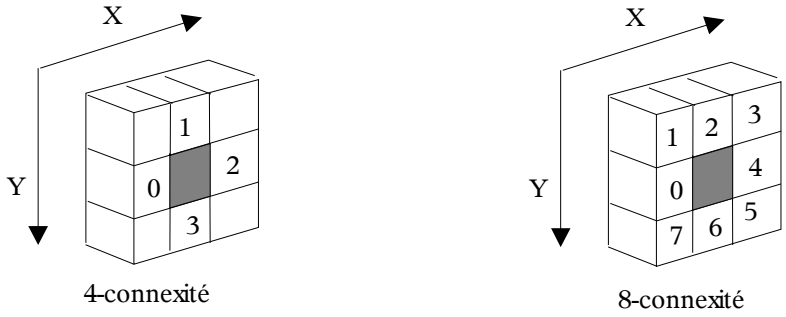

Fig. 1. Le codage de Freeman pour le 2D en 4 et 8 connexité.

En 3D, le choix du codage respecte ainsi les 3 propriétés suivantes :

1. Les voxels v_i pour $i \in [0;2]$ en 6-connexité (resp. [0;12] en 26-connexité) sont vus avant le voxel central dans un balayage 3D et les voxels v_i pour $i \in [3;5]$ (resp. [13;25]) sont vus après.
2. En 6-connexité (resp. en 26-connexité), le voxel v_i et le voxel v_{3+i} (resp. v_{13+i}) sont symétriques par rapport au voxel central.
3. En 26-connexité, pour ces deux ensembles pris séparément, les voxels de numéros consécutifs sont voisins en 6-connexité.

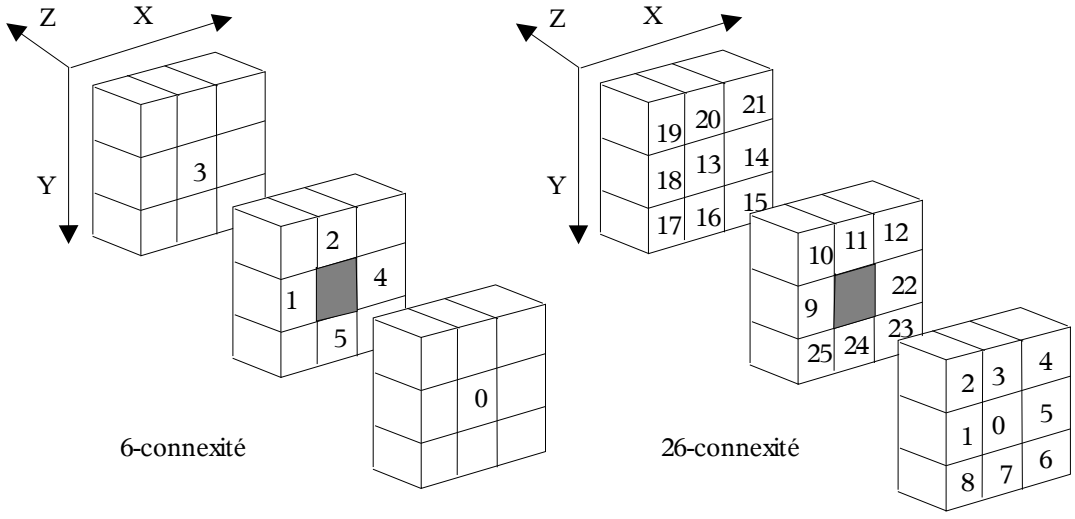

Fig. 2. Le codage de Freeman pour le 3D en 6 et 26 voisinage.

REGIS.CLOUARD@GREYC.ISMRA.FR

1.1.3 Parcours des voisins d'un point selon le codage de Freeman

L'accès aux voisins d'un pixel selon le codage de Freeman se fait à partir de tableaux dans lesquels les décalages sont prédéfinis.

- En 1D

Le tableau v2x donne le décalage en x des 2 voisins possibles: v2x[0] et v2x[1].

```
int voisx; for (v=0;v<2;v++) { voisx=v2x[v]+x; }
```

Le tableau v2 donne le décalage sous la forme d'un Point1d.

```
Point1d vois,p(10,10); for (v=0;v<2;v++) { vois=v2[v]+p; }
```

- En 2D

Les quatre tableaux v4x, v4y et v8x, v8y donnent le décalage en x et y des voisins possibles en 4 ou 8-connexité :

```
int voix, voisy;
for (v=0;v<4;v++) { voisx=v4x[v]+x; voisy=v4y[v]+y; }
for (v=0;v<8;v++) { voisx=v8x[v]+x; voisy=v8y[v]+y; }
```

Les deux tableaux v4 et v8 donnent le décalage sous la forme de Point2d :

```
Point2d vois,p(10,10);
for (v=0;v<4;v++) { vois=v4[v]+p; }
for (v=0;v<8;v++) { vois=v8[v]+p; }
```

Exemple d'utilisation :

```
ims[p] = ims[p+v8[0]]; // Affecter le pixel central à la valeur du voisin 0
```

- En 3D

Les six tableaux v6x, v6y, v6z et v26x, v26y, v26z donnent le décalage en x et y des voisins possibles en 6 ou 26-connexité :

```
int voix, voisy, voisz;
for (v=0;v<6;v++) { voisx=v6x[v]+x; voisy=v6y[v]+y; voisz=v6z[v]+z }
for (v=0;v<26;v++) { voisx=v26x[v]+x; voisy=v26y[v]+y; voisz=v6z[v]+z }
```

Les deux tableaux v6 et v26 donnent le décalage sous la forme de Point3d :

```
Point3d vois,p(10,10,10);
for (v=0;v<6;v++) { vois=v6[v]+p; }
for (v=0;v<26;v++) { vois=v26[v]+p; }
```

- Généralisé

Les tableaux préfixés vc permettent d'accéder aux voisins à partir de n'importe quelle connexité sous la forme de Point: vc[connexite][voisin], ou pour chaque coordonnée: vcx[connexite][voisin],vcy[connexite][voisin], vcz[connexite][voisin] :

```
Point2d vois,p(10,10,10); int conx=4
for (v=0;v<conx;v++) { vois=((Point2*)vc[conx])[v]+p; }
for (v=0;v<conx;v++) { voisx=vcx[conx][v]+p.x; voisy=vcy[conx][v]+p.y; }
```


REGIS.CLOUARD@GREYC.ISMRA.FR

1.2 Paradigme de programmation

L'environnement de programmation utilise le langage orienté objet C++, et bénéficie ainsi de sa représentation objet, de sa portabilité, de sa large diffusion et de l'efficacité de son code.

Les concepts manipulés en traitement d'images sont définis dans notre environnement, soit par un **Objet de base**, soit par une **combinaison d'Objets de base**. Un objet de base de l'environnement est défini par une classe abstraite C++ qui permet de cacher l'implantation des données de l'objet et présente vers l'extérieur un certain nombre de méthodes interfaces qui sont les seules façons de manipuler son contenu. Ce principe évite d'une part les modifications de valeurs des objets non contrôlées et permet d'autre part de changer la représentation interne sans que cela n'ait d'influence sur le programme qui utilise ces objets. La structure de données de chaque objet distingue les attributs de caractérisation qui décrivent les propriétés de l'objet et les données qui représentent l'information. Les objets peuvent ainsi exister avant leurs données.

Par contre, un opérateur est défini par une fonction C++ classique et non par une méthode de l'objet pour conserver le caractère opérationnel de son comportement.

1.3 Types de base

Les types de base reprennent principalement ceux du langage C, mais leur nom est construit en modifiant la première lettre du type C par une majuscule ou en le faisant précéder d'un U pour unsigned. Cette redéfinition des types de bases permet de s'affranchir au maximum des spécificités de la machine et de produire un code portable. De ce fait, il n'existe pas le type `Int` puisque ce type n'est pas normalisé par le langage C. On trouve alors les types :

Char, Uchar, Short, Ushort, Long, Ulong, Float, Double.

Un type supplémentaire est introduit pour représenter le code erreur de retour des fonctions opérateurs : **Errc**. il permet de représenter des valeurs des types :

- énuméré: SUCCESS ou FAILURE ;

REGIS.CLOUARD@GREYC.ISMRA.FR

- de base (Char, Uchar, Long ...) ;

Exemple :

```
Errc a; if (...) a=5 else a=FAILURE; if (a==FAILURE) .. else ...
```

1.4 Objets de base

Il existe 5 types d'objets prédéfinis dans Pandore :

1. Les **points**;
2. Les **dimensions**;
3. Les **images**;
4. Les **régions**;
5. Les **graphes**;

Les objets sont caractérisés par des **attributs** qui définissent leur représentation interne et par des **méthodes** qui définissent leur comportement. Les méthodes participent du caractère convivial de l'environnement car elles permettent la manipulation des objets indépendamment de la façon dont ils sont implantés physiquement. Chaque objet est responsable de la gestion de ses données et du maintien de la cohérence entre les valeurs de ses attributs et de ses données. Les méthodes agissent uniquement sur l'objet appelant.

1.5 Objets composites

Outre les types classiques du C++ (tableaux, map, ..), il existe deux types d'objets composites définis par Pandore :

1. Les **collections**;
2. Les **files** d'objets.

Certains objets peuvent être composés avec plusieurs objets de base. Par exemple, une collection peut contenir n'importe quel type de base ou d'objet de base (e,g., une collection peut contenir une collection). On peut aussi former une file de Points, des graphes de régions...

REGIS.CLOUARD@GREYC.ISMRA.FR

2 Les objets prédéfinis

2.1 Les objets Pandore

Tous les objets Pandore héritent de la super-classe abstraite Pobject (Fig. 3).

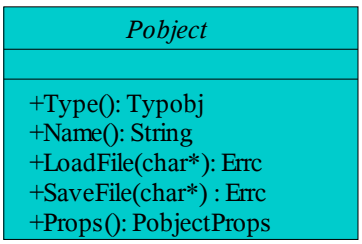

Fig. 3. La classe abstraite Pobject

2.1.1 Les types

Chaque objet Pandore est identifié par un numéro et un nom. Le fichier panfile.h détient la liste énumérée de tous les numéros d'identification des types Pandore sous la forme d'identificateurs. Un identificateur est construit avec le nom de la classe de l'objet précédé de Po_ :

```
Po_Img2duc (une image 2d de niveaux de gris en octets), Po_Point2d, etc.
```

Le numéro et le nom d'un objet sont accessibles par les méthodes:

- `Typobj Type()`: retourne le numéro d'identification d'un objet ;
- `String Name()`: retourne le nom de l'objet.

Ce numéro permet ainsi de reconnaître le type d'un objet quelconque:

```
Pobject *p= new Img2duc;
if (p == Po_img2duc) std::cout << p->Name() << std::endl;
```


REGIS.CLOUARD@GREYC.ISMRA.FR

2.1.2 *Les attributs*

Chaque type d'objet possède ses propres attributs accessibles par des méthodes accesseurs. Une structure PobjectProps permet de regrouper tous les attributs échangeables entre objets comme la dimension des images, l'espace couleur, ... Cette structure est intéressante pour la création d'un objet à partir des caractéristiques d'un autre objet sans avoir à rechercher tous les attributs à renseigner. La structure est définie comme suit:

```
struct PobjectProps{
   Long nbands;              // Nombre de bandes.
   Long ncol;                // Nombre de colonnes.
   Long nrow;                // Nombre de lignes.
   Long ndep;                // Nombre de plans.
   PColorSpace colorspace;   // Pour les images couleur: Espace couleur.
   Ulong nlabels;            // Pour les regions: nombre de labels.
   Long size;                // Pour les graphes: nombre de sommets
};
```

La structure d'un objet est accessible par la méthode:

- struct PobjectProps Props() : retourne la liste des valeurs d'attribut de l'objet.

Exemples d'utilisation :

```
Img2duc ims1(40,125);
Reg2d rgs(ims1.Props());              // Reprend la taille de l'image ims.
Graph2d *g=new Graph2d(ims1.Props()); // Reprend la taille de l'image ims1.
Imc2duc ims2;
ims2.New(g.Props());                  // Reprend la taille du graphe g.
```

2.1.3 *Les méthodes*

Chaque objet possède généralement 4 catégories de méthodes :

1. Création, initialisation et destruction de la représentation interne;
2. Consultation des valeurs de données;
3. Transfert des données à partir ou vers des fichiers;
4. Éventuellement des utilitaires qui facilitent leur utilisation.

2.1.4 *Les fichiers*

La plupart des objets Pandore (sauf les dimensions et les listes) peuvent être sauvés dans un fichier normalisé (suffixé « .pan » par pure convention). Les fichiers sont construits avec un entête commun, un entête spécifique de l'objet puis les données stockées en binaire. Ceci fait que les fichiers sont dépendant de

REGIS.CLOUARD@GREYC.ISMRA.FR

l'architecture matérielle (e.g., Inter, Motorola). Il peut alors être nécessaire d'utiliser l'opérateur `pan2pan` pour convertir une image d'une architecture dans une autre.

L'entête général d'un fichier Pandore

Nom	Bits	Signification
Magic_number	10	PANDORE05
Unused	2	Complément à 32 bits
Typobj	4	Le numéro d'identifcation de l'objet.
Creator	9	Nom du créateur de l'image.
Date	10	Date de création
Unused	1	Complément à 32

La partie spécifique de l'objet

Cette partie est composée des valeurs d'attributs de l'objet.

La partie données

Les données sont stockées sans compression.

2.2 L'objet Point

Un point définit des coordonnées spatiales : (x) en 1D, (x,y) en 2D et (x,y,z) en 3D. Il existe trois types Point selon la dimension considérée 1D, 2D et 3D (Fig. 4).

2.2.1 Types

Il n'existe que trois types de points:

```
Point1d : Un point 1D.
Point2d : Un point 2D.
Point3d : Un point 3D.
```

2.2.2 Attributs

Les attributs d'un point sont :

```
Long x : L'abscisse (pour le 1D, 2D et 3D).
Long y : L'ordonnée (pour le 2D et 3D).
Long z : La profondeur (pour le 3D).
```

Ces attributs sont accessible en lecture et en écriture:

```
Point2d p; p.x=12; std::cout << p.y << std::endl;
```


REGIS.CLOUARD@GREYC.ISMRA.FR

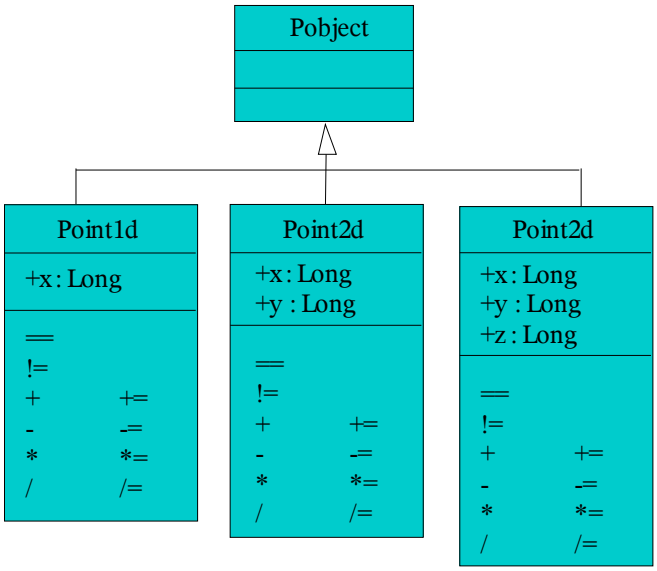

Fig. 4. Les classes de Point.

- `Typobj Type()`: l'identificateur de l'objet (*Po_Point1d, Po_Point2d, Po_Point3d*)
- `String Name()`: le nom du type de l'objet .

2.2.3 Méthodes

Création

```
Point2d p1;                          // = Point2d p1(0,0)
Point2d p2(12,24);
Point2d *p4 = new Point2d(3);        // p4->x=p4->y=3;
```

Initialisation

```
p=Point2d(12,24);
p=1; /* x=y=1 */ p1=p2;
```

Consultation

Les méthodes ne sont que les opérations classiques entre points, ou entre un point et un entier de type Long.

