
Labeled object representation and manipulation with ITK

Gaëtan Lehmann²

August 16, 2007

¹INRA, UMR 1198; ENVA; CNRS, FRE 2857, Biologie du Développement et Reproduction, Jouy en Josas, F-78350, France.

Abstract

Richard Beare has recently introduced a new filter to efficiently labelize the connected component with ITK, and has also proposed to use the run-length encoding used in that filter to implement some of the most useful binary mathematical morphology operators: the opening by attribute. Following that idea, and after have searched a way to use the ITK's spatial objects for this task, a new set of classes have been developed to represent and manipulate the labeled images and the objects within them in ITK. Those new classes have been used to implement several labeled images manipulation based on object attributes, as well as the binary specialization of some mathematical morphology filter already included in ITK, and not related to the attribute of the objects. With those last filters, this contribution comes with 49 new classes, and should greatly enhance the binary mathematical morphology in ITK.

All the source codes are provided, as well as a full set of tests and several usage examples of the new classes.

Contents

1	Introduction	2
2	Definitions	3
2.1	Label	3
2.2	Labeled image	3
2.3	Binary image	3
2.4	Attribute	3
3	Existing classes and naming convention in ITK	3
4	Data representation	4
4.1	itk::LabelCollectionImage	4
4.2	itk::LabelObject and its specializations	4
	itk::ShapeLabelObject attributes	5
	itk::StatisticsLabelObject attributes	6
4.3	itk::LabelObjectLine	6

5 General view of the usage	7
5.1 Generating the itk::LabelCollectionImage	7
5.2 Valuating the attributes	7
5.3 Manipulating the itk::LabelCollectionImage	7
5.4 Generating an itk::Image from the itk::LabelCollectionImage	8
6 Prebuilt mini-pipeline filters	8
6.1 Binary filters	8
6.2 Label filters	9
7 Binary specialization of mathematical morphology filters	9
8 Usage examples	9
8.1 Prebuilt pipelines	9
Binary shape opening	9
Statistics relabel	10
Label shape keep N objects	11
Binary fill holes	12
8.2 LabelObject and LabelCollectionImage manipulation	13
AttributeLabelObject	13
8.3 Reading attribute values	17
9 Threading support	18
10 In place filtering	19
11 Wrappers support	19
12 Known bugs and future work	19
13 Conclusion	20
14 Acknowledgments	20

1 Introduction

Identifying the objects in an image is a very common task, often realized by producing an image of the same size with a single pixel value per object. This image is called a labeled image. There are several ways to create this image. It can be done by searching the connected components in a binary image, it can be produced directly by some algorithms, like the watershed transform, it can even be simply done by hand, etc.

2 Definitions

In that article, some terms will be cited very frequently. I will try to define them, in the context of the image analysis.

2.1 Label

A label is an identifier of something with the same characteristics in the image. Those characteristics can be whatever you want, for example, the range of pixel values, the same object in sense of connected component, etc. A label can be represented by anything and only need to be unique in the image. It doesn't even require to be ordered. In practice, we choose to use the integral number types, for several reasons: they are commonly used in image analysis, they efficiently represent the label in memory, and it's easy to find the next label by adding 1.

2.2 Labeled image

A labeled image is an image which contains several labeled pixels. Often, the labels are representing some objects placed on a background, and so the labeled image may use a particular label for the background.

2.3 Binary image

A binary image is an image with two labels: a foreground label and a background label. In practice, the binary images are using a pixel type able to store more than those two values. The foreground is thus defined with a particular label, and the other label in the image are considered as the background. A side effect of that is that a labeled image can be considered as a binary image, and so, it lets us manipulate a single object in a labeled image.

2.4 Attribute

An attribute is a value of any type associated with a label. It can be for example the size of an object, the mean of its pixels intensities, etc.

3 Existing classes and naming convention in ITK

In ITK, the labeled and the binary images are implemented as a simple *itk::Image*. The pixel types used are most of the time integral, signed or unsigned, but may be of other types. Several definitions of a binary image or used in ITK. Depending on the class which implements it, a binary image can be:

- All the pixels with a given value are in the foreground. The others are in the background. That's the definition proposed in that article.
- All the pixels with a given value are in the background. The others are in the foreground.
- All the pixels greater than a value (zero by default, or the mean of the maximum value in the image and the minimum value in the image) are in the foreground. The others are in the background.

This definition is often used in the levelset framework, where a border can be defined at a subpixel resolution.

All those definitions should be uniformized to enhance user experience with ITK. In that article (and all the others from the same author), the first one is the only one used.

The filter which are manipulating binary images are often prefixed with the word "Binary", to differentiate the grayscale version which don't have a prefix. It seem to be a quite good practice which have been kept in that article.

The filter dedicated to the manipulation of labeled images have the word "Label" somewhere in there name. Again, it seem to be a good practice which have been kept in that article.

4 Data representation

The labeled images are often used to represent the connected components of an image. In this contribution, another representation has been chosen.

The objects contained in the image, as connected component, can be efficiently stored in memory as a set of lines, using the run-length encoding: a starting point for each line, and the length of the line on a given dimension (by convention, the dimension 0).

The image is a collection of those objects, which also store some values of the image, like its size, its spacing, etc.

4.1 `itk::LabelCollectionImage`

The *itk::LabelCollectionImage* class is in charge of managing the collection of labeled objects of the image, as well as storing the metadata associated with the image like the spacing, the physical position - all the metadata found in *itk::Image*.

The *itk::LabelCollectionImage* provide a part of the API of the *itk::Image* class, and so can be manipulated as an image¹ in many cases. The performance can be very different however, because of the very different data structure used.

The *itk::LabelCollectionImage* is a templated class, which take a single parameter: the type of *labeled object* stored by that class. The dimension of the image is took from the *labeled object* class, and thus don't need to be defined as template parameter of that class. The pixel type of the image also comes from the *labeled object* class.

