PANDORE
Handbook for version 6.6

GREYC - IMAGE laboratory
University of Caen - ENSICAEN

June 4, 2014

Contents

1 Introduction

1.1 Programming Language

The Pandore programming environment is based on the object-oriented programming language C++ and thus takes benefit from its modelling capability, its portability, its large audience and the efficiency of its code.

The image processing concepts are represented as a C++ classes with attributes and member functions. All of these classes are stored within a specific namespace called `pandore`. Therefore, any Pandore files must includes the Pandore header file and uses the related namespace:

```
#include <pandore.h>
using namespace pandore;
```

1.2 Data Types

The Pandore programming environment uses all C++ concepts and first of all the primitive base types such as char, short, long, float, double, unsigned char, unsigned short or unsigned long.

However, Pandore redefines those primitive base types in order to increase the portability. Each base type is redefined so as to always keep the same size whatever the machine word size is. They are renamed by reusing the name of the C type except that the first letter is uppercase and a U is added to unsigned type. For example, long int is redefined as Long and is exactly 32 bits and Ulong redefines the unsigned long int.

One supplementary type `Errc` is defined to encompass all the previous primitive base types. It means that a variable of this type can be set with any of the base type value. Such type is generally used as the return value of the operator function.

=> See **Primitive Base Types**(p. 2).

1.3 Object Types

All the image processing concepts are defined in the Pandore environment either as a basic object or as a compound object. They are all represented by C++ classes.

The basic objects are:

- Dimensions (Dimension1d, Dimension2d, Dimension3d);
- Points (Point1d, Point2d, Point3d);
- Images (Imx1d, Imx2d, Imx3d, Imc2d, Imc3d, Img1d, Img2d, Img3d);
- Region maps (Reg1d, Reg2d, Reg3d).

The compound objects are:

- Collections (Collection);
- Graphs (Graph2d, Graph3d)

=> See **The Pandore objects**(p. 3).

1.4 Image Processing Operators

An image processing operator is defined as a traditional C++ function with formal parameters and not as a member function of the related object, in order to avoid to redefine the Pandore class each time a new operator is added.

A Pandore operator looks like the following template example:

```
Errc operator( const Img2duc &ims, Img2duc &imd, Long par1, Float par2 ) {
   <content>
   return SUCCESS;
}
```

The encoding takes advantage of some programming paradigms implemented as idioms. Basically:

- Images and region maps are considered as traditional arrays (1D vector, 2D matrix, 3D volume);
- Access to pixel neighbours uses predefined arrays (see **Accessing Image Pixel**(p. 15));
- The main() function is standardized (see **Operator Programming**(p. 30)).

Moreover, there exist some facilities to write generic operators by the way of a preprocessor (See **Preprocessing of operators**(p. 37)).

1.5 Image Processing Applications

An image processing application is a chain of operators. The C++ file is built by including all necessary operators using #include and by developing one or several functions which link operators in order to yield new Pandore objects by successive transformations of the input Pandore objects. In such a chain, output objects of the former operators are input objects of the latter.

=> See **Application Programming**(p. 36)

2 Types

2.1 Primitive Base Types

2.1.1 Primitive Types

Of course, Pandore uses all the C++ primitive base types (char, unsigned char, int, long, long long, long double ...). Nevertheless, some of these types had been redefined so as to be independent from the host platform and to produce portable code.

These types redefine the related C types so as to keep always the same size whatever is the machine word size. There are renamed by reusing the name of the C type except that the first letter is uppercase and a U is added to unsigned type.

The primitive types are:

- `Char` (or `int1`): a tiny integer (8 bits) [-128,127].
- `Uchar`(or `uint1`): a tiny unsigned integer (8 bits) [0,+255].
- `Short` (or `int2`): a short integer (16 bits) [-32768,+32767].

- `Ushort` (or `uint2`): a short unsigned integer (16 bits) [0,+65635].
- `Long` (or `int4`): a long integer (32 bits) [-2147483648,+2147483647].
- `Ulong` (or `uint4`): a long unsigned integer (32 bits) [0,+4294967295].
- `Float` (or `float4`): a real (32 bits) [-3.40e^{+38} ,+3.40e^{+38}] with a precision of 1.17e^{-38} ;
- `Double` (or `float8`): a long real (64 bits) [-1.79e^{+308} ,+1.79e^{+308}] with a precision of 2.22e^{-308} .

Warning:

- The types Int or Uint do not exist.
- Since these types are redefined so as to be independent from the host architecture, Long is not necessarily equivalent to the corresponding C++ long.

2.1.2 The Errc type

The type `Errc` allows the representation of any value of the predefined types, plus a new enumerated type: {SUCCESS, FAILURE}. This type is generally used as the return type for the operator function.

A variable of this type can be set with any value of the primitive type and can be changed at any time. For example:

```
Errc Operator() {
   if ( ... ) return (Char)0;
   if ( ... ) return 50.0F
   return SUCCESS;
}

void main() {
   Errc a;
   a=Operator();
   if (a==SUCCESS) ...
   else if (a>50.0F) ...;
}
```

2.2 The Pandore objects

2.2.1 The Basis Class Pobject

All the Pandore objects inherit from the basis class `Pobject`. A Pandore object is an object that can be loaded from a file and saved in a file. Pandore distinguishes six objects divided in four basic objects and two compound objects.

2.2.2 The Basic Objects

There exists only 4 types of basic Pandore objects:

1. The `point` (Point1d, Point2d, Point3d) represents a location in coordinate space specified in Long precision;
2. The `dimension` (Dimension1d, Dimension2d, Dimension3d) encapsulates size measures specified in Long precision;

3. The `image` (Imx1d, Img1d, Imx2d, Img2d, Imc2d, Imx3d, Img3d, Imc3d) is an array of pixels;
4. The `region` map (Reg1d, Reg2d, Reg3d) is an array of labels.

2.2.3 The Compound Objects

The compound objects are composition of several base types or basic Pandore objects or even compound Pandore objects. There exists 2 types of compound objects:

1. The `collection` (Collection) is a map of any base type or Pandore object referenced by a name;
2. The `graph` (Graph2d, Graph3d) is a graph of indexes, where an index refers to an element in an array. Any array is available which allows to create a graph of anything: a graph of points (index refers to element in an array of points), a graph of region maps (an array of region maps), a graph of integer, etc.

2.2.4 The Pandore Object Types

Each Pandore object is identified by a magic number and a name. The file `panfile.h` contains the enumerated list `Typobj` of all the magic numbers. Each item in the enumerated list is built with the name of the class and the prefix `Po_`. For instance `Po_Img2duc` is set with the magic number of the object `Img2duc` (a 2D gray levels image of bytes) or `Po_Point2d` corresponds to a `Point2d`.

The magic number and the name are accessible from the member function:

- `String Name()`: returns the name of the object (eg. Img2duc, Point2d);
- `Typobj Type()`: returns the magic number of the object (eg. Po_Img2duc, Po_Point2d).

For example:

```
Pobject *p = new Img2duc(122,256);
if (p->Type() == Po_Img2duc)
   std::cout << p->Name() << std::endl;
```

2.2.5 The Pandore Object Attributes

Each object is defined by its own list of attributes. For example, the class `Imc2duc` is defined by the dimension of the array and the color space or the class `Reg2d` is defined by the dimension and the higher label value. However, there exists a structure named `PobjectProps` which gathers all the attributes that are exchangeable between objects, such as the dimension, the number of bands, the color space, the higher label value for region maps, etc. This structure can be used to create an object with the properties of an another object.

For example, to build a region map with the same size than a given image, use:

```
Img2duc ims1(40,125);
Reg2d rgs(ims1.Props());
```

For example, to build a graph from a region map dimension and then an image from the resulted graph dimension use:

```
Reg2d rgs(120,256);
Graph2d *g=new Graph2d(rgs.Props());
Imc2duc ims2;
ims2.New(g.Props());
```

2.2.6 The Pandore Object member functions

A Pandore object has 4 categories of member functions:

1. Construction of the internal representation;
2. Data consultation;
3. File transfer;
4. Miscellaneous functions.

2.2.7 The Pandore Object Files

A Pandore object can be saved in or loaded from a normalized file (suffixed by ".pan" by pure convention). The file are composed of a common heading followed by an object specific heading and then the data.

Note:

Even if the files are binary files, loading is independent from the platform architecture. A Pandore object saved in MSB (Most Significant Bit first) platform can be loaded on LSB (Most Significant Bit first) platform and vice versa.

2.3 The Dimensions

2.3.1 Definition

A dimension encapsulates size measures specified in Long precision.

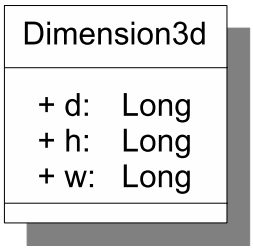

Figure 1: The class Dimension3d.

2.3.2 Types

There are three different classes of Dimension according to the dimension:

- Dimension1d: dimension in 1D;
- Dimension2d: dimension in 2D;
- Dimension3d: dimension in 3D.

2.3.3 Public Attributes

Dimensions are characterized by size measures:

- Long w: the length (for 1D, 2D and 3D dimensions);
- Long h: the height (for 2D and 3D dimensions);
- Long d: the depth (for 3D dimension).