- == : test d'égalité.
- != : test d'inégalité.
- + ,+=: addition.
- - ,-=: soustraction.

REGIS.CLOUARD@GREYC.ISMRA.FR

- *, *=: multiplication.
- / , /=: division.

Exemples d'opérations valides:

```
Point2d p1(5,10), p2;
p1==p2; p1==1;
p1!=p2; p2!=2;
p1 + p2, p1+=p2, p1+1, p2+=2;
p1 * p2, p1*=p2, p1*1, p2*=2;
```

Transfert

Les méthodes de transferts permettent uniquement de stocker un point ou un tableau de points dans une collection :

```
Collection cold;
```

1)Écriture puis lecture d'un point dans une collection

```
Point2d p(12,12), *p2;
cold.SETPOBJECT("essai",Point2d,&p);           // Ecriture
p2=(Point2d*)cold.GETPOBJECT("essai",Point2d); // Lecture
```

2) Écriture puis lecture d'un tableau de points dans une collection

```
Point2d **p3=new Point2d*[12], **p4;
for (int i=0; i<12; i++) p3[i]=new Point2d(i,i);
cold.SETPARRAY("essai",Point2d,p3,12);          // Ecriture
p4=(Point2d**)cols.GETPARRAY("essai",Point2d);  // Lecture
std::cout << "coordonnées << p4[11]->x << "," << p4[11]->y << std::endl;
```

2.3 L'objet Dimension

Une dimension définit les côtes d'un objet en 1D, en 2D ou 3D. Il existe donc trois types de dimension pour le 1D, 2D et 3D.

2.3.1 Types

Il n'existe que trois types de dimension :

```
Dimension1d : Une dimension 1D.
Dimension2d : Une dimension 2D.
Dimension3d : Une dimension 3D.
```

2.3.2 Attributs

Les attributs d'une dimension sont classiquement:

```
Long w : La largeur (pour les dimensions 1D, 2D et 3D).
Long h : La hauteur (pour les dimensions 2D et 3D).
Long d : La profondeur (pour la dimension 3D).
```

- Ces attributs sont accessibles en lecture et en écriture:

```
Dimension2d d; d.w=12; std::cout << d.h << std::endl;
```


REGIS.CLOUARD@GREYC.ISMRA.FR

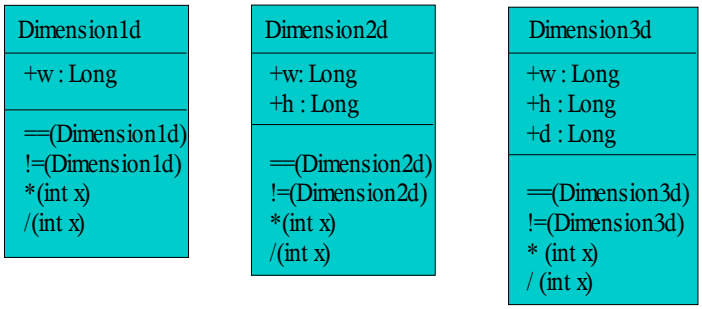

Fig. 5. Les classes de Dimension.

2.3.3 Méthodes

Création

```
Dimension2d d2(12,24);
Dimension2d d1;                 // = Dimension2d(0,0)
Dimension2d *d3=new Dimension2d(45,256);
```

Initialisation

```
Dimension2d d1;                 // = Dimension2d(0,0)
d1.w=12; d1.h=21;
```

Consultation

Les méthodes sont réduites aux opérations de comparaison entre dimensions, et aux opérations *et / avec un entier.

- == : test d'égalité.
- != : test d'inégalité.
- *nombre : multiplication de toutes les dimensions par le nombre.
- / nombre : division de toutes les dimensions par le nombre.

Exemple:

```
Dimension d1(12,13); *d2;
d2 = new Dimension(10,10);
if (d1==d2) d1=d2*5;
```

Transfert

Les dimensions ne peuvent être sauvegardées ni dans un fichier ni dans une collection.

REGIS.CLOUARD@GREYC.ISMRA.FR

2.4 L'objet Image

Une image est représentée par une matrice de pixels. Un pixel code une valeur ou un vecteur de valeurs à des coordonnées données dans la matrice. Les types d'images dépendent du type pixel et de la dimension de l'image.

2.4.1 Types

Les images Pandore peuvent être de 9 types (Fig. 6). Le nom d'une classe d'image est construit par concaténation de ses propriétés autour du préfixe **Im** :

```
Im+[x,g,c]+[1d,2d,3d]+[uc,sl,sf]
```

avec les conventions suivantes:

1. Le spectre : [x,g,c]
   - g : niveau de gris (= image multispectrale à une bande) ;
   - c : couleur. (= image multispectrale à 3 bandes) ;
   - x : multispectral (n bandes).
2. La dimension : [1d,2d,3d]
   - 1d ;
   - 2d ;
   - 3d.
3. Le type des données : [uc,sl,sf]
   - uc : unsigned char (8 bits, valeurs ∈ [0..256]) ,
   - sl : signed long (32 bits, valeurs ∈ [-2e+9..+2e+9]) ;
   - sf : float (valeurs ∈ [-3.40e+38..3.40e+38]) et une précision maximum de 1.4e-45.

Exemples :

```
Imx2duc : Image multispectrale, 2D, d'entiers courts (unsigned char).
Img3dsf : Image en niveaux de gris, 3D, de réels.
Imc2dsl : Image couleurs, 2D, d'entiers signés.
```

2.4.2 Hiérarchie

Il n'existe que trois types d'images selon le type des données. Tous les types d'image sont en fait des Imx3d<T>, où T peut prendre les valeurs Uchar, Slong et

REGIS.CLOUARD@GREYC.ISMRA.FR

Float. Ainsi, une image 2D ne possède qu'un seul plan et une image 1D un seul plan et une seule ligne, une image couleur possède trois bandes et une image en niveaux de gris un seule bande.

L'intérêt est de pouvoir n'écrire qu'un seul corps de fonction (avec template) pour tous les types d'images lorsque l'opération est la même (Voir paragraphe sur la généricité).

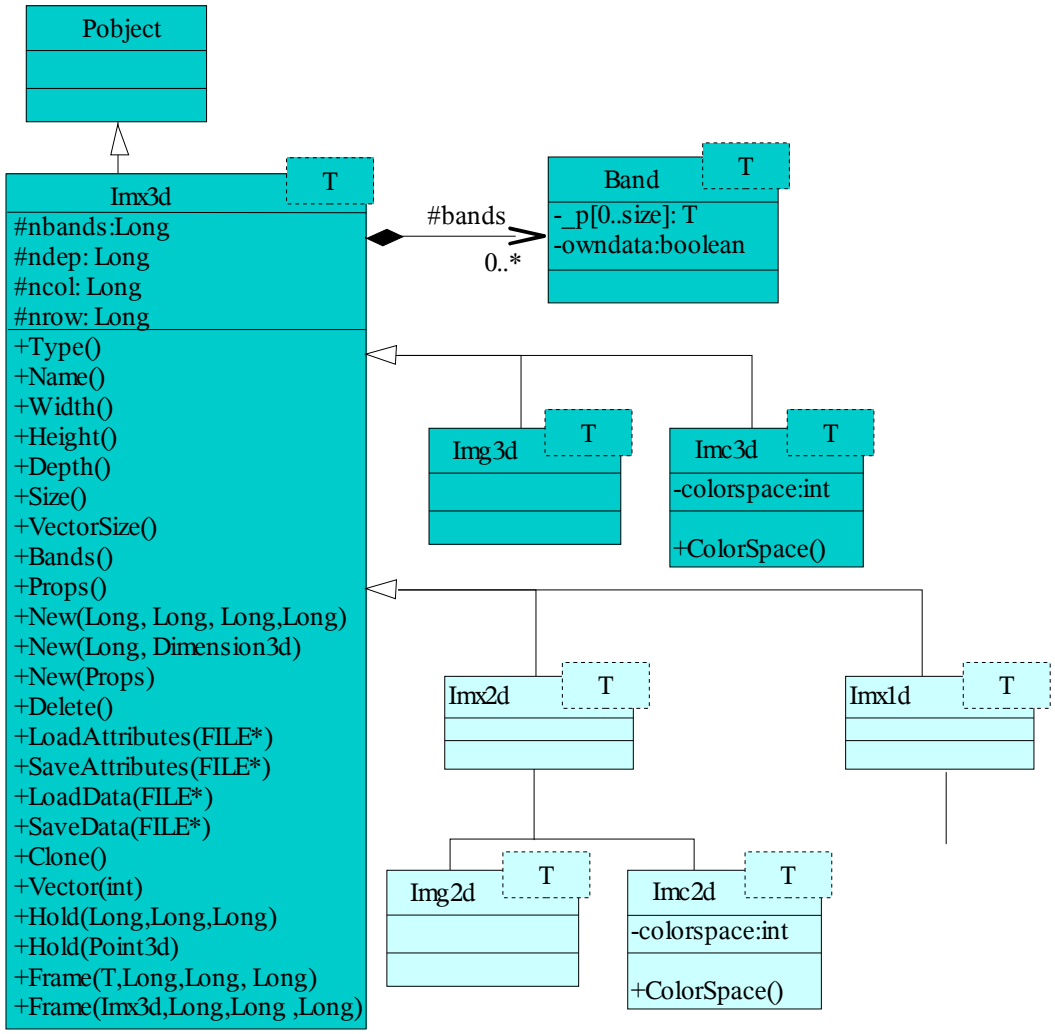

Fig. 6. Modélisation des classes Image. Seule la hiérarchie 2D est développée dans le schéma ci-dessus. T est un type primitif parmi unsigned char, signed long ou float.

REGIS.CLOUARD@GREYC.ISMRA.FR

2.4.3 Attributs

Les valeurs d'attributs sont accessibles par les méthodes suivantes :

- `Typobj Type()`: l'identificateur de l'objet (*Po_Img2duc, Po_Imc3dsf, ...*).
- `String Name()`: le nom du type de l'objet.
- `Long Width()`: le nombre de colonnes.
- `Long Height()`: le nombre de lignes.
- `Long Depth()`: le nombre de plans de l'image en 3D.
- `Dimension2d Size()`: la taille de l'image.
- `int VectorSize()`: la taille du vecteur de données - correspond aussi au nombre de pixels de l'image (voir Vector()).
- `Long Bands()`: retourne le nombre de bandes (cas particuliers : 3 pour les images couleur ; 1 pour les images niveaux de gris).
- `PColorSpace ColorSpace([x])`: pour les images couleur, retourne (si x est absent) ou affecte (si x est présent) l'espace couleur(Imc) parmi {RGB, XYZ, LUV, LAB, HSL, AST, I1I2I3, LCH, WRY, RNGNBN, YCBCR, YCH1CH2, YIQ, YUV}.
- `PobjectProps Props()`: retourne toutes les propriétés d'une image.

2.4.4 Méthodes

Création

La déclaration d'une image se fait par:

- Niveaux de gris, selon que l'on connaisse ou non la taille des donnée :

```
Img2duc ims1(256,512);      // Une image de 256 lignes et de 512 colonnes
Img2duc ims2(ims1.Size());  // Une image de même taille que ims1
Img2duc ims3(ims1.Props()); // Une image de mêmes propriétés que ims1
Img2duc ims4;               // Une image sans taille et sans donnée
Img2duc *ims5=new Img2duc(256,512); // Un pointeur sur une image de 256x512
Img2duc *ims6=new Img2duc;  // Un pointeur sur une image sans données
```

- Couleur :

```
Imc2duc ims7(256,512); im7.ColorSpace(RGB);
Imc2duc *ims8=new Img2duc(256,512);
```

- Multispectrale :

```
Imx2duc ims8(5,256,512); // Une image avec 5 bandes.
```


REGIS.CLOUARD@GREYC.ISMRA.FR

La déclaration d'une image à partir des caractéristiques d'un autre objet Pandore (Image, Carte de régions ou Graphe) se fait par:

```
Imc2duc ims1(obj.Props()); // même taille et même espace couleur que obj.
```

Initialisation

- L'allocation des données d'une image non allouée (Img2duc ims();) se fait par:

```
void New(Long nrow,Long ncol); // pour le 2d
void New(Dimension2d&);        // pour le 2d
```

Exemple:

```
Img2duc ims1; ims1.New(256,512);
        Img2duc *ims5; ims5->New(256,512);
        Img2duc ims3; ims3.New(ims1.Size());
```

Attention: l'allocation des données n'initialise pas les données à 0. Pour cela, il faut utiliser l'affectation d'une image avec une valeur, par exemple:

```
Img2duc ims(256,256);    // Image detaille 256 x 256
ims=0;                   // Affecte toutes les données à 0.
```

- L'allocation des données d'une image à partir des caractéristiques d'un autre objet Pandore (Image, Carte de régions ou Graphe) se fait par:

```
ims1.New(ims2.Props()); // Même valeur d'attributs que l'image ims2
```

- *Cas particulier des images de niveaux de gris.* L'allocation des données d'une image peut se faire à partir de données préalablement allouées **Attention, dans ce cas, la destruction des données n'est pas faite avec la destruction de l'objet.**

On retrouve les déclarations:

```
    Float *d = (Float*)malloc(256*256*sizeof(Float));
    Img2dsf ims1(256,256,d);
    Img2dsf *ims2 = new Img2dsf(256,256,d);
    ims1.delete();
    delete ims2;
```

L'intérêt est par exemple de pouvoir traiter une image multispectrale comme autant d'images en niveaux de gris. Par exemple:

```
    Imc2duc *ims= new Imc2duc(256,256);
    for(b=0;b<ims.Bands();b++){
      Img2duc *imii = new Img2duc(ims.Height,ims.Width,ims.Vector(b));
      Img2duc *imio = new Img2duc(imii->Props());
      gauss:Gauss(*imii,*imio,2.0F);
      *imii=*imio;
      delete imii;
      delete imio;
    }
```


REGIS.CLOUARD@GREYC.ISMRA.FR

En sortie, chacun des plans de l'image ims est traité comme une image 2D. L'intérêt est aussi de pouvoir traiter une image 3D comme autant de plans 2D. Par exemple:

```
Img3duc *ims= new Img3duc(12,200,256);
for(d=0;d<ims.Depth();d++){
  Img2duc *imii = new Img2duc(ims.Height,ims.Width,&ims[d][0][0]);
  Img2duc *imio = new Img2duc(imii->Props());
  gauss:Gauss(*imii,*imio,2.0F);
  *imii=*imio;
  delete imii;
  delete imio;
}
```

Copie

- Le clonage d'une image (une copie parfaite et distincte) se fait par:

```
Pobject* Clone();
```

Exemple (Remarque: Il faut caster puisque Clone() retourne un Pobject*) :

```
Img2duc *imd=(Img2duc*)ims2.Clone();
```

Destruction

- La destruction des données d'une image allouée statiquement se fait par:

```
void Delete();
```

Exemple :

```
Img2duc ims; ims3.New(512,512); ims3.Delete();
Img2duc *ims5=new Img2duc; ims5->New(512,512); ims5->Delete();
```

- La destruction d'une image allouée dynamiquement se fait par:

```
delete
```

Exemple :

```
Img2duc *ims5=new Img2duc(512,512); delete ims5;
```

Consultation

La consultation (en lecture et en écriture) des valeurs de pixels d'une image se fait de deux façons :

1. Sous la forme d'une matrice:
    - Pour les images niveaux de gris:

```
ims1[i][j]=15;
(*ims4)[i][j]=15; avec un pointeur sur une image;
Point2d pt(10,20); ims1[pt]=15; en utilisant un point.
```

    - Pour les images couleur `X[ ], Y[ ], Z[ ]`:

```
ims6.[i][j]=15; ims6.Y[i][j]=10; ims6.Z[i][j]=14;
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
(*ims7).X[i][j]=15; (*ims7).Y[i][j]=10; (*ims7).Z[i][j]=14;
Point2d pt; ims6.X[pt]=15; ims6.Y[pt]=10, ims6.Z[pt]=14;
```

- Pour les images multispectrales `[bande][ ]` (ou couleur en considérant 3 bandes) :

```
ims8[b][i][j]=15; // b: un numéro de bande
(*ims9)[b][i][j]=15;
Point2d pt; ims8[b][pt]=15;
```

2. Sous la forme d'un seul vecteur:

- Pour les images niveaux de gris: `Vector()`

```
Img2dsl ims(256,512);
Ulong *p=ims.Vector();

for (int i=0; i< ims.Size();i++)
   *p++ = 15;
```

- Pour les images couleur: `VectorX(), VectorY(), VectorZ()`

```
Imc2dsl imc(ims.Size());
Long *q=imc.VectorX();
Long *r=imc.VectorY();
Long *t=imc.VectorZ();

for (int i=0; i< imc.VectorSize();i++)
   *q++ = *r++ = *t++ = 127;
```

- Pour les images multispectrales: `Vector(bande)`

```
Imx2dsl imx(ims.Size());
for (int b=0; b<imx.Bands(); b++) {
   for (Long *ps=imx.Vector(b); ps<imx.Vector(b)+imx.VectorSize(); ps++)
        *ps= 127;
```

3. Sous forme d'un tableau de la taille de l'image

- Pour les images niveaux de gris: `X()`: Permet de récupérer les tableaux des données.

```
Img2dsl ims(ims.Size()); Img3dsf ims1(123,124,120);
Long **d=im.X();         Float ***d1=im.X();
```

- Pour les images couleur: `X(), Y(), Z()`: Permet de récupérer les tableaux 2D de chacune des composantes.

```
Imc2dsl imc(ims.Size());
Long **q=imc.X();
Long **r=imc.Y();
Long **t=imc.Z();
```

Affectation

- L'affectation des valeurs d'une image dans une autre **de même taille et**

REGIS.CLOUARD@GREYC.ISMRA.FR

préalablement allouée se fait par:

```
image1 = image2;
```

Exemple:

```
Img2duc *ims1 = new Img2duc(256,512)
Img2duc *imd1 = new Img2duc(256,512)
*ims1=*imd1;

Img2duc imd2(256,512), ims2(256,512);
imd2 = ims2;
```

Il y a recopie alors des valeurs de ims dans imd, même si ims et imd ne sont pas de même type. Dans ce dernier cas, les valeurs sont obtenues par coercition (cast) dans le respect des spécificités du langage C.

- L'affectation d'une valeur pour tous les pixels de l'image se fait par:

```
image = valeur;
```

Exemple:

```
Img2duc *imd1 = new Img2duc(256,512)
*imd1 = 127;


Img2duc imd2(256,512)
imd2 = 127;
```

Transfert

- La lecture d'une image se fait par:

```
Errc LoadFile(Char *filename);
```

Exemple:

```
ims.LoadFile("essai.pan");
```

- L'écriture d'une image se fait par:

```
Errc SaveFile(Char *filename);
```

Exemple:

```
ims.SaveFile("essai.pan");
```

Utilitaires

- Hold(): Test d'appartenance d'un point aux limites d'une image:

```
bool Hold(Long y,Long x);
bool Hold(Point2d pt);
Exemple: ims.Hold(5,10) // retourne true si le point (5,10) est dans ims.
```

- Frame(): Remplissage du bord d'une image:

```
Errc Frame(val, h, l): Avec une valeur val.
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
Errc Frame(val, h): Avec une valeur val et l=h. (en 3D l=h=p)
Errc Frame(ims, h, l): Par recopie du bord d'une autre image.
Errc Frame(ims, h): Par recopie du bord d'une autre image et l=h.
Exemple:      ims2.Frame(MAXUCHAR,5);
```

- Mask(reg)/UnMask(reg,ims): Construction d'une image par masquage et démasquage des pixels par une carte de région:

  Le masquage met à 0 les pixels masqués (i.e. de label=0 dans la carte de régions).

  Le démasquage remet les pixels masqués originaux de ims (seulement si cette image est de même type) et laisse les autres inchangés.

2.5 L'objet Carte de Régions

Une carte de régions est une image où chaque pixel code un numéro de label indiquant le numéro de la région à laquelle il appartient. **Par convention, les pixels de valeur 0 n'appartient à aucune région**. Une carte de régions est ici représentée par une image de unsigned long (Img2dul) plus un attribut supplémentaire détenant le nombre de régions de la carte (en fait le numéro de l'étiquette maximale si toutes les étiquettes ne sont pas utilisées). **Ceci permet de stoker plus de 4 milliards de régions dans une carte.** Toutes les méthodes du type image sont directement applicables à partir d'une carte de régions.

2.5.1 Types

Il n'existe que trois types de carte de régions selon la dimension (Fig. 7):

```
Reg1d : Une carte de régions 1d.
Reg2d : Une carte de régions 2d.
Reg3d : Une carte de régions 3d.
```

2.5.2 Attributs

Les valeurs d'attributs sont accessibles par :

- `Typobj Type()`: l'identificateur de l'objet *(Po_Reg1d, Po_Reg2d, Po_Reg3d)*
- `String Name()`: le nom du type de l'objet.
- `Long Width()`: le nombre de colonnes.
- `Long Height()`: le nombre de lignes.

REGIS.CLOUARD@GREYC.ISMRA.FR

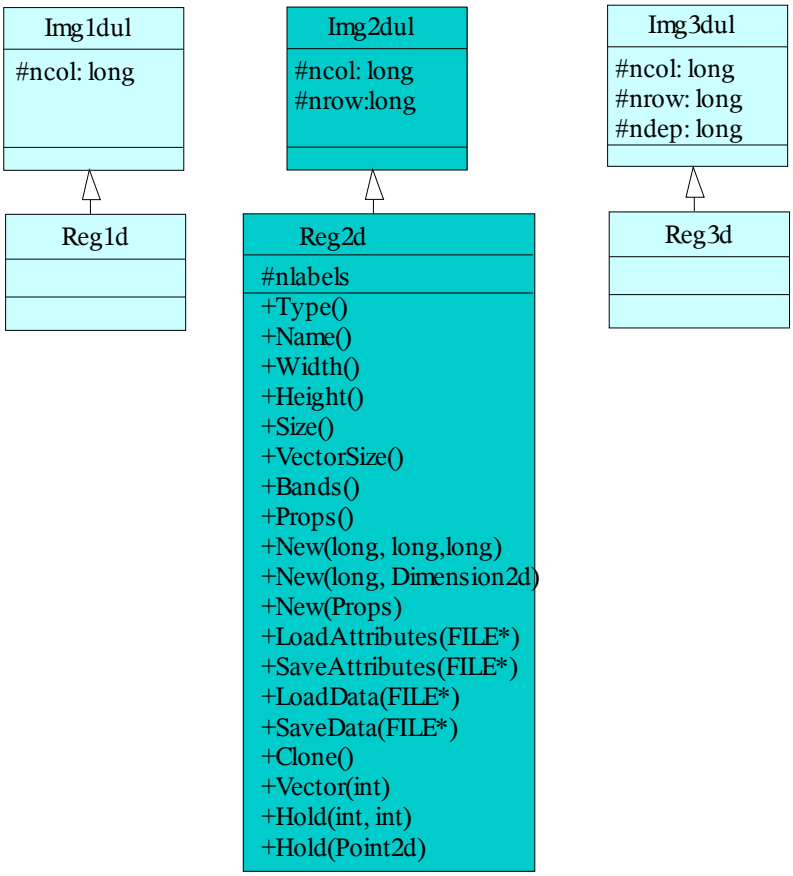

Fig. 7. Les classes région.

- `Long Depth()`: le nombre de plans de l'image en 3D.
- `Dimension2d Size()`: la taille de l'image.
- `Long Labels()`: la valeur de l'étiquette maximale.
- `Long Labels(`*`Long labels`*`)`: affecte le nombre de labels dans la carte.
- `PobjectProps Props()`: les propriétés de la carte de régions (taille, valeur de l'étiquette maximale).

2.5.3 Méthodes

Les méthodes disponibles pour la manipulation de l'objet carte de régions reprennent ceux des images de unsigned long, plus quelques unes spécifiques :

REGIS.CLOUARD@GREYC.ISMRA.FR

Création

- La déclaration d'une carte de régions se fait par:

```
Reg2d rgs1(256,512);
Reg2d rgs2(rgs1->Size());
Reg2d rgs3; // allocation sans taille donc sans données
Reg2d *rgs4; // allocation sans taille donc sans données
```

selon que l'on connaisse ou non la taille des données.

- La création d'une carte de régions à partir des caractéristiques d'un autre objet Pandore (Image, Carte de régions ou Graphe) se fait par:

```
Reg2d rgs1(ims2.Props());
```

Initialisation

- L'allocation des données d'une image allouée sans taille se fait par

```
New(Long, Long);  // En 2d
New(Dimension2d&); // En 2d
```

Exemple

```
Reg2d rgs3; rgs3.New(256,512);
Reg2d *rgs4; rgs4->New(256,512);
```

Attention: A la définition, la carte de régions n'est pas mise à 0. (utiliser rgs=0;)

Copie

- Le clonage d'une carte de régions (une copie parfaite et distincte) se fait par:

```
Pobject* Clone();
```

Exemple (r*emarque: Il faut caster puisque Clone retourne un Pobject**) :

```
Reg2d *rgd=(Reg2d*)rgs1.Clone();
```

Destruction

- La destruction des données se fait par:

```
void Delete();
```

Exemple :

```
rgs1.Delete();
rgs4->Delete();
delete rgs4; // rgs4 est détruite avec ses données.
```

Consultation

- La consultation (en lecture et en écriture) des valeurs de labels d'une carte de

REGIS.CLOUARD@GREYC.ISMRA.FR

régions se fait de deux façons :

- Classiquement, par l'opérateur `[profondeur][ligne][colonne]`:

```
rgs1[i][j]=15;
(*rgs4)[i][j]=15; avec un pointeur sur une carte de régions;
Point2d pt(10,20); rgs1[pt]=15; en utilisant un point.
```

- Rapidement, la carte de régions est vue comme un seul vecteur: `Vector()`

```
int i;
Reg2d rgs(256,512);
Ulong *p=rgs.Vector();

for (i=0; i< rgs.VectorSize();i++)
   *p++ = 15;
```

Transfert

- La lecture d'une carte de régions se fait par:

```
Errc LoadFile(Char *filename);
```

Exemple:

```
rgs.LoadFile("essai.pan");
```

- L'écriture d'une carte de régions se fait par:

```
Errc SaveFile(Char *filename);
```

Exemple:

```
rgs.SaveFile("essai.pan");
```

Utilitaires

- Hold(): Test d'appartenance d'un point aux limites d'une carte de régions:

```
bool Hold(Long l,Long h);
bool Hold(Point2d pt);


rgs.Hold(5,10) // retourne true si le point (5,10) appartient à rgs.
```

- Frame(): Remplissage du bord d'une carte de régions :

```
Errc Frame(val, h, l): Avec une valeur val.
Errc Frame(ims, h, l): Par recopie du bord d'une autre carte de régions.
```

- Mask(reg)/UnMask(reg,ims): Construction d'une carte de régions par masquage et démasquage des pixels par une carte de région :

  Le masquage (Mask()) met à 0 les pixels masqués (i.e. de label=0 dans la carte de régions).

REGIS.CLOUARD@GREYC.ISMRA.FR

Le démasquage (UnMask()) remet les pixels masqués originaux de ims seulement si cette carte de régions est de même type.

2.6 L'objet Graphe

L'objet graphe (Fig. 8) permet de représenter la notion de voisinage entre objets par un graphe valué et non orienté. Il n'existe donc qu'**un arc** entre deux sommets. Chaque sommet connaît l'ensemble de ses voisins, et chacun de ses voisins le connaît comme un de leurs voisins.

Un sommet du graphe (de type GNode) est en effet caractérisé par une valeur de type Float et par l'indice d'un objet dans un tableau d'objets définis par ailleurs. La structure de graphe est déconnectée de la liste des objets qu'il organise. L'intérêt c'est de pouvoir définir des graphes de n'importe quoi. Néanmoins, cela oblige à définir en même temps un graphe et un tableau d'objets. Par exemple, pour définir un graphe de régions, il faut un graphe et un tableau de régions. Chaque sommet i possède un entier (accessible par l'instruction g[i]->Item()) qui indexe un objet particulier dans un tableau annexe.

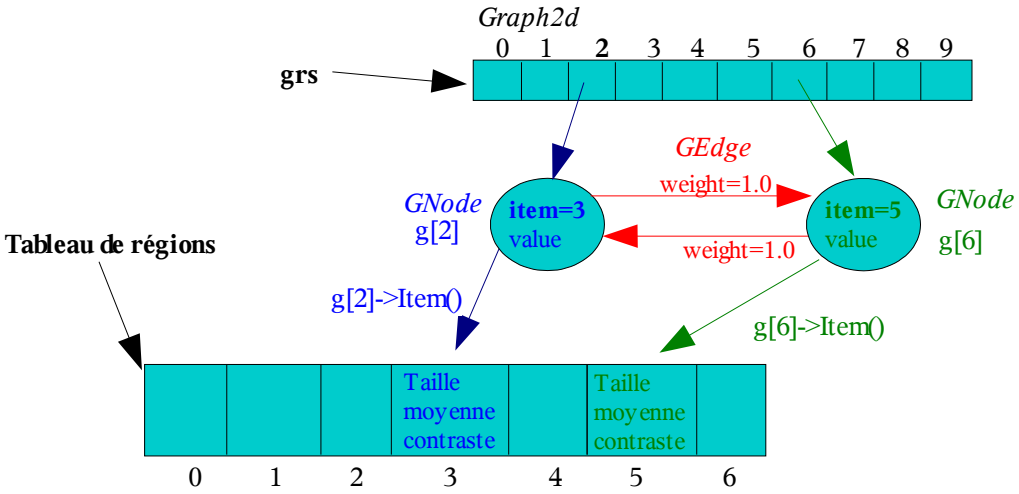

Fig. 8. Dans le graphe 2d grs, les noeuds 2 et 6 sont voisins. Le lien entre le noeud 2 et le noeud 6 a un poids de 1.0 et le lien entre 6 et 2 a un poids de -2.0. Le noeud 6 référence la région 3 dans le tableau des régions, alors que le noeud 6 référence la région 5.

REGIS.CLOUARD@GREYC.ISMRA.FR

Par exemple:

```
struct objet tab[50];    // un tableau d'une structure quelconque.
Long id= grs[i]->Item(); // id est l'index de l'objet dans le tableau tab.
tab[id].taille=120;      // Affecte la taille de l'objet id.
```

Un arc du graphe (de type GEdge) est caractérisé par son poids qui est une valeur de type Float. Il existe deux arcs entre deux voisins du graphe.

2.6.1 Types

Selon la représentation adoptée, il n'existe que deux types de graphes :

```
Graph2d : Un graphe 2d d'objets Pandore quelconques.
Graph3d : Un graphe 3d d'objets Pandore quelconques.
```

2.6.2 Attributs

Attributs de Graphe

Un Graphe est caractérisé par un tableau. **Ceci suppose donc que l'on définisse à l'avance le nombre de sommets que devra supporter le graphe.**

- `Typobj Type()`: le type de l'objet (par un identifiant).
- `Long Size()`: le nombre de sommets.
- `Long Width()`: le nombre de colonnes de l'image contenant le graphe.
- `Long Height()`: le nombre de lignes de l'image contenant le graphe.
- `Long Depth()`: le nombre de plans de l'image contenant le graphe.
- `Dimension ImageSize()`: les dimensions de l'espace image contenant le graphe.
- `PobjectProps Props()`: les propriétés du graphe dans la structure PobjectProps (taille de l'image associée, nombre de sommets).

Attributs de Sommet

L'accès aux valeurs d'un sommet du graphe :

- `Double value`: : la valuation du sommet.
- `Long Item([int x])`: le numéro qui permet d'identifier l'objet dans le tableau d'objet(si x est présent en affectation, sinon en lecture).
- `Point2d seed`: : les coordonnées du sommet dans l'image d'origine.
- `GEdge* Neighbours()`: la liste des sommets adjacents.