They are read write attributes. It means that they are accessible directly without any member function. For example:

```
Dimension2d d;
d.w=12; d.h=20;
std::cout << d.h << std::endl;
```

2.3.4 Construction

A dimension can be created without any argument or with specified sizes, or a specified dimension. For example:

```
Dimension2d d1; // = Dimension2d d1(0,0)
d1.w=12; d1.h=21;
Dimension2d d2(10,24); // w=24; h=10.
Dimension2d *d3=new Dimension2d(d2); // Same size measures than d2.
```

2.3.5 Consultation

The basic arithmetic operators (+,-,*,/,==, !=, +=, -=, *=, /=) between dimension and constant or between dimensions has been redefined to handle dimensions. For example:

```
Dimension d1(12,13), *d2;
d2 = new Dimension(10,10);
if (d1==*d2) d1=(*d2)*5;
d2*=2;
```

2.3.6 File Transfer

To save a dimension in a file, just use:

```
Dimension d;
d.SaveFile("foobar.pan");
```

To load a dimension from a file, just use:

```
Dimension d;
d.LoadFile("foobar.pan");
```

Generally, a dimension is not saved directly in a file but by the means of a collection. To save and load a dimension in a collection, use:

```
Collection cold;
Dimension2d *d1=new Dimension2d(10,20),*d2;
cold.SETPOBJECT("bar",Dimension2d,d1);
d2=(Dimension2d*)cold.GETPOBJECT("bar",Dimension2d);
```

To save and load an array of dimensions in a collection, use:

```
Dimension2d **d3=new Dimension2d*[12], **d4;
for (int i=0; i<12; i++) d3[i]=new Dimension2d(i,i);
cold.SETPARRAY("foo",Dimension2d,p3,12);
d4=(Dimension2d**)cols.GETPARRAY("foo",Dimension2d);
std::cout << "Dimensions: " << d4[11]->w << "," << d4[11]->h << std::endl;
```

2.4 The Points

2.4.1 Definition

A point represents a location in a given coordinate space specified in Long precision.

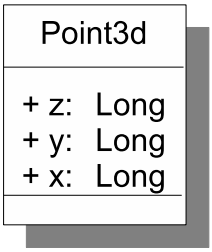

Figure 2: The class Point3d.

2.4.2 Types

There are three different classes of Point according to the dimension:

- Point1d : point in 1D;
- Point2d : point in 2D;
- Point3d : point in 3D.

2.4.3 Public Attributes

Points are characterized by the coordinates:

- Long x: the abscissa (for 1D, 2D and 3D);
- Long y: the ordinate (for 2D and 3D);
- Long z: the depth (for 3D).

They are read write attributes. It means that they are accessible directly without any member function. For example:

```
Point2d p;
p.x=5; p.y=12;
std::cout << p.y << std::endl;
```

2.4.4 Construction

A point can be created without any argument or with specified coordinates or a specified point, or a specified dimension. For example:

```
Point2d p1;     // = Point2d p1(0,0)
p1=1;           // p1.x = p1.y = 1
Point2d p2(12,24);
Dimension d(50,100);
Point2d p3(d);  // =Point(50,100)
Point2d *p4 = new Point2d(3);   // p4->x = p4->y = 3;
```

2.4.5 Consultation

The basic arithmetic operators (+,-,*,/,==, !=, +=, -=, *=, /=) between point and constant or between points has been redefined to handle point. For example:

```
Point2d p1(5,10), p2;
if (p1==p2 || p1==1) {
   p1+=p2; p2+=2+p1;
   p1*=p2; p2*=2;
}
```

2.4.6 File Transfer

To save a point in a file, just use:

```
Point p;
p.SaveFile("foobar.pan");
```

To load a point from a file, just use:

```
Point p;
p.LoadFile("foobar.pan");
```

Generally, a point is not saved directly in a file but by the way of a collection. To save and load a point in a collection, use:

```
Collection cold;
Point2d *p1=new Point2d(10,20),*p2;
cold.SETPOBJECT("bar",Point2d,p1);
p2=(Point2d*)cold.GETPOBJECT("bar",Point2d);
```

To save and load an array of points in a collection, use:

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

2.5 The Images

2.5.1 Definition

An image is considered as an array of pixels. A pixel stores a value or a vector of values at specified coordinates. The various types of image depend on the pixel type and the array dimension.

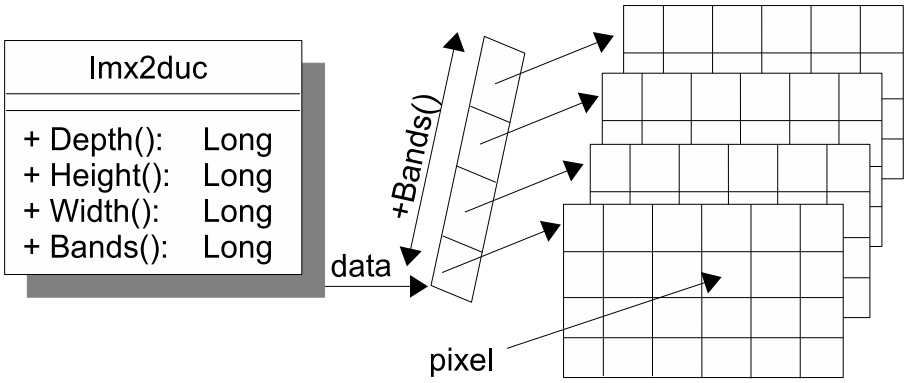

Figure 3: The class Imx2duc.

2.5.2 Types

There are 27 different types of images depending on the spectrum, the dimension and the pixel type. The name of a class is built from the following template:

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

It results from the concatenation of the prefix `Im` and each properties of the image with the following conventions:

1. the spectrum: [g,c,x]
   - g: gray level (= monospectral image);
   - c: color (= multispectral image with 3 bands);
   - x: multispectral (= multispectral image with n bands, n>=1).
2. the dimension: [1d,2d,3d]
   - 1D;
   - 2D;
   - 3D.
3. the type of pixel: [uc,sl,sf]
   - uc: unsigned char (8 bits, values in [0, 256]);
   - sl: signed long (32 bits, values in [-2147483648, 2147483647]);
   - sf: float (32 bits, values in [1.175494351e^{-38} , 3.402823466e^{+38}] with a precision of 1.4e-45).

For example:

- `Imx2duc:` multispectral, 2D image of tiny integers;
- `Img3dsf:` gray level, 3D image of reals;
- `Imc2dsl:` color, 2D image of long signed integer.

Note:

These images can also be named using the C++ template notation:

- `Imx2d<Uchar>`: multispectral, 2D image of tiny integers;
- `Img3d<Float>`: gray level, 3D image of reals;
- `Imc2d<Long>`: color, 2D image of long signed integer.

2.5.3 Hierarchy

Concretely, there are only three types of image:

- `Imx3duc` or `Imx3d<Uchar>`;
- `Imx3dsl` or `Imx3d<Long>`;
- `Imx3dsf` or `Imx3d<Float>`.

It means that all the other types are just a rewriting of these actual types. For example:

- gray level images are Imx3d with only 1 band;
- color images are Imx3d with 3 bands;
- 2D images are Imx3d with only 1 plane.

The major interest is that hierarchy can be used to write generic operator function, Instead of writing one function per Pandore type composition, only one template function had to be written. Such a function looks like the following example:

```
template <typename T>
Errc function( Imx3d<T> &ims, Imx3d<T> &imd, Float p1 ) {

   return SUCCESS;
}
```

This unique function can then be called with any image type. For example:

```
Errc result1, result2;
Img2duc ims1(256,256);
Img2duc ims2(256,256);
Img2dsf ims3(256,256);
Img2dsf ims4(256,256);

result1=function(ims1,ims2,5.0F);
result2=function(ims3,ims4,4.0F);
```

2.5.3.1 Public Attributes Images are characterized by the dimension, the number of bands and the color space. These attribute values are accessible by the way of member functions:

- `Long Width()`: returns the number of columns;
- `Long Height()`: returns the number of rows;
- `Long Depth()`: returns the number of planes;
- `Dimension2d Size()`: returns the size as a dimension;
- `Long Bands()`: returns the number of bands (eg. 3 for color image, 1 for gray level image);
- `int VectorSize()`: returns the size of the data vector for one band (ie. the number of pixels per bands);
- `PColorSpace ColorSpace([x])`: returns (if x is omitted) or set (if x is given) the color space of a color image among: {RGB, XYZ, LUV, LAB, HSL, AST, I1I2I3, LCH, WRY, RNGNBN, YCBCR, YCH1CH2, YIQ, YUV}.
- `PobjectProps Props()`: returns a structure with the image attribute values.

2.5.4 Construction

An image can be created with or without the related data. If the dimension is not given with the constructor, then the data is not allocated.