REGIS.CLOUARD@GREYC.ISMRA.FR

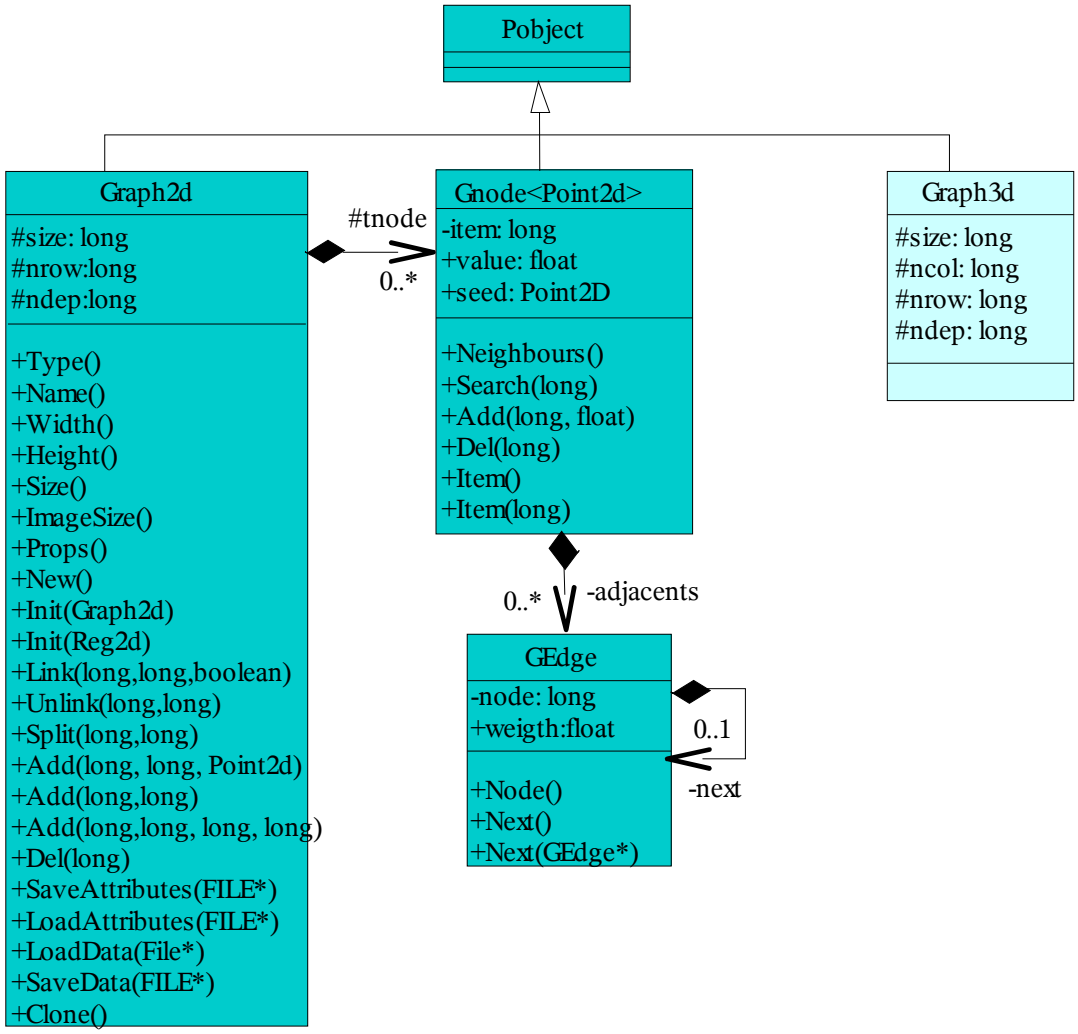

Fig. 9. Les classes de graphe (tnode est un tableau de GNode de dimension size).

Attribut de Arc

L'accès aux attributs d'un arc entre deux sommets :

- `Long Node()`: l'indice du voisin.
- `Double weight`: le poids de l'arc.
- `GEdge* Next([GEdge* x])`: le voisin suivant (si x est présent en affectation, sinon en lecture).

REGIS.CLOUARD@GREYC.ISMRA.FR

2.6.3 Méthodes

Création

- La déclaration d'un graphe :

```
Graph2d g(100) ou Graph2d g; ou Graph2d *g;  // en 2D
Graph3d g(100) ou Graph3d g; ou Graph3d *g;  // en 3D
```

- La création d'un graphe à partir des caractéristiques d'un autre objet Pandore (Image, Carte de régions ou Graphe) se fait par:

```
Graph2d gs1(obj2.Props());
```

Attention, si obj2 est un graphe, alors le nombre de sommets de gs1 est égal au nombre de sommets de obj2. Si obj2 est une carte de régions, alors le nombre de sommets de gs1 est égal au nombre de labels+1 de obj2. Si obj2 est une image, alors le nombre de sommets de gs1 est égal au nombre de pixels de obj2.

Initialisation

- La création d'un graphe vierge pouvant posséder jusqu'à `size` sommet utilise :

```
Errc New(size);
```

Exemple :

```
g.New(100); Graph2d *gs1; gs1->New(100);
```

- La création d'un nouveau graphe à partir de la carte de région rgs utilise (*les germes sont positionnés au point le plus supérieur gauche de la région pour qu'ils soient dans la région à la différence avec le barycentre*) :

```
Errc Init(const Reg2d &rgs);
```

Exemple :

```
g.Init(rgs);
```

- La création d'un nouveau graphe à partir de la carte de région `rgs`, et des coordonnées des germes à partir de la carte des germes (*les germes doivent être des points*) se fait par :

```
Errc Init(const Reg2d &rgs,const Reg2d &grm);
```

Exemple :

REGIS.CLOUARD@GREYC.ISMRA.FR

```
grs.Init(rgs1,germ);
```

- La création d'un nouveau graphe identique à un autre se fait par :

```
Graph2d gs1(grs2->Props()); gs1 = grs2;
```

Destruction

- La destruction d'un graphe se fait par :

```
Errc Delete();
```

Exemple:

```
gs1.Delete(); gs2->Delete();
```

Copie

- Le clonage d'un graphe (copie parfaite et distincte) se fait par :

```
Pobject* Clone();
```

Exemple:

```
Graph2d *gd=(Graph2d*)g2.Clone();
```

Remarque: Il faut caster puisque Clone() retourne un Pobject*

Consultation

- Ajouter s1 comme voisin de s2, et réciproquement. Si l'arc existe déjà met à jour la valeur du poids. La valeur de l'arc est par défaut égale à 1.0 et elle est affectée si add=false ou additionnée à la valeur existante si add=true :

```
Errc Link(Long s1,Long s2, Float w=1.0F, bool add=false);
```

Exemple de création d'un arcs avec la valeur de poids 1.0:

```
grs.Link(10,12);
```

Exemple de mise à jour d'un arcs en additionnant la valeur 5.0 à la valeur existante :

```
grs.Link(10,12,5.0,true);
```

- Supprimer s1 de la liste des voisins de s2 et réciproquement, se fait par :

```
Errc Unlink(Long s1,Long s2);
```

Exemple :

```
grs.Unlink(10,12);
```

- Ajouter un sommet s dans le graphe référençant un objet d'index *item* et des

REGIS.CLOUARD@GREYC.ISMRA.FR

coordonnées (y,x):

```
Errc Add(Long s,Long item ,Long y=0,Long x=0);          // en 2d
Errc Add(Long s,Long item, Long z=0,Long y=0,Long x=0); // en 3d
Errc Add(Long s,Long item ,Point2d pt);                 // en 2d
Errc Add(Long s,Long item ,Point3d pt);                 // en 3d
```

Exemple: Ajouter le sommet 5 référençant l'objet numéro 6 aux coordonnées 3D z=50,y=12,x=10.

```
grs.Add(5,6,Point3d(50,12,10));
```

- La suppression du sommet s du graphe se fait par:

```
Errc Del(Long s);
```

Exemple:

```
grs.Del(10);  // Supprime le sommet d'indice 10.
```

- Pour obtenir l'objet référencé par le noeud i:

```
Long Item();
```

Exemple:

```
long numero_region= g[i]->Item();
```

Attention: g[i] n'existe pas toujours, notamment lorsqu'un noeud a été supprimé avec Del(). Il est préférable au préalable testé son existence avec if ((g[i])) .. g[i]-> ...

- Pour obtenir la liste des voisins d'un sommet:

```
GEdge* Neighbours();
```

Exemple:

```
GEdge* l = g[i]->Neigbours();
```

- Le parcours des voisins se fait donc par:

```
l=l->Next(); jusqu'à l==NULL;


for (GEdge* l = g[i]->Neigbours(); l!=NULL; l=l->Next())
   int voisin = ptr->Node();
```

Transfert

- Le chargement d'un graphe à partir d'un fichier se fait par:

```
Errc LoadFile(char*);
```

Exemple:

```
grs.LoadFile("essai.pan");
```

- La sauvegarde d'un graphe dans un fichier se fait par:

REGIS.CLOUARD@GREYC.ISMRA.FR

```
Errc Save(char*) const;
```

Exemple:

```
ims.Save("essai.pan");
```

Utilitaires

- La fusion de 2 sommets en 1 avec mise à jour de la liste des arcs se fait par :

```
Errc Merge(Long s1,Long s2);
```

Le sommet s1 est conservé et le sommet s2 est détruit. La liste des arcs du sommet s2 est ajoutée à celle du sommet s1. La valeur du poids des arcs communs *est additionnée* dans le résultat.

- La division du sommet s1 pour former un autre sommet s2 se fait par :

```
Errc Split(Long s1,Long s2);
```

Les deux sommets deviennent des copies conformes (même liste de voisins).

2.7 L'objet Collection

Les collections sont des conteneurs d'objets hétérogènes. Il rassemble dans une même structure et dans un même fichier des données de types différents qui peuvent être des valeurs de type simple (Float, Ulong, ...), des tableaux de type simple, des objets Pandore (image, carte de régions, ..) ou des tableaux d'objets Pandore. Les collections sont notamment utilisées pour stocker des vecteurs de valeurs ou des collections d'images.

2.7.1 Type

Il n'y a qu'un type de collection (Fig. 10)

```
Collection;   // Une collection de n'importe quoi.
```

2.7.2 Attributs

Les collections peuvent contenir :

1. des types de base (Char, Uchar, Short, Ushort, Long, Ulong, Float, Double);
2. des tableaux de ces même types;
3. des objets Pandore Image, Carte de régions, Graphes, Collection,

REGIS.CLOUARD@GREYC.ISMRA.FR

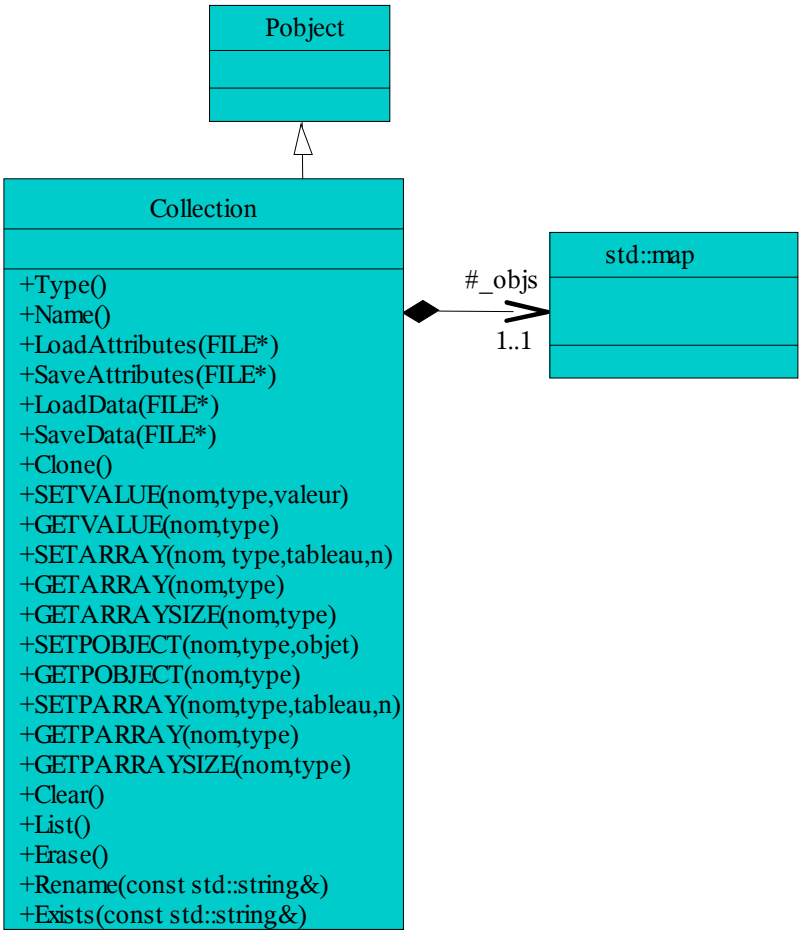

Fig. 10. La classe Collection.

4. des tableaux d'objets Pandore.

Chaque objet à l'intérieur de la collection est identifié par son nom (une chaîne de caractères).

2.7.3 Méthodes

Création

- La déclaration d'une collection se fait par :

```
Collection c1;
Collection *c2 = new Collection;
```


REGIS.CLOUARD@GREYC.ISMRA.FR

Initialisation

- `Clear()` : vide une collection.

Consultation

L'utilisation la plus simple d'une collection se fait avec les macros:

- pour les types de base :

```
SETVALUE(nom,type,valeur);
GETVALUE(nom,type);            -> retourne du type&
```

Exemple :

```
Float f=1.2;
col.SETVALUE("essai",Float,f);
f=GETVALUE("essai",Float);
```

- pour les tableaux de types simples :

```
SETARRAY(nom,type,pointeur_sur_tableau,numbre_d_elements_du_tableau)
GETARRAY(nom,type)             -> retourne un type*
GETARRAYSIZE(nom,type)
```

Exemple :

```
Ushort *t1 = new Ushort[15]; *t2;
col.SETARRAY("tablo",Ushort,t1,15);
int x=col.GETARRAYSIZE("tablo",Ushort);
t2=col.GETARRAY("tablo",Ushort);
```

- pour les objets Pandore :

```
SETPOBJECT(nom,type,pointeur_sur_objet)
GETPOBJECT(nom,type)           -> retourne un type&
```

Exemple :

```
Imc3duc *im1=new Imc3duc(25,45,260), *im2;
col.SETPOBJECT("essai",Imc3duc,im1)
im2=GETPOBJECT("essai",Imc3duc)
```

- pour les tableaux d'objets Pandore :

```
SETPARRAY(nom,type,pointeur_sur_tableau,numbre_d_elements_du_tableau)
GETPARRAY(nom,type)
GETPARRAYSIZE(nom,type)        -> retourne un type**
```

Exemple :

```
Point2d **p1, **p2;

p1=new Point2d*[12];
for (int i=0; i<12; i++) p1[i]=new Point2d(i,i);
cold.SETPARRAY("essai",Point2d,p1,12);
p2=(Point2d**)cols.GETPARRAY("essai",Point2d);
```

Attention 1 : SETARRAY, SETPOBJECT et SETPARRAY ne font pas de copie des tableaux et des objets ajoutés à la collection. Dans l'exemple précédent, il est donc

REGIS.CLOUARD@GREYC.ISMRA.FR

possible de continuer à travailler avec `i` et `f` sans avoir à faire à nouveau un SET par la suite, mais il faut veiller à ne pas effacer ces objets tant que la collection les référence. En conséquence, l'exemple suivant ceci provoque un erreur :

```
Errc Essai(Collection  &cold){
   Point2d *p1[12];
   cold.SETPARRAY("essai",Point2d,p1,12);
}
```

parce que p1 est un tableau local qui est détruit au sortir de la fonction Essai. De ce fait, le tableau dans la collection devient donc un emplacement mort.

Attention 2 : Après un GETXXX, les emplacements mémoire pour le tableau et l'image ont été alloués par la collection lors du chargement du fichier et leur libération est donc à la charge de la collection. Dès la destruction d'une collection tous les objets crées par la collection sont aussi détruits.

Remarque 1 :Dans le case d'un GETXXXX le type fournit en paramètre est vérifié à l'exécution, si l'attribut n'est pas de ce type, le programme s'achève avec un message d'erreur.

Remarque 2 :Dans le cas d'un SETXXXX, si le nom existe déjà alors l'objet correspondant est remplacé.

Copie

- Le clonage d'une collection (une copie mais avec partage des pointeurs vers les données) se fait par :

```
Pobject* Clone();
```

Exemple (*remarque: Il faut caster puisque Clone retourne un Pobject**) :

```
Collection *cold=(Collection*)cold.Clone();
```

Utilitaires

- `List()` retourne la liste de tous les attributs de la collection. Cette liste est de type `std::list<std::string>`.
- `Exists(const std::string&)`: teste l'existence d'un attribut.
- `Erase(const std::strings&)` effacer un attribut.
- `Rename(const std::string&,const std::string&)` permet de renommer un attribut.

REGIS.CLOUARD@GREYC.ISMRA.FR

Cas particuliers d'un nombre de composantes variable

Lorsque pour un même individu il faut retenir plusieurs caractéristiques, nous avons convenu de représenter cela par plusieurs tableaux dans une collection, en utilisant des noms de la forme `nom.n` où `n` est le numéro de la caractéristique.

Par exemple les composantes rouge, verte et bleu de 10 pixels seront représentées par trois tableaux `t.1`, `t.2` et `t.3` de 10 éléments chacun.

Un opérateur souhaitant travailler sur un nombre de composantes variable devra, étant donné un préfixe `nom`, rechercher le plus grand `n` tel que `nom.1`, `nom.2`, ..., `nom.n` existent et soient de types identiques (il est impossible de gérer n types différents puisque les types doivent être connus à la compilation alors que le n n'est déterminé qu'à l'exécution).

Ce rôle est celui de la méthode `NbOf` de la classe `Collection`:

```
Errc NbOf(const std::string &s, std::string &type_out,
          Long &nbrarray_out, Long &minelem_out) const;
```

`s` est le préfixe des noms à rechercher; après l'appel, `type_out` contiendra le type des tableaux trouvés, `nbrarray_out` le nombre de tableaux trouvés (c'est-à-dire `n`) et `minelem_out` la taille du plus petit de ces tableaux. `NbOf` retourne FAILURE si aucun tableau n'est trouvé.

La macro `GETNARRAYS` permet de récupérer un tableau de pointeurs:

```
GETNARRAYS(name,type,nombre,minelem_out) -> retourne du type**
```

`name` est encore un fois le préfixe des noms des tableaux, `type` est le type des éléments des tableaux, `nombre` le nombre de tableaux à obtenir (ce dernier peut-être déterminé avec `NbOf`), et enfin `minelem_out` doit être le nom d'une variable `Long` qui contiendra la taille minimum des tableaux récupérés.

- `src/operatorsP0/classification/kppv.cc` est un exemple d'opérateur utilisant ces deux appels pour manipuler un nombre inconnu de composantes.

Transfert

- La sauvegarde d'une collection se fait par:

```
Collection c;
c.SaveFile(FILE*);
c.SaveFile(char *);
```


REGIS.CLOUARD@GREYC.ISMRA.FR

- La lecture d'une collection se fait par :

```
Collection c;
c.LoadFile(FILE*);
c.LoadFile(char *);
```

2.8 Les objets Piles, Files et Tas

Les listes permettent d'ordonner selon plusieurs modèles, des éléments de n'importe quel type, mais homogènes dans une même liste. A la différence des graphes, les listes utilisent les templates C++, ce qui oblige à préciser à la définition de la liste le type de ses éléments.

2.8.1 Types

Il existe quatre types de liste (Fig. 11) contenant des éléments de type quelconque :

1. `Lifo: Une pile d'éléments (dernier empilé premier dépilé).`
2. `Fifo: Une file d'éléments (premier enfilé premier défilé).`
3. `Heap: Un tas est une liste d'éléments ordonnée de façon croissante selon une valeur de clé (i.e., la clé de valeur minimale est en tête de tas).`
4. `OrderedFifo: Une file prioritaire d'éléments qui se conçoit comme un tableau de fifo (i.e., Les éléments de même clé sont ordonnés selon une file). L'accès à un élément se fait par un numéro de clé, puis par le premier enfilé, premier défilé.`

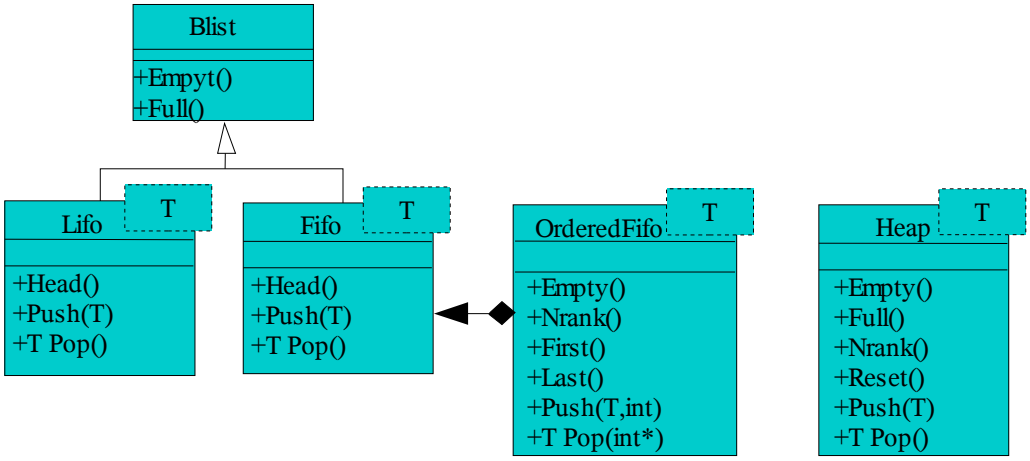

Fig. 11. Modélisation des classes de gestion de listes.

2.8.2 Attributs

Les listes sont caractérisées par leur nombre d'éléments. Accès aux attributs :

- Empty() permet de savoir si la liste est vide.

REGIS.CLOUARD@GREYC.ISMRA.FR

```
f.Empty(), p.Empty() ou t.Empty().
```

- Full() permet de savoir si la liste est pleine.

```
f.Full(), p.Full() ou t.Full().
```

- Pour le tas et la liste prioritaire, Nrank() permet de connaître le nombre de cases occupées.

```
t.Nrank()
```

2.8.3 Méthodes

Les types Lifo (pile), Fifo (file), Heap (tas) et OrderedFifo (file prioritaire) permettent de construire des piles, des files, des tas ou des files prioritaires d'un type donné.

Création

- La déclaration d'une file, d'une pile ou d'un tas d'un *type* donné se fait par :

```
Fifo <TYPE> f;
Lifo <TYPE> l;
Heap <TYPE> t(n); // n est la taille maximale du tas
OrderedFifo <TYPE> fp(n); // n est le nombre maximum de files (=de clés).
```

Par exemple :

```
Fifo <Point2d> f;              // Une file de Point2d
Lifo <int> f;                  // une pile d'entiers
Heap <float> f(100);           // Un tas de 100 float.
OrderedFifo <float> fp(100);  // Une file ordonnée de 100 files de float.
```

Initialisation

La déclaration d'une liste suffit à initialiser la liste à Empty().

Pour les tas, la méthode Reset() permet de vider un tas.

Consultation

- Pour empiler, enfiler ou entasser une valeur du bon type (avec une certaine valeur de clé (réelle) pour le tas et (entière) pour les files prioritaires) :

```
bool f.Push(const TYPE &valeur);
bool p.Push(const TYPE &valeur);
bool t.Push(const TYPE &valeur,Float clé);
bool fp.Push(const &TYPE&,Long clé)
```

Exemple

```
f.Push(new Point2d(10,20));   // Enfile le point 20,10
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
p.Push(10);                    // Empile l'entier 10
t.Push(1.0,50.5);              // Entasse le réel 1.0 avec la clé 50.5;
fp.Push(5.0,10);               // Enfile le float 5.0 dans la file 10.
```

- Pour dépiler, défiler ou détasser (avec la possibilité de récupérer la valeur de la clé pour le tas et les files prioritaires) :

```
valeur=f.Pop()
valeur=p.Pop()
Float clet; valeur=t.Pop(&clet); (Ici la clé est optionnelle).
Long clefp; valeur=fp.Pop(&clefp); (Ici la clé est optionnelle).
```

Parcours

Le parcours avec défilement, dépilement ou détassement des éléments se fait classiquement par :

```
Fifo<TYPE> F;
while (! F.Empty()){
   type x=F.Pop();
}
```

Pour les files et les piles, il y a la possibilité de parcourir les éléments sans les supprimer de la liste. La démarche est la suivante, pour parcourir la file avec le pointeur p :

```
Fifo<type> F
Tlist<type> *p;
p=F.Head();
Fonction(p->Item());     // Application d'une fonction sur l'élément.
p=p->Next();
```

Exemple :

```
Fifo<Point2d> F
Tlist<Point2d> *it;
for (it=F.Head(); it->HasNext(); it=it->Next()) {
   Point2d p=->Item();
   std::cout << "coordonnées: y=" << p.y << "," <<p.x << std::endl;
}
```

Transfert

Les listes ne peuvent pas être sauvegardées ni dans fichier ni dans une collection. Par contre, il est possible de transformer la liste des éléments en un tableau qui peut ensuite être sauvegardé dans une collection. La liste pourra alors être reconstruite à partir du tableau des éléments.

REGIS.CLOUARD@GREYC.ISMRA.FR

3 La programmation d'opérateurs

Ici s'agit ici de compléter la bibliothèque avec de nouveaux opérateurs qui respectent au mieux le modèle d'opérateur Pandore (Voir le manuel de référence). Les opérateurs se distinguent des applications dans le sens où ils n'effectuent qu'un traitement ponctuel.

3.1 Fichier squelette

Le fichier squelette ci-dessous donne la structure commune des fichiers opérateurs.

```
/*
 * @(#)operateur.cc      PANDORE           2004-07-01
 *
 * PANTHEON Project
 *
 * GREYC IMAGE
 * 6 Boulevard Maréchal Juin
 * F-14050 Caen Cedex
 *
 * This file is free software. You can use it, distribute it
 * and/or modify it. However, the entire risk to the quality
 * and performance of this program is with you.
 */

/*
 * (C)Régis Clouard - 2004-07-01
 */

/**
 * Description of the operator.
 */

#include <pandore.h>

/*
 * Comments on function.
 */
Errc Operateur(Img2duc &ims, Img2duc &imd, Short p)
{
  return SUCCESS;
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
}

/*
 * Comments on function.
 */
Errc Operateur(Img2dsl &ims, Img2dsl &imd, Short p)
{
  return SUCCESS;
}

#ifdef MAIN
/*
 * Modify only the following constants, and the operator switches.
 */
#define  USAGE "USAGE : %s parameter [-m mask] [im_in|-] [im_out|-]"
#define  PARC  1  // Number of parameters
#define  FINC  1  // Number of input images
#define  FOUTC 1  // Number of output images
#define  MASK  1  // Level of masking

int main(int argc,char* argv[])
{
  Errc result;               // The result code of the execution.
  Pobject* mask;             // The mask.
  Pobject* objin[FINC+1];    // The input objects;
  Pobject* objs[FINC+1];     // The source objects masked by the mask.
  Pobject* objout[FOUTC+1];  // The output object.
  Pobject* objd[FOUTC+1];    // The result object of the execution.
  Float parv[PARC+1];        // The input parameters.

  ReadArgs(argc,argv,PARC,FINC,FOUTC,&mask,
           objin,objs,objout,objd,parv,USAGE,MASK);

  switch(objd[0]->Type()){
  case Po_Img2duc: {
     Img2duc* const ims=(Img2duc*)objs[0];
     objd[0]=new Img2duc(ims->Props());
     Img2duc* const imd=(Img2duc*)objd[0];

     result=Operateur(*ims,*imd,(Short)parv[0]);
     break;
  }
  case Po_Img2dsl: {
     Img2dsl* const ims=(Img2dsl*)objs[0];
     objd[0]=new Img2dsl(ims->Props());
     Img2dsl* const imd=(Img2dsl*)objd[0];

     result=Operateur(*ims,*imd,(Short)parv[0]);
     break;
  }
  default:
     PrintErrorFormat(objin,FINC);
     result=FAILURE;
  }

  WriteArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,objout,objd,MASK);

  Exit(result);
  return 0;
}
#endif
```

3.2 La fonction opérateur

Les données d'entrée et de sortie sont des objets de l'environnement Pandore. La définition d'objets très spécialisés permet une vérification des types des arguments

REGIS.CLOUARD@GREYC.ISMRA.FR

effectifs par le compilateur à partir du prototype de l'opérateur. Ceci évite à l'opérateur de vérifier lui-même dynamiquement la cohérence des types de ses arguments et contribue à la réduction des sources d'erreurs à l'exécution. Par contre, cela oblige à écrire un opérateur pour chaque type d'objet concerné. Mais ce n'est pas nécessairement un handicap si l'on considère par exemple que pour la plupart des opérateurs, une image de réels ne se traite pas exactement de la même façon qu'une image d'entiers (cas par exemple des opérateurs utilisant l'histogramme des niveaux de gris). Il existe aussi des moyen de faire de la programmation générique (voir §5).

L'exemple ci-dessous présente un exemple d'une fonction opérateur d'érosion.

```
/*
 * Si l'union des points masqués est non nulle
 * alors forcer le point central à FAUX.
 */
Errc Erosion(Img2duc &ims,Img2duc &imd,Short connexite)
{
   Point2d p;
   Uchar min,val;

   imd.Frame(0,1,1);
   if (connexite == 4){
      for (p.y=1; p.y < ims.Width()-1; p.y++)
         for (p.x=1; p.x < ims.Height()-1; p.x++){
            min=ims[p+v4[0]];
            for (int v=1; v < 4; v++)
               if ((val=ims[p+v4[v]] < min)
                  min = val;
            imd[p] = min;
         }
   }else{ // connexite == 8.
      for (p.y=1; p.y <= ims.Height()-1; p.y++)
         for (p.x=1; p.x < ims.Width()-1; p.x++){
            min=ims[p+v8[0]];
            for (int v=1; v < 8; v++)
               if ((val=ims[p+v8[v]] < min)
                  min = val;
            imd[p] = min;
         }
   }
   return SUCCESS;
}
```

3.3 La fonction Main()

La fonction main a en charge la lecture des arguments d'entrée de la ligne de commande, l'aiguillage vers la bonne fonction opérateur en fonction des entrées, la création des objets de sortie et l'affectation de la valeur de sortie.

REGIS.CLOUARD@GREYC.ISMRA.FR

```
#ifdef MAIN
#define USAGE "USAGE: %s connexite [-m mask] [im_in|-] [im_out|-]"
#define PARC  1
#define FINC  1
#define FOUTC 1
#define MASK  1

int main(int argc,char* argv[])
{
   Errc result;              // The result code of the execution.
   Pobject *mask;            // The region map.
   Pobject *objin[FINC+1];   // The input objects;
   Pobject *objs[FINC+1];    // The source objects masked by mask.
   Pobject *objout[FOUTC+1]; // The output object unmasked by mask.
   Pobject *objd[FOUTC+1];   // The result object.
   Float parv[PARC+1];       // The input parameters.

   ReadArgs(argc,argv,PARC,FINC,FOUTC,&mask,
            objin,objs,objout,objd,parv,USAGE,MASK);
   switch(objs[0]->Type()){
   case Po_Img2duc :{
      Img2duc* const ims=(Img2duc*)objs[0];
      objd[0]=new Img2duc(ims->Props());
      Img2duc* const imd=(Img2duc*)objd[0];

      result=Erosion(*ims,*imd,(Short)parv[0]);
   }break;
   case Po_Img2dsl :{
      Img2dsl* const ims=(Img2dsl*)objs[0];
      objd[0]=new Img2dsl(ims->Props());
      Img2dsl* const imd=(Img2dsl*)objd[0];

      result=Erosion(*ims,*imd,(Short)parv[0]);
   }break;
   default    :
      PrintErrorFormat(objin,FINC);
      result=FAILURE;
   }

   WriteArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,objout,objd);

   Exit(result);
   return 0;
}
#endif
```

3.3.1 La lecture des arguments d'entrée

La fonction ReadArgs() effectue la vérification et la lecture des arguments d'entrée.

```
ReadArgs(argc,argv,PARC,FINC,FOUTC,&mask,
  objin,objs,objout,objd,parv,USAGE,MASK);
```

ReadArgs dépend d'un ensemble de constantes qui permettent de prototyper la ligne de commande de l'opérateur :

- la chaîne présentant l'USAGE pour une aide en ligne;
- le nombre de paramètres exact (PARC);
- le nombre d'image d'entrée (FINC);
- le nombre d'image de sortie (FOUTC);

REGIS.CLOUARD@GREYC.ISMRA.FR

- L'opération de masquage: (MASK) (pour plus de détail voir le § 3.1.4)
  - MASK=0: pas de masquage ni entrée ni en sortie.
  - MASK=1: masquage + démasquage.
  - MASK=2: (version 3) uniquement le masquage en entrée.
  - MASK=3: (version 3) uniquement le démasquage en sortie.