For example, the creation of a 2D gray level image of bytes:

```
Img2duc ims1(256,512);      // Data: array 256 rows x 512 columns.
Img2duc ims2(ims1.Size());  // Data: array size = ims1 array size.
Img2duc ims3(ims1.Props()); // Data: array size = ims1 array size.
Img2duc ims4;               // No data.
Img2duc *ims5=new Img2duc(256,512); // Data: array size 256x512.
Img2duc *ims6=new Img2duc;  // No data.
```

For example, the creation of a 2D color image of Float:

```
Imc2dsf ims7(256,512);  // 256 rows x 512 columns.
im7.ColorSpace(RGB);    // Set the color space to RGB
Imc2dsf *ims8=new Imc2dsf(256,512);
im8->ColorSpace(YUV);   // Set the color space to YUV
```

For example, the creation of a 3D multispectral image (5 bands) of Long:

```
Imx2dsl ims9(5,256,512); // 5 bands, 256 rows and 512 columns.
Imx2dsl ims10;           // No data.
```

2.5.5 Allocation

If the image has been created without data, the member function `New()` creates the data array -if data already exist they are first deleted. For example:

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

Warning:

The creation of the data does not initialize pixel values to 0. However, this can simply be done using:

```
Img2duc ims(256,256);
ims=0;
```

2.5.6 Allocation from a predefined array

It is possible to allocate the data from a predefined vector. In this case, the destruction is not done with the object. This assumes that the predefined vector have the correct size (eg., number of bands ∗ depth ∗ height ∗ width ∗sizeof(item) items); For example:

```
Float *d = (Float*)malloc(3*256*512*sizeof(Float));
Imx2dsf ims1(3,256,512,d);
Imx2dsf *ims2 = new Imx2dsf(3,256,512,d);
ims1.delete();
delete ims2;
free(d);
```

For example, to process a multispectral image as several gray level images, use:

```
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:PGaussianFiltering(*imii,*imio,2.0F);
  *imii=*imio;
  delete imii;
  delete imio;
}
```

For example, to process a 3D multispectral image as several 2D grayscale images, use:

```
Imx3duc *ims= new Imx3duc(3,12,200,256);
for(b=0;b<ims->Bands();b++){
  for(d=0;d<ims->Depth();d++){
    Img2duc *imii = new Img2duc(ims->Height(),ims->Width(),&(*ims)(b,d,0,0));
    Img2duc *imio = new Img2duc(imii->Props());
    gauss:PGaussianFiltering(*imii,*imio,2.0F);
    *imii=*imio;
    delete imii;
    delete imio;
  }
}
```

Warning:

Unfortunately, it is not possible to process directly a 3D multispectral image as several multispectral images since a 3D multipsctral image is not coded as several 2D multispectral 2D images. It is necessary to copy the required pixel from the 3D image to the 2D images and vice versa with the processed pixels.

2.5.7 Destruction

To delete the data without deleting the object itself, use the member function `Delete()`. For example:

```
Img2duc ims3(512,512);
ims3.Delete();
ims3.New(512,245);

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

2.5.8 Consultation

Access to pixel value can be done in three ways:

1. As a multidimensional array:
   - For gray level image only: ( )

     ```
     ims1(i,j)=15;
     (*ims4)(i,j)=15;
     Point2d pt(10,20);
     ims1[pt]=15;
     ```

   - For any image type: (band,)

     ```
     Imx2duc ims8, *ims9;
     ims8(b,i,j)=15; // b the band number.
     (*ims9)(b,i,j)=15;
     Point2d pt; ims8(b,pt)=15;
     ```

   - For color image only: X( ), Y( ), Z( )

     ```
     ims6.X(i,j)=15; ims6.Y(i,j)=10; ims6.Z(i,j)=14;
     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;
     ```

2. As a unique vector:
   - For any image: Vector() returns the beginning of the data vector.

     ```
     Img2dsl ims(256,512);

     for (Ulong *p=ims.Vector(); p<ims.Vector()+ims.VectorSize();)
        *p++ = 15;
     ```

   - For any image type: Vector(band) returns the beginning of the specified band vector.

     ```
     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++= 127;
     ```

   - For color image only: VectorX(), VectorY(), VectorZ() return the beginning of each color bands.

     ```
     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;
     ```

2.5.9 Setting Data Values

To copy the value of an image into an other image already allocated and with the same size, use operator =. For example:

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

To set a constant to all the pixel values use operator =. For example:

```
Imx3duc ims(3,12,123,245);
ims=127; // Set 127 to all the pixel values
```

Warning:

The pixel are directly copied. If the type of the source pixel is different from the destination, the values are casted using the C casting convention. For example:

```
Img2duc *imd1 = new Img2duc(256,512)
*imd1 = 127;
Img2duc imd2(256,512)
imd2 = 127.5; // Use 127
```

2.5.10 File Transfer

To save an image in a file, just use:

```
Img2duc img;
img.SaveFile("foobar.pan");
```

To load an image from a file, just use:

```
Img2duc img;
img.LoadFile("foobar.pan");
```

2.5.11 Miscellaneous

The member function `Hold()` tests whether a point is in or out of the image boundary. For example:

```
Img2duc ims(100,200);
Point2d p(-1,-1);
ims.Hold(p);            // returns false;
ims.Hold(5,10);         // returns true;
ims.Hold(10,199);       // returns true;
ims.Hold(10,200);       // returns false;
```

The member function `Frame()` sets the border of the image with a given value or with the pixel of a given image. For example:

```
ims2.Frame(127,5);      // set the border (5x5) with the value 127.
ims2.Frame(ims1,5,6);   // set the border (5x6) with the pixel of the image ims1.
```

2.6 Accessing Image Pixel

Image pixel access uses traditional C array element access. Pixel neighbour access uses predefined arrays indexed by conventional Freeman encoding.

2.6.1 Accessing Image Pixel

The declaration of image dimension uses the following conventional order:

- **(row, column)** for 2D image.
- **(depth, row, column)** for 3D image;

Thus, access to a image pixel follows the same convention:

```
value=ims[row][column]; for 2D image
value=ims[depth][row][column]; for 3D image
```

For example:

```
Img3duc image1(64,128,256); // 64 slices, 128 rows and 256 columns
Img2duc *image2 = new Img2duc(128,256); // 128 rows and 256 columns
int z=12,y=11,x=10;

image1[z][y][x]=12;
(*image2)[y][x]=12;
```

2.6.2 Accessing Neighbour Pixels

The Freeman encoding assigns a code (an integer) to each immediate neighbour of a pixel (or a voxel in 3D). The encoding depends on the dimension (2D or 3D) and on the connexity (4 or 8 for 2D; 6 or 26 for 3D).

2.6.2.1 Freeman Encoding for 2D Image

Convention For 2D, the encoding is given as follows:

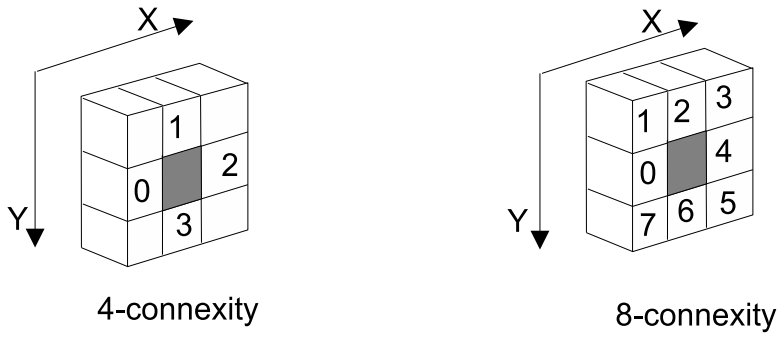

Figure 4: The Freeman encoding for 2D for 4-connexity and 8-connexity.

This encoding guaranties two properties:

1. Pixels v_i for i in [0;1] in 4-connexity (resp. [0;3] in 8-connexity) are visited before the central pixel and pixels v_i for i in [2;3] (resp. [4;7]) are visited after.
2. In 4-connexity, pixel v_i and pixel v_{2+i} are symmetrical (resp. v_i and v_{4+i} in 8-connexity).

Using Predefined Arrays Access to a specified neighbour is done by using predefined arrays.

The four arrays v4x[ ], v4y[ ] and v8x[ ], v8y[ ] indicate the x and y shifts to operate from the central pixel to access a given neighbour in 4 and 8 connexity respectively:

```
int shiftx, shifty;
for (v=0;v<4;v++) { shiftx=v4x[v]+x; shifty=v4y[v]+y; } // In 4-connexity
for (v=0;v<8;v++) { shiftx=v8x[v]+x; shifty=v8y[v]+y; } // In 8-connexity
```

For example, to perform a mean filter (set the central pixel with the mean of its neighbours), use:

```
Long x,y;
for (int y=1;y<ims.Height()-1;y++)
   for (int x=1;x<ims.Width()-1;x++) {
        float sum=0.0F;
        for (int v=0;v<8;v++)
                sum+=ims[y+v8y[v]][x+v8x[v]];
        ims[y][x]=sum/8;
   }
```

The two arrays v4[ ] and v8[ ] give the shifts as a 2D point for 4 and 8 connexity respectively:

```
Point2d shiftp, p(10,10);
for (v=0;v<4;v++) { shiftp=v4[v]+p; } // In 4-connexity.
for (v=0;v<8;v++) { shiftp=v8[v]+p; } // In 8-connexity.
```

For example, to perform a mean filter (set the central pixel with the mean of its neighbours), use:

```
Point2d p;
for (int p.y=1;p.y<ims.Height()-1;p.y++)
   for (int p.x=1;p.x<ims.Width()-1;p.x++) {
        float sum=0.0F;
        for (int v=0;v<8;v++)
                sum+=ims[p+v8[v]]];
        ims[p]=sum/8;
   }
```

2.6.2.2 Freeman Encoding for 3D Image

Convention For 3D, the encoding is given as follows:

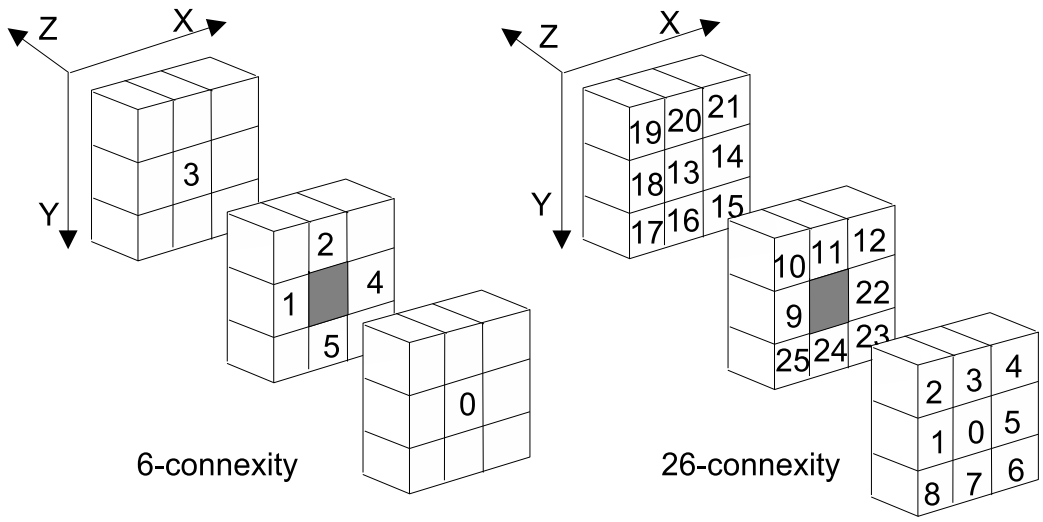

Figure 5: The Freeman encoding for 3D for 6-connexity and 26-connexity.

This encoding guaranties two properties:

1. Voxels v_i for i in [0;2] in 6-connexity (resp. [0;12] in 26-connexity) are visited before the central pixel and voxels v_i for i in [3;5] (resp. [13;25]) are visited after.
2. In 6-connexity, voxel v_i and voxel v_{3+i} are symmetrical (resp. v_i and v_{13+i} in 26-connexity).

Using Predefined Arrays Access to a specified neighbours is done by using predefined arrays.

The six arrays v6x[ ], v6y[ ], v6z[ ] and v26x[ ], v26y[ ], v26z[ ] indicate the x, y and z shifts to operate from the central pixel to access a given neighbour in 6 and 26 connexity respectively:

```
int shiftx, shifty, shiftz;
for (v=0;v<6;v++) { shiftx=v6x[v]+x; shifty=v6y[v]+y; shiftz=v6z[v]+z }
for (v=0;v<26;v++) { shiftx=v26x[v]+x; shifty=v26y[v]+y; shiftz=v6z[v]+z }
```

For example, to perform a mean filter (set the central pixel with the mean of its neighbours), use:

```
Long x,y,z;
for (int z=1;z<ims.Depth()-1;z++)
   for (int y=1;y<ims.Height()-1;y++)
        for (int x=1;x<ims.Width()-1;x++) {
            float sum=0.0F;
            for (int v=0;v<26;v++)
                sum+=ims[z+v26z[v]][y+v26y[v]][x+v26x[v]];
            ims[z][y][x]=sum/26;
       }
```

The two arrays v6[ ] and v26[ ] give the shifts as a 3D point for 6 and 26 connexity respectively:

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

For example, to perform a mean filter (set the central pixel with the mean of its neighbours), use:

```
Point2d p;
for (int p.z=1; p.z<ims.Depth()-1; p.z++)
   for (int p.y=1; p.y<ims.Height()-1; p.y++)
        for (int p.x=1; p.x<ims.Width()-1; p.x++) {
            float sum=0.0F;
            for (int v=0;v<26;v++)
                sum+=ims[p+v26[v]];
            ims[p]=sum/26;
       }
```

2.6.3 Generalized Access

Arrays prefixed by *vc* allow generalized access to a specified neighbour from a given connexity:

- as a Point: vc[connexity][code],
- as coordinates: vcx[connexity][code], vcy[connexity][code], vcz[connexity][code].

For example:

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

2.7 The Region Maps

2.7.1 Definition

A region map is an image of labels. A region is defined by a set of connected labels with the same value. Each pixel in a region map stores a label which is a Ulong value (long unsigned integer). This means that a region map can store **4294967294 different regions**.

Note:

By pure convention, label 0 is considered as a non region.

Concretely, a region map is represented by an `Img2dul` image (image with Ulong pixels). It implies that all image member functions are applicable to a region map.

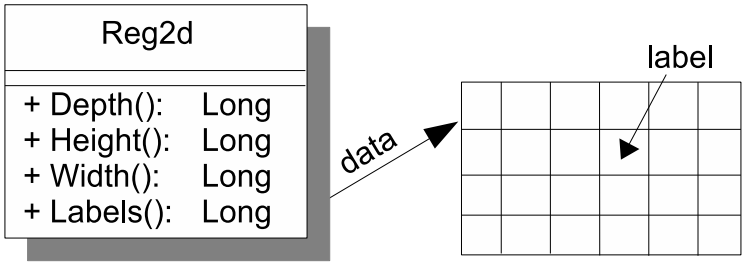

Figure 6: The class Reg2d.

2.7.2 Types

There are 3 different types of region map according to the dimension:

- `Reg1d:` region map in 1D;
- `Reg2d:` region map in 2D;
- `Reg3d:` region map in 3D.

2.7.3 Public Attributes

A region map is first of all a Ulong image plus an attribute that indicates the value of the higher label. These attribute values are accessible by the way of member functions:

- `Long Width()`: returns the number of columns;
- `Long Height()`: returns the number of rows;
- `Long Depth()`: returns the number of planes;
- `Dimension2d Size()`: returns the size as dimension;
- `int VectorSize()`: returns the size of the data vector;
- `Long Labels()`: returns the higher label value;
- `Long Labels(Long labelmax)`: sets the higher label value;
- `PobjectProps Props()`: returns a structure with the region attribute values.

2.7.4 Construction

A region map can be created with or without the related data. If the dimension is not given with the constructor, then the data is not allocated. For example, creation of a 2D region map can be done as follows:

```
Reg2d rgs1(256,512);      // Data: 256 rows, 512 columns
Reg2d rgs2(rgs1.Size()); // Data: same size than rgs1
Reg2d rgs3;               // No data
Reg2d *rgs4=new Reg2d;    // No data
```

The creation of a region map from the properties of an other Pandore object can be done by using:

```
Reg2d rgs1(ims2.Props()); // same size than ims1
```

2.7.5 Allocation

If the region map has been created without data, the member function `New()` creates the data array -if data already exist they are first deleted. For example:

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

Warning:

The creation of the data does not initialized pixel values to 0. However, this can simply be done using:

```
Reg2d rgs(256,256);
rgs = 0;        // This also sets Labels(0).
rgs = 127;      // This also sets Labels(127).
```

2.7.6 Allocation from a predefined array

It is possible to allocate the data from a predefined vector:

```
Ulong *d = (Ulong*)malloc(128*256*sizeof(Ulong));
Reg2d rgs1(256,128,d);
Reg2 *rgs2 = new Reg2(256,128,d);
rgs1.delete();
delete rgs2;
free(d);
```

Warning:

In this case, the destruction is not done with the object.

For example, to process a 3D region map as several 2D region maps:

```
Reg3d *rgs= new Reg3d(12,200,256);
for(d=0;d<rgs->Depth();d++) {
 Reg2d *regii=new Reg2d(rgs->Height(),rgs->Width(),&(*rgs)(d,0,0));
 Reg2d *regio = new Reg2d(regii->Props());
 gauss:Gauss(*regii,*regio,2.0F);
 *regii=*regio;
 delete regii;
 delete regio;
}
```

2.7.7 Destruction

To delete the data of a region map without deleting the region map itself, use the member function `Delete()`. For example:

```
Reg2d rgs1(120,245);
rgs1.Delete();
Reg2d rgs4 = new Reg2d(120,245);
rgs4->Delete();
rgs4->New(400,200);
```

2.7.8 Consultation

Access to label value can be done in three ways:

1. As a multidimensional array (depth,row,column):

   ```
   rgs1(i,j)=15;
   (*rgs4)(i,j)=15;
   Point2d pt(10,20);
   rgs1[pt]=15;
   ```

2. As a unique vector: `Vector()`

   ```
   Reg2d rgs(256,512);
   for (Ulong *p=rgs.Vector(); *p< rgs.Vector()+rgs.VectorSize();)
      *p++ = 15;
   ```

3. As a separate multidimensional array: `X()`

   ```
   Reg2d reg1(10,20);
   Reg3d reg2(123,124,120);
   ULong **d=reg1.X();
   Ulong ***d1=reg2.X();
   ```

2.7.9 File Transfer

To save a region map in a file, just use:

```
Reg2d reg;
reg.SaveFile("foobar.pan");
```

To load an region map from a file, just use:

```
Reg2d reg;
reg.LoadFile("foobar.pan");
```

2.7.10 Miscellaneous

The member function `Hold()` tests whether a point is in or out of the region map boundary. For example:

```
Reg2d rgs(100,200);
Point2d p(-1,-1);
rgs.Hold(p);            // returns false;
rgs.Hold(5,10);         // returns true;
rgs.Hold(10,199);       // returns true;
rgs.Hold(10,200);       // returns false;
```

The member function `Frame()` sets the border of the region with a given value or with the pixel of a given region. For example:

```
rgs2.Frame(127,5);      // set the border (5x5) with the value 127.
rgs2.Frame(rgs1,5,6);   // set the border (5x6) with the pixel of the region rgs1.
```

2.8 The Graphs

2.8.1 Definition

A graph object allows the representation of the neighbourhood relation between elements located in the spatial domain. A graph is composed of nodes linked to other nodes by edges.

The two types of graph are supported: directed and undirected graphs. With undirected graph, edges are symmetrical and with directed graph edges are non symmetrical.

The graph is disconnected from the array of objects it organizes. An index in each node is the mean to reference an object in a separate array, as a pointer, but without using a specified type. This principle allows the definition of any type of graph (with the same graph type): a graph of points, a graph of region maps, a graph of images, etc. This implies to define at the same time the graph and the array of objects. For example, to define a graph of region maps, one has to define a graph and an array of region maps. In the same way, edges can be described by an external array which lists properties for each edges.

2.8.1.1 Node A node (type `GNode`) is characterized by:

- a value (Double) –that can be a weight for example–,
- its spatial coordinates (Point2d or Point3d)

- an index that references an element in an external array of elements (Long).

Each node `i` is characterized by an integer –accessible by `g[i]->Item()`– which indexes an element in the array of elements. For example, suppose the structure of elements `tab` that contains at least the field `size` and a graph `grs`:

```
struct element tab[50]; // An array of 50 elements (Long).
id=grs[i]->Item();      // id is the index in the array of objects.
tab[id].size=120;       // Sets the size of the object to 120 pixels.
```

2.8.1.2 Edge An edge (type `GEdge`) is characterized by its weight which is a Double value and an index to an external array that can describe the properties of the edge. Several edges can be created between two nodes. Each edge is identified by an index with can be used to reference an external array that decribes the edge properties.

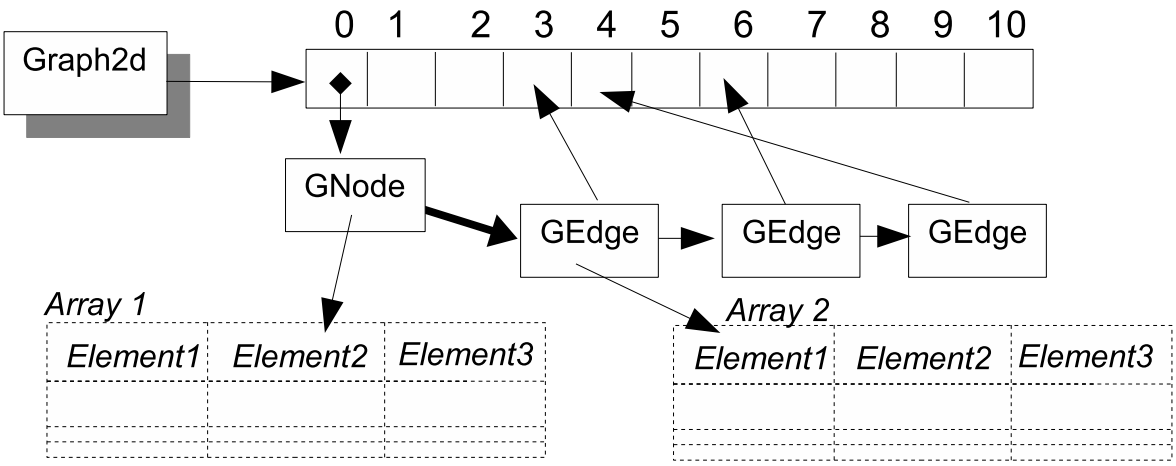

Figure 7: Representation of a graph with 11 nodes. The node #1 represents the element #2 of the array #1. The node #0 has the neighbour nodes 3, 4 and 6. The edge #1 is described by the element #1 of the array #2.

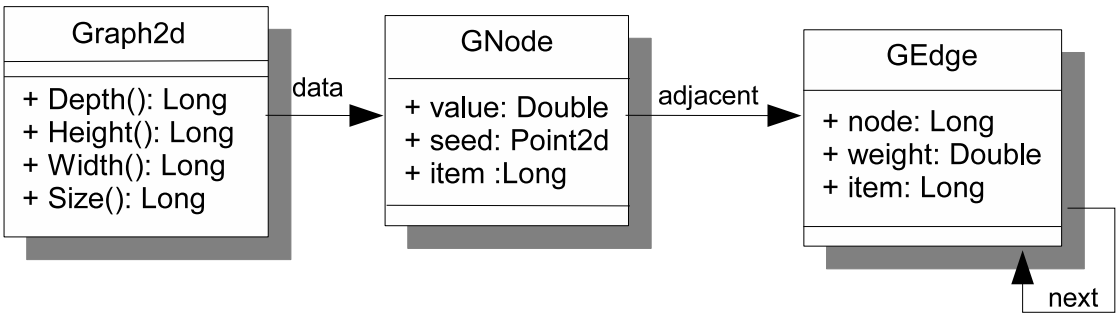

Figure 8: The class diagram for Graph2d.

2.8.2 Types

There are 2 different types of graph according to the dimension:

- `Graph2d:` graph in 2D
- `Graph3d:` graph in 3D.

2.8.3 Public Attributes

A graph is characterized by the array of nodes and a spatial domain. This implies that the number of nodes must be known before the creation of a graph.

2.8.3.1 Public Attributes of graph Attribute values of graph are accessible by the way of:

- `bool isDirected()`: returns true if the graph is directed, false otherwise.
- `Long Size()`: returns the number of nodes;
- `Long Width()`: returns the number of columns of the related space;
- `Long Height()`: returns the number of rows of the related space;
- `Long Depth()`: returns the number of planes of the related space;
- `Dimension ImageSize()`: returns the dimension of the related space;
- `PobjectProps Props()`: returns a structure with the graph attribute values.

2.8.3.2 Public Attributes of nodes Attribute values of node are accessible by the way of:

- `Double value`: stores the value of the node;
- `Long Item([int x])`: returns (if x is omitted) or sets (if x is given) the index of the element in the array of elements;
- `Point2d seed`: stores the coordinates of the node;
- `GEdge* Neighbours()`: returns the list of neighbour nodes.

2.8.3.3 Public Attributes of edges Attribute values of an edge between two nodes is accessible by the way of:

- `Long Node()`: returns the node number of the neighbour;
- `Long Item([int x])`: returns (if x is omitted) or sets (if x is given) the index of the element in the array of elements;
- `Double weight`: stores the weight;
- `GEdge* Next([GEdge* x])`: returns (if x is omitted) or sets (if x is given) the next neighbour.

2.8.4 Construction

To create a new graph use the followings constructors.

```
Graph2d( bool directed =false);
Graph2d( Long s, bool directed =false );
Graph2d( Long s, Long h, Long w, bool directed =false );
Graph2d( Long s, const Dimension2d &d, bool directed =false);
Graph2d( const PobjectProps &p );

Graph3d( bool directed =false);
Graph3d( Long s, bool directed =false );
Graph3d( Long s, Long d, Long h, Long w, bool directed =false );
Graph3d( Long s, const Dimension3d &d, bool directed =false);
Graph3d( const PobjectProps &p );
```

For example, to create a new **directed** graph with or without related data, use:

```
Graph2d g(100,256,512,false);      // Data: 100 nodes; Space: 256 rows x 512 columns.
Graph2d g;                         // No data, no space.
Graph2d *g=new Graph2d(200);       // Data: 200 nodes; no space.