Donc, si Mask=1 ou Mask=2 alors les entrées sont masquées avant d'être envoyés vers les fonctions opérateurs.

3.3.2 L'aiguillage vers la fonction opérateur

Il faut ensuite préciser l'appel de la bonne fonction en fonction du type des images d'entrée dans le switch.

```
switch(objs[0]->Type()){
case Po_Img2duc :{
}
```

Le choix de la bonne fonction se fait à partir du `Type()` d'un objet Pandore.

3.3.3 Affichage d'un message d'erreur en cas d'entrées incorrectes

```
PrintErrorFormat(objin,FINC);
```

Cette fonction permet d'afficher un message d'erreur sur la sortie erreur lorsque l'une des entrées n'est reconnue par l'aiguillage précédent.

3.3.4 L'écriture des objets de sortie

La fonction WriteArgs() effectue la vérification et la lecture des arguments d'entrée.

```
WriteArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,objout,objd);
```

Cette fonction reprend les mêmes valeurs de constantes que la fonction `ReadArgs`. Cette fois, si mask=1 ou mask=3 alors les images de sortie des fonction opérateurs sont démasquées avant d'être retournées comme image de sortie.

L'affectation de la valeur de sortie.

```
Exit(result);
```

L'exécution d'un opérateur Pandore crée une valeur de sortie qui peut être d'un des types de base Pandore : Errc, Uchar, Char, Short, Ushort, Long, Ulong, Float, ou Double. Cette valeur est ensuite récupérable par l'opérateur `statut`.

REGIS.CLOUARD@GREYC.ISMRA.FR

3.4 Masquage et démasquage des entrées et des sorties

Le masquage permet d'appliquer un même opérateur à une image entière ou à une partie déterminée par un masque. Le masque est ici une carte de régions dont les valeurs à 0 indiquent les pixels à masquer.

Pour implanter cela, les images d'entrée sont préalablement filtrées par la carte des régions : Tous les pixels de l'image initiale qui ne font pas partie des régions de la carte sont mis à 0, les autres sont laissés à leur valeur initiale. Après application de l'opérateur, les images de sortie sont reconstruites à partir des valeurs de pixels de l'image d'entrée non atteintes par le masque et celles qui ont été modifiées par l'opérateur).

Le principe du masquage se compose donc d'une construction de l'image à traiter par sélection des pixels masquées (**Mask**), puis d'une opération de composition de l'image de sorties avec l'image traitée et les pixels non masqués (**UnMask**).

Pour un opérateur donné, la sortie numéro i est construite par démasquage sur l'entrée numéro i correspondante si celle ci est du même type.

La constante du main MASK permet de spécifier le type de masquage à implémenter pour l'opérateur.

- MASK=0: pas de masquage ni entrée ni en sortie.
- MASK=1: masquage + démasquage.
- MASK=2: (version 3) uniquement le masquage en entrée.
- MASK=3: (version 3) uniquement le démasquage en sortie.

Remarque : Quelques fois, l'opération de masquage peut provoquer des problèmes sur les bords des régions masquées (e.g. moyenneur). Dans ce cas, l'opération de masquage se limitera au "démasquage". D'autres fois, il ne faut pas recopier les pixels masqués de l'image source dans l'image destination. L'opération se limitera cette fois au masquage. Le choix du type de masquage est indiqué par la valeur de la constante MASK du main().

REGIS.CLOUARD@GREYC.ISMRA.FR

4 La programmation d'applications

Les applications se distinguent des opérateurs dans le sens où ils s'écrivent comme l'enchaînement de plusieurs opérateurs. Puisque les opérateurs sont disponibles à la fois comme commandes du système d'exploitation et comme codes C++, la programmation d'une application peut prendre deux formes:

1. Un script de commandes (Shell sous UNIX / Batch sous DOS);
2. Un programme C++.

Les scripts seront préférés aux programmes C++ lors la phase de mise au point. Les programmes C++ seront utilisés lors de la phase d'optimisation. Nous ne traitons ici que de la programmation d'applications sous la forme de programmes C++. Pour la description de la façon de réaliser des scripts d'opérateurs Pandore, il faut se reporter au manuel de référence.

4.1 Une application sous la forme d'un programme C++

Le programme C++ d'une application Pandore consiste simplement à appeler les fonctions opérateurs Pandore. Il suffit pour cela, d'inclure directement le fichier C++ correspondant à l'opérateur dans le fichier du programme applicatif. Évidemment tout ce qui est utilisable pour l'écriture des opérateurs, l'est aussi pour l'écriture des applications. Nous ne présentons ici que ce qui est particulier aux applications.

REGIS.CLOUARD@GREYC.ISMRA.FR

4.2 L'inclusion des opérateurs

L'appel des opérateurs se fait directement par inclusion du fichier c++ dans un espace de nom et en prenant soin d'annuler l'inclusion du `main` correspondant.

Par exemple, les directives suivantes permettent d'utiliser l'opérateur `moyenneur` dans une application sous Unix :

```
#undef MAIN
namepace moyenneur{
   #include "morphologie/moyenneur.cc"
}
```

ou sous Windows :

```
#undef MAIN
namepace moyenneur{
   #include "morphologie\moyenneur.cpp"
}
```

L'espace de nom permet d'éviter les conflits de noms de variables et de fonctions. Grâce à cette inclusion, il est alors possible d'utiliser à l'intérieur du programme les fonctions du fichier. Par exemple, toujours dans le cas du moyenneur:

```
retour = moyenneur::Moyenneur(ims,imd,8);
```

4.3 La fonction Main()

Les flux de données entre les opérateurs sont faits d'objets Pandore et de valeurs de retour des fonctions.

Voici un exemple d'un programme C++ :

```
/* -*- c-basic-offset: 3 -*-
 * @(#)application.cc         PANDORE                    2004-07-01
 */

#include <pandore.h>

#undef MAIN

ifdef _WIN32
namespace moyenneur{
#include "lissage/moyenneur.cc"
}
#else
#endif

namespace binarisationvariancemax{
#include "classification/binarisationvariancemax.cc"
}

#define USAGE  "USAGE : %s [im_in|-] [im_out|-]"
#define PARC   0    // Number of parameters
#define FINC   1    // Number of input images
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
#define FOUTC 1    // Number of output images

int main(int argc, char *argv[])
{
   Pobject* mask;             // The mask.
   Pobject* objin[FINC+1];    // The input objects;
   Pobject* objs[FINC+1];     // The source objects masked by the mask.
   Pobject* objout[FOUTC+1];  // The output object.
   Pobject* objd[FOUTC+1];    // The result object of the execution.
   Float parv[PARC+1];        // The input parameters.

   ReadArgs(argc,argv,PARC,FINC,FOUTC,
            &mask,objin,objs,objout,objd,parv,USAGE);

   Img2duc* const ims=(Img2duc*)objs[0];
   objd[0]=new Img2duc(ims->Props());
   Img2duc* const imd=(Img2duc*)objd[0];

   Img2duc im1(ims->Props());
   Img2duc im2(ims->Props());
   Img2duc im3(ims->Props());

   im1=*ims;
   for (int i=0;i<2;i++){
      moyenneur::Moyenneur(im1,im2,8);
      im1=im2;
   }

   Long seuil=binarisationvariancemax::BinarisationVarianceMax(im2,im3);

   printf("Seuillage à la valeur : %d\n",seuil);
   *imd=im3;

   WriteArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,objout,objd);

   Exit(SUCCESS);
   return 0;
}
```


REGIS.CLOUARD@GREYC.ISMRA.FR

5 La programmation générique

Afin de facilité l'écriture d'opérateur contenant des fonctions pour un grand nombre de types, et d'automatiser l'écriture des fonctions `main`, Pandore propose des facilités pour les déductions et les sélections de types et un préprocesseur pour permettre la génération des fonctions opérateurs adaptés à différents types d'objet Pandore à partir d'un seul code générique.

5.1 Déduction et sélection de types

Plusieurs structures sont définies pour pouvoir déduire les propriétés d'un type, ou choisir entre deux types. Ces déductions deviennent nécessaires lorsqu'un algorithme est écrit de façon indépendante des types d'images manipulées (le type devient paramètre de la fonction).

5.1.1 Type des données: ValueType

Pour un type d'image `T` donné, le type `T::ValueType` représente le type des élements de `T`.

```
Par exemple, Img2dsf::ValueType est Float.
```

5.1.2 Limites des valeurs: Limits

`Limits<T>::max()` et `Limits<T>::min()` renvoient respectivement le maximum et le minimum du type primitif `T` (Uchar, Slong, Float....).

```
Par exemple Limits<Ushort>::max() retourne la valeur 65535.
```


REGIS.CLOUARD@GREYC.ISMRA.FR

5.1.3 Sélection de types: Select

Il est parfois nécessaire de choisir entre deux types. Par exemple un algorithme peut prendre deux types `T1` et `T2` en entrée, et vouloir donner un résultat avec *le type non signé de plus grande taille parmi `T1` et `T2`*. Un tel choix peut-être effectué grâce à la structure `Select`.

1. `Select<T1,T2>::LargestUnsigned`: La version unsigned du plus grand des 2.
2. `Select<T1,T2>::LargestSigned`: La version signed du plus grand des 2.
3. `Select<T1,T2>::SmallestUnsigned`: La version unsigned du plus petit des 2.
4. `Select<T1,T2>::SmallestSigned`: La version unsigned du plus petit des 2.
5. `Select<T1,T2>::Largest`: Le plus grand des 2 (signed > unsigned).
6. `Select<T1,T2>::Smallest`: Le plus petit des 2 (signed > unsigned).

```
Par exemple:
Select<Uchar,Short>::LargestSigned vaut Short;
elect<Img3duc,Img3dsl>::LargestSigned vaut Img3dsl.
```

Le résultat des 4 premiers champs de Select peut ne pas faire partie des deux types donnés en paramètres.

```
Par exemple: Select<Uchar,Short>::LargestUnsigned qui vaut Ushort.
```

Par contre, les deux derniers (largest, smallest) retournent un type appartenant obligatoirement à l'un des deux paramètres:

```
Par exemple: Select<Uchar,Char>::Largest qui vaut Char
             Select<Uchar,Char>::Smallest qui vaut Uchar
```

5.2 Utilisation des templates pour l'écriture générique

Pour écrire un code d'opérateur générique, il est possible de profiter du fait que tous les types d'images sont des sous-classes des trois classes suivantes :

```
Imx2duc, Imx3dsl, Imx2dsf.
```

Dans ce cas, seul le main conserve le switch global, par contre il existe plus que trois opérateur voire un seul par l'utilisation des templates C++ :

REGIS.CLOUARD@GREYC.ISMRA.FR

```
template <typename T1, typename T2>
Errc Operateur(Imx3d<T1> &ims,Imx3d<T2> &imd,Short nbvois)
{
   ...
   for (int b=0; b<ims.Bands(); b++){ // Pour chaque bande...
        ...imd[b][z][y][x] = (T2)f(ims[b][z][y][x]);
   }
   ...
   return(SUCCESS);
}

#define USAGE  "USAGE : %s [im_in|-] [im_out|-]"
#define PARC   0    // Number of parameters
#define FINC   1    // Number of input images
#define FOUTC  1    // Number of output images

int main(int argc, char *argv[])
{
   Pobject* mask;             // The mask.
   Pobject* objin[FINC+1];    // The input objects;
   Pobject* objs[FINC+1];     // The source objects masked by the mask.
   Pobject* objout[FOUTC+1];  // The output object.
   Pobject* objd[FOUTC+1];    // The result object of the execution.
   Float parv[PARC+1];        // The input parameters.

   ReadArgs(argc,argv,PARC,FINC,FOUTC,
            &mask,objin,objs,objout,objd,parv,USAGE);

  switch(objd[0]->Type()){
   case Po_Img2duc: {
      Img2duc* const ims=(Img2duc*)objs[0];
      objd[0]=new Img2duc(ims->Props());
      Img2duc* const imd=(Img2duc*)objd[0];

      result=Operateur(*ims,*imd,(Short)parv[0]);
      break;
   }
   case Po_Img2dsl: {
      Img2dsl* const ims=(Img2dsl*)objs[0];
      objd[0]=new Img2dsl(ims->Props());
      Img2dsl* const imd=(Img2dsl*)objd[0];

      result=Operateur(*ims,*imd,(Short)parv[0]);
      break;
   }
   return 0;
}
```

5.3 Utilisation du préprocesseur pour l'écriture générique

Pour simplifier l'écriture d'opérateur génériques, Pandore possède son propre macro-processeur. Ce préprocesseur est appelé par le Makefile pour convertir des fichiers portant l'extension `.cct` (du code C++ contenant des macros) en fichier d'extension `.cc` (du code C++ uniquement) avant compilation.

Ce préprocesseur ne reconnaît que les lignes débutant par `##'. Les lignes commençant par `##;' sont supprimées et peuvent servir à placer des commentaires.

REGIS.CLOUARD@GREYC.ISMRA.FR

5.3.1 *Squelette d'un programme générique*

Un fichier operateur.cct respecte plus ou moins le squelettte de programme ci-dessous.

```
/* -*- mode: c++; c-basic-offset: 3 -*-
 * @(#)operateur.cc      PANDORE           2004-07-01
 *
 * PANTHEON Project
 *
 * GREYC IMAGE
 * 6 Boulevard Maréchal Juin
 * F-14050 Caen Cedex
 *
 * This file is free software. You can use it, distribute it
 * and/or modify it. However, the entire risk to the quality
 * and performance of this program is with you.
 */

/*
 * (C)Creator - 2001-02-01
 * (R)Revisor - 2001-04-25
 */

/**
 * Comments on the operator.
 */

#include <pandore.h>

/*
 * Comments on the function.
 */
##begin Operateur(TYPE1, VOISS)
Errc Operateur(TYPE1 &ims, TYPE1 &imd, Short nbvois)
{
  <Code...>
     if (nbvois == VOISS)
       <code...>
    return(SUCCESS);
}
## append loadcases
   if (objs[0]->Type() == Po_$TYPE1) {
      TYPE1 *const ims=(TYPE1*)objs[0];
      objd[0]=new TYPE1(ims->Props());
      TYPE1 *const imd=(TYPE1*)objd[0];
      result=Operateur(*ims,*imd,(Short)parv[0]);
   } else
## end
##end

##;Genere Operateur pour toutes les Img1d.., Img2d.., Img3d..
##forall(Operateur,/Img.d/)

##;Genere Operateur pour les Reg2d et Reg3d
##forall(Operateur,/Reg[23]d/)

#ifdef MAIN
#define  USAGE "USAGE : %s connexity [-m mask] [im_in|-] [im_out|-]"
#define  PARC  1
#define  FINC  1
#define  FOUTC 1
#define  MASK  3

##main(PARC,FINC,FOUTC,MASK,USAGE)
#endif
```


REGIS.CLOUARD@GREYC.ISMRA.FR

5.3.2 *La partie code de l'opérateur*

La code générique est encadré par ## BEGIN et ##END.

```
##begin OPERATEUR(TYPE1, VOISS)
Errc Operateur(TYPE1 &ims, TYPE1 &imd, Short nbvois)
{
  <Code...>
     if (nbvois == VOISS)
       <code...>
    return(SUCCESS);
}

## append loadcases
   if (objs[0]->Type() == Po_$TYPE1) {
      TYPE1 *const ims=(TYPE1*)objs[0];
      objd[0]=new TYPE1(ims->Props());
      TYPE1 *const imd=(TYPE1*)objd[0];
      result=Operateur(*ims,*imd,(Short)parv[0]);
   } else
## end
##end
```

Elle est composée de deux parties, le code de la fonction opérateur elle-même et le code qui sera ajouté dans le main() pour orienter vers cette fonction lorsque l'entrée correspondra au type précisé (Po_$TYPE1).