Graph3d g(10,10,25,12,false);      // Data: 10 nodes; Space 10 planesx25 rowsx12 columns
Graph3d g;                         // No data; No Space, and undirected.
```

For example to create a new **directed** graph with or without related data, use:

```
Graph2d g(100,256,512,true);       // Data: 100 nodes; Space: 256 rows x 512 columns.
Graph2d g(true0;                   // No data, no space.
Graph2d *g=new Graph2d(200,true);  // Data: 200 nodes; no space.
Graph3d g(10,10,25,12,true);       // Data: 10 nodes; Space 10 planesx25 rowsx12 columns
Graph3d g(true);                   // No data; No Space, and undirected.
```

To create a graph from the properties of another Pandore object, use:

```
Graph2d s1(obj2.Props());
```

Note:

If `obj2` is a graph then the number of nodes of `gs1` is equal to the number of node `obj2`. If `obj2` is a region map then the number of nodes of graph `gs1` is equal to the higher value of labels +1. If `obj2` is an image then the number of nodes of graph `gs1` is equal to the number of pixels.

2.8.5 Initialisation

If a graph has been created without data, the member function `New()` allocates the data. If the data already exist they are deleted before the reallocation. For example:

```
g.New(100,256,128);
Graph2d *gs1; gs1->New(100,256,128);
```

To create a new graph from a region map use the member function `Init(Reg2d &rgs)`. Seeds are set to the coordinates of the upper left point of the region (not the centre of mass). For example:

```
Reg2d rgs(100,100);
Graph2d grs(false);
grs.Init(rgs);
```

To create a new graph from a region map `rgs` and a seed map `seed` -seeds are given as **punctual regions** in a region map- use the member function `Init(const Reg2d &rgs, const Reg2d &seed)`. For example:

```
Reg2d rgs1;
Reg2d seed;
grs.Init(rgs1,seed);
```

To create a graph from an another graph use operator `=`. For example:

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

2.8.6 Destruction

To delete graph's data without deleting the graph itself use member function `Delete()`. For example:

```
Graph2 gs1(12,256,256);
gs1.Delete();
gs1.New(120,256,256);
gs2->Delete();
```

2.8.7 Adding nodes

To add node s in the graph that indexes the element i at coordinates (y,x), use the member function `Add()`. For example, to add the node 5 that indexes the element 6 at the 3D coordinates z=50, y=12, x=10:

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

The shorter instruction

```
grs.Add(5,6);
```

do not consider the coordinate.

To get the index of the element represented by node `s1` use the member function `Item()`. For example:

```
Long nbreg = g[i]->Item();
```

Warning:

Node `g[i]` does not always exit, for instance when a node has been deleted with the member function `Del()`. So it is necessary to check first the existence of the node before using it:

```
if ((g[i]!=NULL))
        ... g[i]-> ...
```

2.8.8 Deleting nodes

To delete the node `s` from the graph, use the member function `Del()`. For example:

```
grs.Del(10);    // delete node 10.
```

2.8.9 Linking nodes

To add node `s1` in the list of neighbours of the node `s2` use the member function `link()`. An edge is created between `s1` and `s2`, and if the graph is undirected the symmetrical edge between `s2` and `s1` is also added. If the edge already exists the weight is updated either by setting a new value if parameter `add=false` or by adding the new value to the current value if `add=true`. By default, the weight value is 1.0.

For example, to create an edge between nodes 10 and 12 with weight=1.0:

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

For example, to add the value 5.0 to the current value of the weight:

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

To create several edges between two nodes; it is necessary to identify each edge with an integer. This integer can next be used to reference an external array that described the edge properties. For example, the following code snippet defines an array with a color for each edge. The edge between the node 10 and 12 is created and indexed with the 20th and the 22th colors.

```
Color[100] colors;
grs.Link(10,12,20);
grs.Link(10,12,22);
```

To get the list of neighbours of a node use the member function `Neighbours()`. For example:

```
GEdge* l = g[i]->Neighbours();
l=l->Next();
for (GEdge* l = g[i]->Neighbours(); l!=NULL; l=l->Next())
   Long node = ptr->Node();
```

2.8.10 Unlinking nodes

To delete node `s1` from the list of neighbours of node `s2` use the member function `Unlink(Long s1,Long s2)`. If the graph is undirected the symmetrical edge is also deleted. For example:

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

If several edge are used between nodes, use the edge index to delete a specified edge. For example, to delete the edge indexed by 2:

```
grs.Unlink(10,12,2);
```

2.8.11 File Transfer

To save a graph in a file, just use:

```
Graph2d grs;
grs.SaveFile("foobar.pan");
```

To load a graph from a file, just use:

```
Graph2d grs;
grs.LoadFile("foobar.pan");
```

2.8.12 Miscellaneous

The member function `Merge()` merges 2 nodes (`n1` and `n2`) into 1 node. Is also merges the list of neighbours. The node `n1` is kept and the node `n2` is deleted. The list of edges of node `n2` is added to the list of edges of node `n1`. The weight of the common edges are added. For example:

```
Graph2d grs;
Reg2d rgs(120,102);
grs.Init(rgs);
Reg2d::ValueType r1=10,r2=12;
gd.Merge(r1,r2); // Merges regions r1 and r2.
```

The member function `Split()` splits one node into two nodes. The new node is a copy of the old node. Both nodes have the same attribute values and the same list of neighbours. For example:

```
Graph2d grs;
Reg2d rgs(120,102);
grs.Init(rgs);
Reg2d::ValueType r1=10,r2=120;
gd.Split(r1,r2); // Split region r1 in r1 and r2.
```

2.9 The Collection

2.9.1 Definition

A Collection is a bundle of heterogeneous data (*a la* struct C). Each data in a collection is indexed by a name. Available types of data are:

- primitive C types (Uchar, Char, Short, Long, Uchar ... - except int);
- arrays of primitive C types;
- Pandore objects (even collection);
- arrays of Pandore objects.

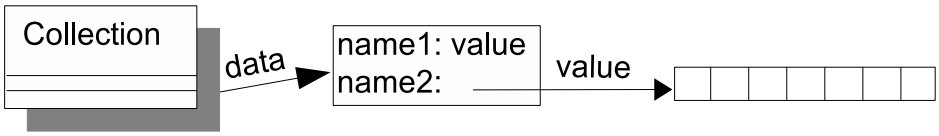

Figure 9: The class Collection.

2.9.2 Types

There exists only 1 type of Collection:

- Collection: a collection of anything.

2.9.3 Construction

To declare a collection, just use:

```
Collection c1;
```

or

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

To copy a specified collection in an other collection, use:

```
Collection *c1, *c2;
...
*c1=*c2;
```

or

```
Collection *c2;
...
Collection *c3=c2->Clone();
```

2.9.4 Destruction

To delete the data of a collection without deleting the collection itself, just use:

```
c1.Delete();
```

2.9.5 Consultation

The easiest way to set a data in a collection is to use the following macros:

- `SETVALUE( name, type, value )` (*type is from Pandore basic types: Uchar, Slong, Ulong, Float... - int is not allowed*);
- `SETARRAY( name, type, pointer_to_array, number_of_items )` (*type is from Pandore basic types: Uchar, Slong, Ulong, Float... - int is not allowed*);
- `SETPOBJECT( name, type, pointer_to_object )`;
- `SETPARRAY( name, type, pointer_to_object, number_of_items )`.

The easiest way to get a data from a collection is to use the following macros:

- `GETVALUE( name, type )`;
- `GETARRAY( name, type )` + `GETARRAYSIZE( name, type )`;
- `GETPOBJECT( name, type )`;
- `GETPARRAY( name, type )` + `GETPARRAYSIZE( name, type )`.

Examples:

1. Set and get a primitive value named "foo" in the collection:

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

2. Set and get an array of primitive values named "bar" in the collection:

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

3. Set and get a Pandore object (Imc3duc) named "foo" in the collection:

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

4. Set and get an array of Pandore objects (Point2d) named "bar" in the collection:

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

Warning:

`SETARRAY`, `SETPOBJECT` and `SETPARRAY` do not make a copy of the data, it is just a reference to the object. Consequently, the following example generates an error because p1 is a local array that is deleted at the end of the function Bar.

```
Errc Bar( Collection  &cold ) {
   Point2d p1[12];
   cold.SETPARRAY("foo",Point2d,p1,12);
}
```

Remarks:

In case of SETXXXX, if the name already exists then the related data is replaced without destruction.

Note:

In the special case of a variable number of data with a same name, the convention is to use several arrays prefixed with the name: name.0, name.1, name.2... For example, the red, green and blue for 10 pixels can be represented by three arrays: t.1, t.2, t.3 with 10 items each. The member function `NbOf()` returns the number of components and `GETNARRAYS()` returns the list of arrays. For example:

```
Collection col;
Long minsize, size;
std::string type;

col.NbOf("foo",type,size,minsize);
```

```
if (type == "Array:Char") {
   Char **foo=cold.GETNARRAYS("foo",Char,size,minsize)
   for (int i=0; i<size; i++) {
      Char* foo_i=foo[i];
      for (c=0; c<minsize; c++) {
       < ... using foo_i[c] ... >
      }
   }
}
```

2.9.6 File Transfer

To save a collection, just use:

```
Collection col;
col.SaveFile("foobar.pan");
```

To load a collection, just use:

```
Collection col;
col2.LoadFile("foobar.pan");
```

3 Programming

3.1 Operator Programming

3.1.1 Atomic Operator

Operator differs from application in what it is **atomic**. The concept of atomicity does not refer to the grain-size of the operator but to the fact that the control inside the operator must be solved. An operator is not atomic when:

- Some parts of the operator can be exchanged by another;
- Some parts of the operator can be controlled individually;
- Some parts of the operator need a special skill to control it.

In such cases, the operator should must be split into several atomic operators.

The goal of such a principle is threefold:

- It reduces the number of parameters to be tuned. The less is the number of parameters the less is difficult to use it;
- It reduces the operational knowledge required to select the operator, to tune its parameter and to control its execution;
- It reduces the number of needed operators in the library when some combining operators exist.

3.1.2 Operator Template File

The following template gives the general structure of an operator file.

```
/* -*- mode: c++; c-basic-offset: 3 -*-
 *
 * Copyright (c) 2013, GREYC.
 * All rights reserved
 *
 * You may use this file under the terms of the BSD license as follows:
 *
 * "Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in
 *     the documentation and/or other materials provided with the
 *     distribution.
 *   * Neither the name of the GREYC, nor the name of its
 *     contributors may be used to endorse or promote products
 *     derived from this software without specific prior written
 *     permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
 *
 * For more information, refer to:
 * https://clouard.users.greyc.fr/Pandore/
 */

#include <pandore.h>
using namespace pandore;

Errc Operator( const Img2duc &ims, Img2duc &imd, Short p ) {
   // Body
   return SUCCESS;
}

Errc Operator( const Img2dsl &ims, Img2dsl &imd, Short parameter ) {
   // Body
   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    0  // 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 objects.
   Pobject* objd[FOUTC + 1];    // The result objects of the execution.
   char* 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 = Operator(*ims, *imd, atoi(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 = Operator(*ims, *imd, atoi(parv[0]));
      break;
   }
   default:
      PrintErrorFormat(objin, FINC);
      result = FAILURE;
   }

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

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

3.1.3 The operator() Function

Inputs and outputs are Pandore objects. To a first approximation, it is necessary to define one function per available object composition. For example, from the previous example, a function is defined for `Img2duc` x `Img2duc` and one for `Img2dsl` x `Img2dsl`.

However, it is possible to write only one function when the algorithm is the same for each type. There are two different ways to write generic functions:

1. The first one uses the hierarchy of the objects -see below **Writing Generic Operator Function Using Hierarchy**(p. 32).
2. The second one uses the preprocessor -For more details see **Preprocessing of operators**(p. 37).

3.1.3.1 Writing Generic Operator Function Using Hierarchy For instance, all images and region maps inherit from the three classes Imx3d<Uchar>, Imx3d<Long>, Imx3d<Float>. So, the generic function can be defined as a template function. Such a function can be called with any image and region map types:

```
template <typename T1, typename T2>
```

```
Errc Operator ( const Imx3d<T1> &ims , Imx3d<T2> &imd, int p) {
   // Body
   return  SUCCESS;
}
```

The following example is the morphological erosion function for 2D images:

```
template <typename T>
Errc Erosion( const Img2d<T> &ims, Img2d<T> &imd, int connexity ) {
   Point2d p;
   T min,val;

   if (connexity != 4 && connexity != 8)
        return FAILURE;
   imd.Frame(0,1,1);
   if (connexity == 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 { // connexity == 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;
}
```

3.1.3.2 Value Type For a given type of Pandore object T the field `T::ValueType` gives access to its data type. For example:

```
Img2dsf::ValueType -> Float

template <typename T1, typename T2>
Errc Operator ( const Imx3d<T1> &ims , Imx3d<T2> &imd, int p) {
   for (int b=0; b<ims.Bands(); b++) {
      T1::ValueType *ps=ims.Vector(b);
      T2::ValueType *pd=imd.Vector(b);
      for ( ; p<ims.Vector()+ims.VectorSize(); ps++,pd++ ) {
        *pd = T2(*ps *2);
      }
   }
   return  SUCCESS;
}
```

3.1.3.3 Type Limits The two traits `Limits<T>::max()` and `Limits<T>::min()` return respectively the maximum and the minimum values of the primitive type T (Uchar, Slong, Float....). For example:

```
Limits<Ushort>::max() -> 65535
Limits<Img2duc::ValueType>::max() -> 255
```

3.1.3.4 Type Deductions Sometimes, it necessary to choose between two types. For example, an algorithm can take two types T1 and T2 as inputs and returns a result type that is the larger unsigned type between the two input types. This can be done using the trait `Select`:

1. `Select<T1,T2>::LargestUnsigned`: returns the largest unsigned of the two types;
2. `Select<T1,T2>::LargestSigned`: returns the largest signed of the two types;
3. `Select<T1,T2>::SmallestUnsigned`: returns the smallest unsigned of the two types.
4. `Select<T1,T2>::SmallestSigned`: returns the smallest signed of the two types;
5. `Select<T1,T2>::Largest`: returns the largest of the two types (signed > unsigned);
6. `Select<T1,T2>::Smallest`: returns the smallest of the two types (signed > unsigned).

For example:

```
Select<Uchar,Short>::LargestSigned -> returns Short
Select<Img3duc,Img3dsl>::LargestSigned -> returns Img2sdl
Select<Uchar,Short>::LargestUnsigned -> Ushort
Select<Uchar,Char>::Largest -> Char
Select<Uchar,Char>::Smallest -> Uchar
```

3.1.4 The main() Function

The `main()` function is used to generate a standalone program with the operator. When the operator is used as a function of another program then the main() must be discarded. That is why the `main()` is enclosed between the two C directives `#ifdef MAIN` and `#end`. If the value of macro MAIN is defined then the operator is compiled as a standalone program else simply as a separate module.

3.1.4.1 Reading Inputs The function `ReadArgs()` makes the verification of the argument command line and reads the input files and the parameters.

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

ReadArgs uses a set of constants that prototypes the operator command line:

- the text that describes the usage of the operator (USAGE);
- the number of parameters (PARC);
- the number of input Pandore files (FINC);
- the number of output Pandore files (FOUTC);

Parameters are accessible by two ways:

- By using classical argv[] array. Parameters are located from [1..PARC] if there is no mask given, or [3..PARC+2] if there is a mask given.
- By using parv[] array. Parameters are located from [0..PARC-1] and are of type char*.

3.1.4.2 Masking and Unmasking The **masking operation** allows to apply a same operator to the whole image or to a determined part specified by a mask. The mask is a region map where the pixels with label 0 indicate the masked part.

First of all, the input image is built with the given input image masked by the given region map. All the pixel of the initial image that are masked by the region are set to 0, all the other are kept with their initial value.

Then, the operator is applied on the whole image even on the masked pixels.

Finally, the output image is built by the **unmasking operation** on the processed image. The output pixels are set with the new value if they are not masked or with their initial value if the are masked.

Remarks:

Sometimes, the masking operation cannot be applied as such since some pixel values are replaced by 0. For example:

- consider the mean operator: each pixel is replaced by the mean of all its direct neighbours. However, the masking and unmasking operations produce incorrect pixel values on boundaries of the mask since some neighbour pixels are set to 0. In that case, only the unmasking operation must be operated.
- consider the binarization operation: the input and the output are of different types. So the unmasking operation cannot be applied. Only the masking operation must be applied.

The constant MASK is used to specify whether the masking and unmasking should be applied on the given operator.

- MASK=0: neither masking nor unmasking operation allowed.
- MASK=1: both masking and unmasking operations allowed.
- MASK=2: only masking operation allowed.
- MASK=3: only unmasking allowed.

3.1.4.3 The Switch The switch control structure selects the convenient operator function from the input objects. The type of object is known from the `Type()` member function.

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

3.1.4.4 Writing Outputs The function `WriteArgs()` creates the output results.

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

This function uses the same arguments than the ReadArgs function. This time, if MASK=1 or MASK=3 then output images are unmasked before return.

To set the output result value, use:

```
Exit(result);
```

This value can be get by the command `pstatus`.

3.2 Application Programming

3.2.1 Application Programming

An application is a chain of operators. Because operators are available both as executable commands and as C++ functions, an application can be built in two ways:

1. A script (any scripting language is available: Perl, Bash, Msdos, etc);
2. A C++ program.

Script should be preferred to C++ program during the prototyping phase. C++ program should be preferred for the final product.

3.2.1.1 Application as C++ Program The C++ program consists in a sequence of operator calling, one after the other.

3.2.1.2 A Template File Here is a template file example for an image processing application:

```
/* -*- mode: c++; c-basic-offset: 3 -*-
 *
 * Copyright (c) 2013, GREYC.
 * All rights reserved
 *
 * You may use this file under the terms of the BSD license as follows:
 *
 * "Redistribution and use in source and binary forms,  with or without
 * modification,  are permitted provided that the following conditions are
 * met:
 *   * Redistributions of source code must retain the above copyright
 *     notice,  this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above copyright
 *     notice,  this list of conditions and the following disclaimer in
 *     the documentation and/or other materials provided with the
 *     distribution.
 *   * Neither the name of the GREYC,  nor the name of its
 *     contributors may be used to endorse or promote products
 *     derived from this software without specific prior written
 *     permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,  INCLUDING,  BUT NOT
 * LIMITED TO,  THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,  INDIRECT,  INCIDENTAL,
 * SPECIAL,  EXEMPLARY,  OR CONSEQUENTIAL DAMAGES (INCLUDING,  BUT NOT
 * LIMITED TO,  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA,  OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY,  WHETHER IN CONTRACT,  STRICT LIABILITY,  OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE,  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
 *
 * For more information,  refer to:
 * https://clouard.users.greyc.fr/Pandore/
 */

#include <pandore.h>
using namespace pandore;

#undef MAIN
```

```
// Inclusion of needed operators (Unix)
namespace MyOps{
#include "classe/operator.cpp"
}

#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 objects.
   Pobject* objd[FOUTC + 1];    // The result objects of the execution.
   char* parv[PARC + 1];        // The input parameters.

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

   // Read input image.
   Img2duc* const ims=(Img2duc*)objs[0];
   // Create output image
   objd[0] = new Img2duc(ims->Props());
   Img2duc* const imd=(Img2duc*)objd[0];

   // Call of operator(s).
   Errc result = MyOpos::Operator(*ims, *imd, (float)atof(parv[0]));

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

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

Inclusion of Operators The most simple way to build the application program is to include directly the operator C++ file into the application file. To avoid conflicts with local name, each inclusion should be encapsulated in a namespace.

For example, to include the meanfilter operator, use:

```
#undef MAIN
namepace meanfilter {
#include "morphology/meanfilter.cpp"
}
```

With such inclusion, it is now obvious to use the operator function. For instance:

```
result = meanfilter::MeanFilter(ims,imd,8);
```

4 Preprocessor

4.1 Preprocessing of operators

In order to increase the genericity of the operator functions, Pandore provides its own preprocessor. The goal is to write only one operator function and then to generate several C++ functions, one for each required type composition. The preprocessor will be used when the use of the hierarchy is insufficient (see **Hierarchy**(p. 10)).

The preprocessor is written in Perl. It converts generic files (files suffixed by .cct or .ht) files into C++ file. Pratically, it is called directly by the `Makefile`. To convert a .cct file into .cpp file or a file .ht into file .h the `Makefile` uses the commands:

```
perl -Ietc/macros etc/macros/template.pl etc/macros/pand_macros file.cct > file.cpp
perl -Ietc/macros etc/macros/template.pl etc/macros/pand_macros file.ht > file.h
```

where `etc/macros` directory is a subdirectory of the `Pandore` directory.

The preprocessor recognizes lines beginning by '##'. Lines beginning with '##;' are considered as preprocessor comments and are discarded from the generated C++ file.

4.1.1 Generic Program Template File

A .cct file respects more or less the following template file:

```
/* -*- mode: c++; c-basic-offset: 3 -*-
 *
 * Copyright (c) 2013, GREYC.
 * All rights reserved
 *
 * You may use this file under the terms of the BSD license as follows:
 *
 * "Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in
 *     the documentation and/or other materials provided with the
 *     distribution.
 *   * Neither the name of the GREYC, nor the name of its
 *     contributors may be used to endorse or promote products
 *     derived from this software without specific prior written
 *     permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
 *
 * For more information, refer to:
 * https://clouard.users.greyc.fr/Pandore/
 */

#include <pandore.h>
using namespace pandore;

##begin Operator(TYPE1, VOISS)

Errc Operator( const 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=Operator(*ims,*imd,(Float)atof(parv[0]));
   } else
## end
##end

##; Generates Operator for Img1d<T>, Img2d<T> and Img3d<T>.
##forall(Operator,/Img.d/)

##; Generates Operator for Reg2d and Reg3d
##forall(Operator,/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    0

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

The template file is composed of three parts:

1. the **body of the operator function** enclosed between the `##begin` and `##end` macros.
2. the **instanciation of the function** realized by the `##forall` macro;
3. the **main** generated by the `##main` macro.

4.1.2 The ##begin Macro

The code enclosed between the `##begin` and `##end` macros is duplicated as many times as it is instantiated from Pandore objects. For example, the code can be instantiated from Img2duc, Graph3d, Reg2d, ... In order to adapt the code from the various types of Pandore objects, the `begin` macro has some parameters that are set during instanciation. These parameters defined things like dimension, connexity, point and available loops from the type of the Pandore object. For example, if the Pandore type, is a `Img2duc` then, the dimension is Dimension2d, the point is Point2d and the loop is a double for loop.

The declaration of a generic function looks like the following command -A variable is included only if it is used in the body of the function:

```
##begin Operator( TYPE, VOISL, POINT, DIM, LOOPP )
```

4.1.2.1 Parameters of the ##begin Macro

The list of available parameters is:

- **DIM:** the dimension type [`Dimension1d`, `Dimension2d`, `Dimension3d`].
- **VOISS:** the lower connexity type for the given dimension [2 in 1D, 4 in 2D and 6 in 3D]
- **VOISL:** the higher connexity type for the given dimension [2 en 1D, 8 in 2D and 26 in 3D]
- **POINT:** the point type [`Point1d`, `Point2d`, `Point3d`].

- **LOOPP:** the loop with a variable declared with the POINT type.

```
   POINT p;
##LOOPP(ims,p)
```

  generates for a 2D image:

```
   Point2d p;
   for (p.y=0; p.y<ims.Height(); p.y++)
        for (p.x=0; p.x<ims.Width(); p.x++)
```

- **LOOPPIN:** the inverse loop with a variable declared with the POINT type.

```
   POINT p;
##LOOPPIN(ims,p)
```

  generates for a 2D image:

```
   for (p.y=ims.Height()-1; p.y>=0; p.y--)
        for (p.x<ims.Width()-1; p.x>=0; p.x--)
```

- **LOOPPB:** The inner loop with a variable declared with the POINT type and the size of the border.

```
   POINT p;
##LOOPPB(ims,p,5)
```

  generates for a 2D image:

```
   Point2d p;
   for (p.y=5; p.y<ims.Height()-5; p.y++)
        for (p.x=5; p.x<ims.Width()-5; p.x++)
```

- **LOOPPINB:** The inner inverse loop with a variable declared with the POINT type and the size of the border.

```
   POINT p;
##LOOPPINB(ims,p,5)
```

  generates for a 2D image:

```
   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--)
```

For example:

```
##begin Operator( TYPE, POINT, DIM, VOISL, LOOPP )
Errc Operator(TYPE &ims) {
   POINT p;
   Float sum=0.0F;
   DIM d;
   d=ims.Size();
   TYPE imd;
   imd.New(d);

##LOOPPB(ims,p,5) // Inner loop
   {
```

```
      for ( v=0; v<VOISL; v++ )  // VOISL can be 8 for 2D or 26 for 3D.
         sum += ims[p+v$VOISL[v]]
   }

   imd[p]=sum/VOISL;
   imd.Frame(ims,1);
   return SUCCESS;
}
```

is instantiated as follows for an Img2duc image:

```
Errc Operator(Img2duc &ims) {
   Point2d p;
   Float sum=0.0F;
   Dimension2d d;
   d=ims.Size();
   Img2duc imd;
   imd.New(d);

   for (p.y=5; p.y<ims.Height()-5; p.y++)
        for (p.x=5; p.x<ims.Width()-5; p.x++)
   {
      for ( v=0; v<8; v++ )
         sum += ims[p+v8[v]]
   }

   imd[p]=sum/8;
   imd.Frame(ims,1);
   return SUCCESS;
}
```

Note:

The character `$` can be used to separate a preprocessor variable from other words. For instance, the word `Po_TYPE` must be rewritten as `Po_$TYPE` to be instantiated as Po_Imx3dsf if `TYPE` is `Imx3dsf`.

4.1.2.2 The ##append Macro The `append` macro is used to add a case in the `main()` switch. All the variables defined in the parameter list of the begin macro and the variables defined in the main function can be used in the code between `##append` and `##end` macros. Most of the append contents look like the following code:

```
##begin Operator( TYPE,...., LOOPPB, POINT )
....
## append
   if (objs[0]->Type()==Po_$TYPE) {
      TYPE *const ims=(TYPE)objs[0];
      objd[0]=new TYPE(ims->Props());
      TYPE *const imd(TYPE)obd[0];
      result=Operator(*ims,*imd,(TYPE::ValueType)atof(parv[0]));
   }
## end
##end
```

The previous code will be instancied as follows if TYPE=Imx3dsf:

```
   if (objs[0]->Type()==Po_Imx3dsf) {
      Imx3dsf *const ims=(Imx3dsf)objs[0];
        objd[0]=new Imx3dsf(ims->Props());
      Imx3dsf*const imd(Imx3dsf)obd[0];
      result=Operator(*ims,*imd,(Imx3dsf::ValueType)atof(parv[0]));
   }
## end
##end
```

4.1.2.3 The ##forall Macro The macro `##forall` is used to generate the list of operator functions from the list of the chosen types. The parameters are the type of Pandore objects.

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

The type can be specified by using regular expressions. For example:

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

generates:

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

4.1.3 The ##main macro

The main() function is generated from the macro ##main. The macro uses five parameters:

1. The string that describes the usage of operator.
2. The number of parameters;
3. The number of input Pandore files;
4. The number of output Pandore files;
5. the value of the mask flag.

The easier way to parametrize this macro is to define constants as follows:

```
#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
```

The contents of the macro ##append is added to the `main()` in the switch structure.

4.1.4 Example

Here is the complete source of the generic Add operator, which builds a new image (resp. a new graph) from the pixel to pixel mean of two images (resp. node to node of two graphs).

```
#include <pandore.h>

##; Images
##begin Add(IMG1, IMG2, VARS, LOOPP, POINT)
Errc Add(IMG1 &ims1, IMG2 &ims2, Select<IMG1,IMG2>::Signed &imd) {
   POINT p;
```

```
## LOOPP(ims1,p)
   imd[p]=(Select<IMG1,IMG2>::Signed::ValueType)((ims1[p]+ims2[p])/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>::Signed(ims1->Size());
        Select<IMG1,IMG2>::Signed*const imd=(Select<IMG1,IMG2>::Signed*)objd[0];
        result=Add(*ims1,*ims2,*imd);
   } else
## end
##end

##; Graphs
##begin AddGraph(TYPE)
Errc Add(TYPE &gs1,TYPE &gs2,TYPE &gd) {
   int i;

   gd.Init(gs1);
   for (i=1;i<=gd.size;i++)
      if ((g[i]))
         gd[i]->attr=((gs1[i]->attr+gs2[i]->attr)/2.0F);

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

Index