5.3.3 *Génération des opérateurs en fonction de types choisis*

Pour générer la liste des opérateurs en fonction des types choisis, il suffit d'appeler la macro définie en donnant les valeurs de type :

```
##OPERATEUR(type1, type2, type3 , ...)
```

La macro forall permet de faire une génération automatique à partir d'un filtre des types. La syntaxe pour lister les types acceptés reprend celle des expressions régulières:

```
#forall(Operateur,/type1/, /type2/, /type3/, ...)
```

Par exemple:

```
#forall(Operateur,/Imc[23]duc/,/Img2ds/)
```

permet de générer l'opérateur pour les combinaisons :

```
##Operateur(imc2duc, Img2dsl)
##Operateur(imc2duc, Img2dsf)
##Operateur(imc3duc, Img2dsl)
##Operateur(imc3duc, Img2dsf)
```

5.3.4 *Les arguments de la macro begin*

Ces arguments permettent de passer à l'opérateur le voisinage, la dimension, le type de point et les boucles de parcours d'images correspondant à la dimension de

REGIS.CLOUARD@GREYC.ISMRA.FR

l'image (1d, 2d ou 3d). Ces arguments sont passé avec le ##begin lorsqu'ils sont nécessaires.

```
##BEGIN Operateur<TYPE,...., VOISL, POINT, DIM, LOOP, VARS, ACC ... >
```

Les voisinages (connexité): VOIS

VOISS: retourne la valeur du voisinage le plus petit (2 en 1D, 4 en 2D et 6 en 3D)

VOISL: retourne la valeur du voisinage le plus grand (2 en 1D, 8 en 2D et 26 en 3D)

Les points: POINT

POINT : retourne le type de point (Point1d en 1D, Point2d en 2d et Point3d en 3D).

Les dimensions: DIM

DIM : retourne le type de point (Dimension1d en 1D, Dimension2d en 2d et Dimensiont3d en 3D).

Le parcours d'image génériques: LOOP (VARS et ACC)

La macro LOOP et ses variations permet de parcourir une image pixel par pixel quelque soit la dimension de l'image. Le parcours peut se faire à partir de variables (x,y,z) ou de points (Point1d, Point2d, Point3d).

1. LOOP: Parcours avec des variables de boucle VARS et l'accès par ACC.

Exemple : parcours de toute l'image ims en balayage vidéo.

```
##BEGIN Operateur<TYPE,...., LOOP, VARS, ACC>
...
int VARS
##LOOP(ims,VARS)
{
   std::cout << (float)ims[ACC] << std::endl;
}
...
```

est équivalent en 2d à :

```
int x,y;

for (y=0; y<ims.Height(); y++)
   for (x=0; x<ims.Width(); x++)
   {
        std::cout << (float)ims[y][x] << std::endl;
   }
```

2. LOOPIN: Parcours inverse

Exemple : parcours de toute l'image ims en balayage vidéo inverse.

REGIS.CLOUARD@GREYC.ISMRA.FR

```
##BEGIN Operateur<TYPE,...., LOOPIN, VARS, ACC>
...
int VARS
##LOOPIN(ims,VARS)
{
   std::cout << (float)ims[ACC] << std::endl;
}
```

est équivalent à :

```
int x,y;

for (y=ims.Height()-1; y>=0 y--)
   for (x<ims.Width()-1; x>=0; x--)
   {
        std::cout << (float)ims[y][x] << std::endl;
   }
```

3. LOOPB: Parcours intérieur en balayage vidéo.

Exemple : parcours de l'image ims en laissant un bord de 5 en balayage vidéo.

```
##BEGIN Operateur<TYPE,...., LOOPB, VARS, ACC>
...
int VARS
##LOOPB(ims,VARS,5)
{
   std::cout << (float)ims[ACC] << std::endl;
}
```

est équivalent à :

```
int x,y;

for (y=5; y<ims.Height()-5; y++)
   for (x=5; x<ims.Width()-5; x++)
   {
        std::cout << (float)ims[y][x] << std::endl;
   }
```

4. LOOPINB: Parcours intérieur en balayage vidéo inverse.

Exemple : parcours de l'image ims en laissant un bord de 5 en balayage vidéo inverse.

```
##BEGIN Operateur<TYPE,...., LOOPINB, VARS, ACC>
...
int VARS
##LOOPINB(ims,VARS,5)
{
   std::cout << (float)ims[ACC] << std::endl;
}
```

est équivalent à :

```
int x,y;

for (y=ims.Height()-5-1; y>= 5; y--)
   for (x=ims.Width()-5-1; x>= 5; x--)
   {
        std::cout << (float)ims[y][x] << std::endl;
   }
```

5. LOOPP: Parcours avec un point déclaré par POINT.

Exemple : parcours de toute l'image ims en balayage vidéo.

REGIS.CLOUARD@GREYC.ISMRA.FR

```
##BEGIN Operateur<TYPE,...., LOOPP, POINT>
...
POINT p;
##LOOPP(ims,p)
{
   std::cout << (float)ims[p] << std::endl;
}
```

est équivalent en 2d à :

```
Point2d p;

for (p.y=0; p.y<ims.Height(); p.y++)
   for (p.x=0; p.x<ims.Width(); p.x++)
   {
        std::cout << (float)ims[p] << std::endl;
   }
```

6. LOOPPIN : Parcours inverse

Exemple : parcours de toute l'image ims en balayage vidéo inverse.

```
##BEGIN Operateur<TYPE,...., LOOPPIN, POINT>
...
POINT p;
##LOOPPIN(ims,p)
{
   std::cout << (float)ims[p] << std::endl;
}
```

est équivalent à :

```
Point2d p;

for (p.y=ims.Height()-1; p.y>=0 p.y--)
   for (p.x<ims.Width()-1; p.x>=0; p.x--)
   {
        std::cout << (float)ims[p] << std::endl;
   }
```

7. LOOPPB: Parcours intérieur en balayage vidéo.

Exemple : parcours de l'image ims en laissant un bord de 5 en balayage vidéo.

```
##BEGIN Operateur<TYPE,...., LOOPPB, POINT>
...
POINT p;
##LOOPPB(ims,p,5)
{
   std::cout << (float)ims[p] << std::endl;
}
```

est équivalent à :

```
Point2d p;

for (p.y=5; p.y<ims.Height()-5; p.y++)
   for (p.x=5; p.x<ims.Width()-5; p.x++)
   {
        std::cout << (float)ims[p] << std::endl;
   }
```

8. LOOPPINB: Parcours intérieur en balayage vidéo inverse.

Exemple : parcours de l'image ims en laissant un bord de 5 en balayage vidéo inverse.

REGIS.CLOUARD@GREYC.ISMRA.FR

```
##BEGIN Operateur<TYPE,...., LOOPPINB, POINT>
...
POINT p;
##LOOPPINB(ims,p,5)
{
   std::cout << (float)ims[p] << std::endl;
}
```

est équivalent à :

```
Point2d p;

for (p.y=ims.Height()-5-1; p.y>= 5; p.y--)
   for (p.x=ims.Width()-5-1; p.x>= 5; p.x--)
   {
        std::cout << (float)ims[p] << std::endl;
   }
```

5.3.5 Le main() générique

La déclaration du main contient uniquement la définition des constantes donnant le nombre de paramètre, le nombre de fichiers d'entrée de sortie et l'option de masquage suivi de la macro ##main...

```
#ifdef MAIN
#define USAGE   "USAGE : %s connexite [-m mask] [im_in|-][im_out|-]"
#define PARC    1
#define FINC    1
#define FOUTC   1
#define MASK    3

##main(PARC,FINC,FOUTC,MASK,USAGE)
#endif
```

Au main() sera ajoutés tous les cas définis par la macro forall.

5.4 Détail du préprocesseur de PANDORE

Définition et utilisation d'une macro

La paire ##begin/ ##end permet de définir des macros. Le texte suivant

```
##; texte decrivant la macro et n'apparaissant pas dans le r'ésultat
##begin une_macro
  Une macro peut contenir
  plusieurs lignes.
##end
```

définit la macro une_macro. L'appel de cette macro se fait simplement en écrivant ##une_macro. Ainsi

```
Texte avant.
##une_macro
Texte après.
```

générerait les quatre lignes suivantes :

REGIS.CLOUARD@GREYC.ISMRA.FR

```
Texte avant.
  Une macro peut contenir
  plusieurs lignes.
Texte après.
```

Les macros peuvent prendre des paramètres, il suffit de les lister après le nom lors de la définition.

```
##begin citation(BLOC)
------
BLOC.
------
##end
##citation(Exemple)
```

donne

```
------
Exemple.
------
```

Les appels de macros peuvent aussi être composés. Avec les définitions précédentes,

```
##citation(une_macro)
```

produirait

```
------
  Une macro peut contenir
  plusieurs lignes.
.
------
```

Dans l'avant dernier exemple (`##citation(Exemple)`), Exemple ne correspondait pas au nom d'une macro et n'avait donc pas été évalué. Ce n'est pas le cas ici avec `une_macro` qui est évaluée avant l'appel de `citation`.

En revanche,

```
##citation(citation)
```

donne

```
------
citation.
------
```

car la macro `citation` intérieure ne peut pas être évaluée sans arguments.

5.4.1 Ajout de texte ou de paramètres

Il peut arriver que l'on souhaite augmenter une macro ou lui ajouter un paramètre.

REGIS.CLOUARD@GREYC.ISMRA.FR

Il suffit alors d'utiliser une paire ##append/ ##end. L'exemple suivant ajoute une ligne et un paramètre à la macro une_macro.

```
##append une_macro Une
  Une de plus.
##end
##une_macro(1)
```

donne

```
  1 macro peut contenir
  plusieurs lignes.
  1 de plus.
```

##append est équivalant à ##begin lorsque la macro n'existe pas (autrement ##begin permet de redéfinir une macro existante).

Particularités

Retours à la ligne

Les macros dont le nom commence par un point d'exclamation s'évaluent sans laisser aucun retour à la ligne :

```
##begin !une_macro
  Une macro peut contenir
  plusieurs lignes.
##end
##citation(!une_macro)
```

devient

```
------
  Une macro peut contenir  plusieurs lignes..
------
```

De même, la macro prédéfinie dnl permet de supprimer tous les sauts de ligne (Discard New Line) des arguments qui lui sont passés et concatène le résultat.

```
##dnl(citation("Hello world"),citation(Hello World))
```

générerait, sans passer à la ligne :

```
------"Hello world".------------Hello World.------
```

dlnl agit de même, mais supprime uniquement le dernier (last) saut de ligne :

```
##citation(dlnl(une_macro))
```

donne

```
------
  Une macro peut contenir
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
   plusieurs lignes..
  ------
```

Coupures de mots

Lorsqu'une macro est évaluée, ses arguments sont comparés à tous les mots de sa définition pour y être substitués. Un mot est une séquence de lettres, chiffres ou `_'. Il arrive que l'on souhaite substituer les arguments sur des parties de mot, il suffit alors de simuler une coupure dans le mot avec `$'.

```
##begin test(bc)
abc
a$bc
a$bcd
a$bc$d
##end
##test(XX)
```

produirait

```
abc
aXX
a$bcd
aXXd
```

Évaluation récursive

Le contenu d'une macro, n'est jamais interprété avant l'évaluation de la macro, même s'il contient des ##. Ainsi l'évaluation d'une macro peut entraîner la définition ou l'évaluation d'autres macros, comme dans cet exemple :

```
##begin IncFactory(VarName,IncValue)
## begin Inc_$VarName
VarName += IncValue;
## end
##end
##;------------------
##IncFactory(y,1)
##IncFactory(x,7)
##Inc_y
##Inc_x
```

donne

```
y += 1;
x += 7;
```

5.4.2 Utilisation avec Pandore

Pour montrer l'utilisation de ce préprocesseur avec Pandore, nous allons prendre un opérateur existant, celui de l'addition, et le réécrire progressivement avec ce

REGIS.CLOUARD@GREYC.ISMRA.FR

préprocesseur.

Chaque opérateur de Pandore est composé de deux parties, dans un premier temps, les fonctions de traitement correspondant à tous les types supportés sont écrites, ensuite un main doit être écrit de façon à appeler la bonne fonction selon le types des paramètres passés à l'opérateur.

Les fonctions

L'opérateur Add de Pandore contient la méthode Add en quatre exemplaires :

```
Errc Add(Img2duc &ims1, Img2duc &ims2, Img2duc &imd)
{
   register int y,x;

   for (y=0; y<ims1.nrow; y++)
   for (x=0; x<ims1.ncol; x++)
        imd[y][x] = (Uchar)((ims1[y][x] + ims2[y][x])/2);

   return SUCCESS;
}

Errc Add(Img2duc &ims1, Img2dsl &ims2, Img2dsl &imd)
{
   register int y,x;

   for (y=0; y<ims1.nrow; y++)
   for (x=0; x<ims1.ncol; x++)
        imd[y][x] = (Slong)((ims1[y][x] + ims2[y][x])/2);

   return SUCCESS;
}

Errc Add(Img2dsl &ims1, Img2dsl &ims2, Img2dsl &imd)
{
   register int y,x;

   for (y=0; y<ims1.nrow; y++)
   for (x=0; x<ims1.ncol; x++)
        imd[y][x] = (Slong)((ims1[y][x] + ims2[y][x])/2);

   return SUCCESS;
}

Errc Add(Graph2d &gs1,Graph2d &gs2,Graph2d &gd)
{
   Ushort i;

   gd.Init(gs1);

   for (i=1;i<=gd.size;i++)
        gd[i]->value=(float)((gs1[i]->value+gs2[i]->value)/2);

   return SUCCESS;
}
```


REGIS.CLOUARD@GREYC.ISMRA.FR

Laissons les graphes de côté pour le moment. Nous pouvons regrouper les trois premières routines en une macro, il suffit de la paramétrer par les éléments variables.

```
##begin Add(IMG1,IMG2,TYPE)
Errc Add(IMG1 &ims1, IMG2 &ims2, IMG2 &imd)
{
   register int y,x;

   for (y=0; y<ims1.nrow; y++)
   for (x=0; x<ims1.ncol; x++)
        imd[y][x] = (TYPE)((ims1[y][x] + ims2[y][x])/2);

   return SUCCESS;
}
##end
```

Les trois routines peuvent alors être générés avec

```
##Add(Img2duc,Img2duc,Uchar)
##Add(Img2duc,Img2dsl,Long)
##Add(Img2dsl,Img2dsl,Long)
```

À ce point, nous nous rendons compte que TYPE peut-être déduit de la valeur de IMG2, il est inutile d'en faire un paramètre de la macro :

```
##begin Add(IMG1,IMG2)
Errc Add(IMG1 &ims1, IMG2 &ims2, IMG2 &imd)
{
   register int y,x;

   for (y=0; y<ims1.nrow; y++)
   for (x=0; x<ims1.ncol; x++)
        imd[y][x] = (IMG2::ValueType)((ims1[y][x] + ims2[y][x])/2);

   return(SUCCESS);
}
##end
```

De même, l'image résultat ne devrait par être nécessairement du type de la seconde opérande, mais plutôt choisie en fonction des valeurs de IMG1 et IMG2 :

```
##begin Add(IMG1,IMG2)
Errc Add(IMG1 &ims1, IMG2 &ims2, Select<IMG1,IMG2>::LargestSigned &imd)
{
   register int y,x;

   for (y=0; y<ims1.nrow; y++)
   for (x=0; x<ims1.ncol; x++)
        imd[y][x]=(Select<IMG1,IMG2>::LargestSigned::ValueType)
                                              ((ims1[y][x]+ims2[y][x])/2);

   return(SUCCESS);
}
##end
```

Ils est alors possible d'instancier une quatrième version de la routine

```
##Add(Img2dsl,Img2duc)
```


REGIS.CLOUARD@GREYC.ISMRA.FR

Si nous souhaitons maintenant disposer des versions 3D de ces opérations, plusieurs éléments doivent aussi varier :

1. la déclaration des variables ;
2. la boucle sur les points de l'image ;
3. les accès aux valeurs de ces points.

Il existe des macros (définies dans `tools/macros/pand_macros`) utiles dans ces cas là :

```
##begin !var2def
y,x
##end
##begin !acc2def
y][x
##end
##begin for2d(IMG,Y,X)
        for (Y=0;Y<IMG.nrow;Y++)
        for (X=0;X<IMG.ncol;X++)
##end
```

Les mêmes existent en 3D.

```
##begin Add(IMG1,IMG2,TYPE,VARS,LOOP,ACC)
Errc Add(IMG1 &ims1, IMG2 &ims2, Select<IMG1,IMG2>::LargestSigned &imd)
{
   register int VARS;

##LOOP(ims1,VARS)
         imd[ACC] = (Select<IMG1,IMG2>::LargestSigned::ValueType)
                                                ((ims1[ACC]+ims2[ACC])/2);

   return(SUCCESS);
}
##end
```

L'instanciation des quatre routines s'écrit maintenant

```
##Add(Img2duc,Img2duc,!var2def,for2d,!acc2def)
##Add(Img2duc,Img2dsl,!var2def,for2d,!acc2def)
##Add(Img2dsl,Img2duc,!var2def,for2d,!acc2def)
##Add(Img2dsl,Img2dsl,!var2def,for2d,!acc2def)
```

La ligne `##LOOP(ims1,VARS)` sera remplacée par `##for2d(ims1,y,x)` avant d'être évaluée puis à nouveau remplacée par la double boucle attendue.

`!var2def` et `!acc2def` utilisent les variables `y` et `x`, si cela n'avait pas convenu (par exemple si ces variables sont déjà utilisées ailleurs), il aurait fallu utiliser `!vard2d(x,y)` à la place.

Les version 3D des mêmes routines sont instanciables naturellement avec

```
##Add(Img3duc,Img3duc,!var3def,for3d,!acc3def)
##Add(Img3duc,Img3dsl,!var3def,for3d,!acc3def)
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
##Add(Img3dsl,Img3duc,!var3def,for3d,!acc3def)
##Add(Img3dsl,Img3dsl,!var3def,for3d,!acc3def)
```

Une macro prédéfinie (dans `tools/macros/init.pl`) permet de générer ces quelques lignes en une seule. La ligne

```
##forall(Add,Img3du,Img3du)
```

Est strictement équivalente au bloc de quatre ##Add précédant. Le mot `Img3du` signifie *l'ensemble des types dont le nom contient* `Img3du`. Ici, deux ensemble sont donnés, et sont utilisé pour générer les instanciations de `Add`. Les paramètres `VARS`, `LOOP` et `ACC` sont remplacés automatiquement (la macro `forall` sera vue plus en détail par la suite). En écrivant :

```
##forall(Add,Img2d,Img2d)
##forall(Add,Img3d,Img3d)
```

Nous générons des opérateurs d'addition pour tous les couples d'images de même dimension (il existe 5 types pour chaque dimension, donc ces deux lignes génèrent 50 fonctions !).

La routine main()

Voici le `main` original de l'opérateur `Add`.

```
#ifdef MAIN

/*
 * Modify only the following constants, and the function call.
 */
#define  USAGE "USAGE : %s [-m mask] [im_in1|-] [im_in2|-] [im_out|-]"
#define  PARC  0
#define  FINC  2
#define  FOUTC 1
#define  MASK 1

int
main(int argc,char* argv[])
{
   Errc  result;             // The result code of the execution.
   Pobject* mask;            // The region mask.
   Pobject* objin[FINC+1];   // The input objects;
   Pobject* objs[FINC+1];    // The source objects masked by the mask.
   Pobject* objout[FOUTC+1]; // The output object.
   Pobject* objd[FOUTC+1];   // The result object of the execution.
   Float  parv[PARC+1];      // The input parameters.

   ReadArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,
            objout,objd,parv,USAGE,MASK);

   switch(objs[0]->Type()){
   case Po_Img2duc :{
        Img2duc* const ims1=(Img2duc*)objs[0];
        switch(objs[1]->Type()){
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
            case Po_Img2duc :{
                  Img2duc* const ims2=(Img2duc*)objs[1];
                  objd[0]=new Img2duc(ims1->nrow,ims1->ncol);
                  Img2duc* const imd=(Img2duc*)objd[0];

                  result=Add(*ims1,*ims2,*imd);
            }break;
            case Po_Img2dsl :{
                  Img2dsl* const ims2=(Img2dsl*)objs[1];
                  objd[0]=new Img2dsl(ims1->nrow,ims1->ncol);
                  Img2dsl* const imd=(Img2dsl*)objd[0];

                  result=Add(*ims1,*ims2,*imd);
            }break;
            default    :
                  fprintf(stderr,"ERREUR:Operandes incompatibles\n");
                  result = -1;
            }
      }
      break;
      case Po_Img2dsl :{
            Img2dsl* const ims1=(Img2dsl*)objs[0];
            switch(objs[1]->Type()){
            case Po_Img2duc :{
                  Img2duc* const ims2=(Img2duc*)objs[1];
                  objd[0]=new Img2dsl(ims1->nrow,ims1->ncol);
                  Img2dsl* const imd=(Img2dsl*)objd[0];

                  result=Add(*ims2,*ims1,*imd);
            }break;
            case Po_Img2dsl :{
                  Img2dsl* const ims2=(Img2dsl*)objs[1];
                  objd[0]=new Img2dsl(ims1->nrow,ims1->ncol);
                  Img2dsl* const imd=(Img2dsl*)objd[0];

                  result=Add(*ims1,*ims2,*imd);
            }break;
            default    :
                  fprintf(stderr,"ERREUR:Operandes incompatibles\n");
                  result = -1;
            }
      }break;
      case Po_Graph :{
            switch(objs[1]->Type()){
            case Po_Graph :{
                  Graph* const gs1=(Graph*)objs[0];
                  Graph* const gs2=(Graph*)objs[1];

                  objd[0]=new Graph(gs1->size);
                  Graph* const gd=(Graph*)objd[0];

                  result=Add(*gs1,*gs2,*gd);
            }break;
            default    :
                  fprintf(stderr,"ERREUR:Operandes incompatibles\n");
                  result = -1;
            }
      }break;
      default    :
            fprintf(stderr,"ERREUR:Operateur   non   encore   defini   pour   cet
objet\n");
            result = -1;

      }
      WriteArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,objout,objd);

      Exit(result);
  }
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
  #endif
```

L'addition ne gérait que 2 types d'images. Maintenant elle en gère 10. Pour être complet, il faudrait écrire 50 cas pour le switch. L'idée est d'utiliser ##append pour augmenter, à chaque instanciation de Add, une macro contenant tous les cas du main :

```
  ##begin Add(IMG1, IMG2, VARS, LOOP, ACC)
  Errc Add(IMG1 &ims1, IMG2 &ims2, Select<IMG1,IMG2>::LargestSigned &imd)
  {
     register int         VARS;

  ##      LOOP(ims1,VARS)
     imd[ACC]=(Select<IMG1,IMG2>::LargestSigned::ValueType)((ims1[ACC]+
                                                             ims2[ACC])/2);

     return(SUCCESS);
  }

  ## append loadcases
          if ( ( objs[0]->Type() == Po_$IMG1 ) &&
          ( objs[1]->Type() == Po_$IMG2 ) ) {
             IMG1* const ims1=(IMG1*)objs[0];
           IMG2* const ims2=(IMG2*)objs[1];
           objd[0]=new Select<IMG1,IMG2>::LargestSigned(ims1->Size());
           Select<IMG1,IMG2>::Signed*  const  imd=(Select<IMG1,IMG2>::Signed*)
objd[0];
           result=Add(*ims1,*ims2,*imd);
     } else
  ## end
  ##end
```

Pour la copie des dimensions de l'image source, on a utilisé la méthode Size(), ce qui permet une écriture identique quelque soit la dimension. A chaque instanciation de la macro Add, la macro loadcases de voit ajouter le cas de chargement correspondant. Le main se réduit alors à :

```
  #ifdef MAIN

  #define USAGE  "USAGE : %s [mask] [im_in1|-] [im_in2|-] [im_out|-]"
  #define PARC   0
  #define FINC   2
  #define FOUTC  1

  int main(int argc,char* argv[])
  {
     Errc   result;             // The result code of the execution.
     Pobject* mask;             // The region mask.
     Pobject* objin[FINC+1];    // The input objects;
     Pobject* objs[FINC+1];     // The source objects masked by the mask.
     Pobject* objout[FOUTC+1];  // The ouput object.
     Pobject* objd[FOUTC+1];    // The result object of the execution.
     Float  parv[PARC+1];       // The input parameters.

     ReadArgs(argc,argv,PARC,FINC,FOUTC,&mask,
                objin,objs,objout,objd,parv,USAGE);
  ##       loadcases
     {
           fprintf(stderr,"ERREUR:Operateur   non   encore   defini   pour   cet
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
objet\n");
          result = -1;
     }
     WriteArgs(argc,argv,PARC,FINC,FOUTC,&mask,objin,objs,objout,objd);

     Exit(result);
  }

  #endif
```

Ce qui s'abrège encore plus en utilisant la macro `main` définie dans `tools/macros/pand_macros`:

```
  ##main(0,2,1,1,"USAGE : %s [mask] [im_in1|-] [im_in2|-] [im_out|-]")
```

Finalement, l'opérateur `Add` peut-être écrit, complètement sous la forme suivante :

```
  #include "pandore.h"

  ##; ---- les images
  ##begin Add(IMG1, IMG2, VARS, LOOP, ACC)
  /*
   * Consult    :    Point/point inconditionnellement.
   * Fonction   :    (pix1 + pix2)/ 2;
   */
  Errc Add(IMG1 &ims1, IMG2 &ims2, Select<IMG1,IMG2>::Signed &imd)
  {
     register int          VARS;

  ##      LOOP(ims1,VARS)
          imd[ACC]=(Select<IMG1,IMG2>::Signed::ValueType)((ims1[ACC]+ims2
[ACC])/2);
     returnSUCCESS;
  }

  ## append loadcases
          if ( ( objs[0]->Type() == Po_$IMG1 ) &&
          ( objs[1]->Type() == Po_$IMG2 ) ) {
             IMG1* const ims1=(IMG1*)objs[0];
           IMG2* const ims2=(IMG2*)objs[1];
           objd[0]=new Select<IMG1,IMG2>::Signed(ims1->Size());
           Select<IMG1,IMG2>::Signed*const    imd=(Select<IMG1,IMG2>::Signed*)
objd[0];         result=Add(*ims1,*ims2,*imd);
     } else
  ## end
  ##end

  ##; ---- les graphes
  ##begin AddGraph(TYPE)
  /*
   * Affecte la difference des valeurs des deux graphes.
   */
  Errc Add(TYPE &gs1,TYPE &gs2,TYPE &gd)
  {
     Ushort     i;
     gd.Init(gs1);

     for (i=1;i<=gd.size;i++)
          gd[i]->attr=((gs1[i]->attr+gs2[i]->attr)/2);

     return(SUCCESS);
  }
  ## append loadcases
          if ( ( objs[0]->Type() == Po_$TYPE ) &&
          ( objs[1]->Type() == Po_$TYPE ) ) {
             TYPE* const gs1=(TYPE*)objs[0];
```


REGIS.CLOUARD@GREYC.ISMRA.FR

```
            TYPE* const gs2=(TYPE*)objs[1];
            objd[0]=new TYPE(gs1->Size());
            TYPE* const gd=(TYPE*)objd[0];
            result=Add(*gs1,*gs2,*gd);
      } else
## end
##end

##forall(Add,Img2d,Img2d)
##forall(Add,Img3d,Img3d)
##forall(AddGraph,Graph)

#ifdef MAIN
##main(0,2,1,1,"USAGE : %s [-m mask] [im_in1|-] [im_in2|-] [im_out|-]")
#endif
```

Avec l'avantage que si plus tard de nouveaux types sont ajoutés dans les ensembles `Img2d`, `Img3d` ou `Graph`, il ne serait pas nécessaire de toucher à nouveau au source de cet opérateur.

La macro ##forall

`forall` est un macro qui prend en paramètres un nom de macro et une liste d'ensembles (E_1, E_2, ..., E_n). Parmi les arguments de la macro passée, il faut ensuite distinguer les arguments connus des inconnus. Un argument est connu lorsque sont nom est dans la première colonne du tableau suivant :

	1d	2d	3d	Description
ACC	`!acc1def`	`!acc2def`	`!acc3def`	Utilisation : ims[ACC]
VARS	`!var1def`	`!var2def`	`!var3def`	Variable de boucle
LOOP	`for1d`	`for2d`	`for3d`	Boucle sur indices [0..max]
LOOP	`for1dp`	`for2dp`	`for3dp`	Boucle sur point
LOOPIN	`for1din`	`for2din`	`for3din`	Boucle inversée sur indices
LOOPPIN	`for1dpin`	`for2dpin`	`for3dpin`	Boucle inversée sur point
LOOPB	`for1db`	`for2db`	`for3db`	Boucle intérieure sur indices [n..max-n].
LOOPPB	`for1dpb`	`for2dpb`	`for3dpb`	Boucle intérieure sur point
LOOPINB	`for1dinb`	`for2dinb`	`for3dinb`	Boucle intérieure inversée sur indices
LOOPPINB	`for1dpinb`	`for2dpinb`	`for3dpinb`	Boucle intérieure inversée sur point
POINT	`Point1d`	`Point2d`	`Point3d`	Type de point
DIM	`1d`	`2d`	`3d`	
VOISS	`2`	`4`	`6`	Nombre de voisins en connexité minimale
VOISL	`2`	`8`	`26`	Nombre de voisins en connexité maximale

Si la macro passée à `forall` a la forme `m(i1,c1,i2,c2,c3)` où les i_j sont des arguments inconnus et les c_j des arguments connus, `forall` attend alors deux ensembles E_1 et E_2 (autant que d'inconnus) dont les éléments servirons de valeurs

REGIS.CLOUARD@GREYC.ISMRA.FR

pour i_1 et i_2. Les arguments connus seront eux initialisés à partir du dernier inconnu rencontré (si le dernier argument inconnu à été remplacé par le type d'une image2d alors `POINT` sera remplacé par `Point2d`).

Par exemple si `m` est paramétré par `(IMG1,VAR,IMG2,LOOP,VOISL)`, alors la ligne suivante :

```
##forall(m,Img2du,Img.duc)
```

(où `Img2du` désigne l'ensemble { `Img2duc`, `Img2dsl` }, et `Img.duc` l'ensemble { `Img2duc`, `Img3duc` })provoquera l'évaluation de :

```
##m(Img2duc,!var2def,Imd2duc,for2d,8)
##m(Img2duc,!var2def,Img3duc,for3d,26)
##m(Img2dsl,!var2def,Img2duc,for2d,8)
##m(Img2dsl,!var2def,Img3dcs,for3d,26)
```

Les noms des ensembles sont en réalité des expressions régulières qui sont comparées à tous les types existants (définis dans `include/main.h`). Écrire `##forall(Add,g2d,g2d)` plutôt que `##forall(Add,Img2d,Img2d)` permet de travailler aussi sur le type `Reg2d`.

Lorsqu'aucun type n'est valable pour l'expression, elle est utilisée en paramètre de la macro. Ceci permet passer autre chose que des types aux arguments inconnus de la macro.

REGIS.CLOUARD@GREYC.ISMRA.FR

Index

REGIS.CLOUARD@GREYC.ISMRA.FR

REGIS.CLOUARD@GREYC.ISMRA.FR