4.2 `itk::LabelObject` and its specializations

The *itk::LabelObject* class represent the label objects. It has two main features:

- It manage the set pixels which compose the object. The pixels are stored using the run-length encoding.
- It has a label.

¹It doesn't support the *itk::Image* iterators though

No attribute are stored in this class, which can thus be seen as the base class for the objects with attributes, or which can be used when no attributes are required.

The *itk::LabelObject* class is templated and takes two required template parameters:

- the type of the label,
- the dimension of the image.

Several subclasses are provided with that contribution, to cover the most common usages of the labeled objects manipulation:

- *itk::AttributeLabelObject* is able to store a generic attribute. It is generic in the sense that its type is given in template parameter.
- *itk::ShapeLabelObject* contains numerous attribute related to the shape of the labeled object. Computing the values of those attributes does not require a feature image.
- *itk::StatisticsLabelObject* contains numerous statistics about the grey values of a feature image in the same place than the labeled object. Computing the values of those attributes *does* require a feature image.

The classes *itk::ShapeLabelObject* and *itk::StatisticsLabelObject* have been created to reduce the number of filters made to manipulate the attributes, and to make the computation of all the set of attributes much efficient. In the early stage of development, all the attributes were managed as in *AttributeLabelObject*, and a set of 8 classes made to manipulate a single attribute were provided, leading to a huge number of classes.

The scalar values of the attributes of the *itk::ShapeLabelObject* and the *itk::StatisticsLabelObject* classes are often given both in pixel and in physical units, in order to be able to give some parameter independant of the image spacing.

Both *itk::ShapeLabelObject* and *itk::StatisticsLabelObject* are templated classes. They take the same template parameters than the *itk::LabelObject* class. The 2 first template parameters of the *itk::AttributeLabelObject* class or the same than the *itk::LabelObject* class. The third one is the attribute type.

itk::ShapeLabelObject attributes

- *Size* is the size of the object in number of pixels. Its type is *unsigned long*.
- *PhysicalSize* is the size of the object in physical unit. It is equal to the *Size* multiplicated by the physical pixel size. Its type is *double*.
- *Centroid* is the position of the center of the shape in physical coordinates. It is not constrained to be in the object, and thus can be outside if the object is not convex. Its type is *Point; double, ImageDimension* ζ .
- *Region* is the bounding box of the object given in the pixel coordinates. The physical coordinate can easily be computed from it. Its type is *ImageRegion; ImageDimension* ζ .

- *RegionElongation* is the ratio of the longest physical size of the region on one dimension and its smallest physical size. This descriptor is not robust, and in particular is sensitive to rotation. Its type is *double*.
- *SizeRegionRatio* is the ratio of the size of the object region (the bounding box) and the real size of the object. Its type is *double*.
- *SizeOnBorder* is the number of pixels in the objects which are on the border of the image. This attribute is particularly useful to remove the objects which are touching too much the border. Its type is *unsigned long*.
- *FeretDiameter* is the diameter in physical units of the sphere which include all the object. The feret diameter is computed by default only if the dimension of the image is 2, because of its high computation cost for higher dimensions. Its type is *double*.

itk::StatisticsLabelObject attributes

- *Minimum* is the minimum value in the feature image for the object. Its type is the feature image pixel type.
- *MinimumIndex* is the index position in the image where the first minimum was found. Its type is *Index; ImageDimension* ζ .
- *Maximum* is the maximum value in the feature image for the object. Its type is the feature image pixel type.
- *MaximumIndex* is the index position in the image where the first maximum was found. Its type is *Index; ImageDimension* ζ .
- *Mean* is the mean of the pixel values in the object. Its type is *double*.
- *Sum* is the sum of all the pixel values in the objects. Its type is *double*.
- *Sigma* is the standard deviation of the pixels values in the objects. Its type is *double*.
- *Variance* is the variance of the pixels values in the objects. Its type is *double*.
- *Median* is the median of the pixels values in the object. Its type is *double*
- *CenterOfGravity* is the center of gravity of the object. Its type is *Point; double* ζ .
- *Kurtosis* is the kurtosis of the pixel values in the objects. Its type is *double*.
- *Skewness* is the skewness of the pixel values in the objects. Its type is *double*.

4.3 itk::LabelObjectLine

itk::LabelObjectLine is the object used to store the position and the size of a single line.

5 General view of the usage

5.1 Generating the `itk::LabelCollectionImage`

The `itk::LabelCollectionImage` class provide some methods to fill the image "by hand", like the usual `SetPixel()` method. However, the most efficient way is to convert a labeled image or a binary image stored in an `itk::Image` to a `itk::LabelCollectionImage`, by using `itk::BinaryImageToLabelCollectionImageFilter` or `itk::LabelImageToLabelCollectionImageFilter`.

5.2 Valuating the attributes

The label objects produced by those filters have no attribute value set, and thus, the attributes must be valuated. Some filters are provided for the most common used ones:

- `itk::ShapeLabelCollectionImageFilter` to fill the attributes of the `itk::ShapeLabelObjects`,
- and `itk::ShapeLabelCollectionImageFilter` to fill the attributes of the `itk::StatisticsLabelObjects`.

For the `itk::AttributeLabelObject` class or other classes, the user must set the value by himself, for example by implementing a subclass of `itk::InPlaceLabelCollectionImageFilter`.

5.3 Manipulating the `itk::LabelCollectionImage`

Once created and, optionally, valuated, several filters are provided to manipulate the `itk::LabelCollectionImage`:

- An opening can be performed with the `OpeningLabelCollectionImageFilter` classes. Those classes will remove all the objects with an attribute value lower or greater than a given value. Because we often can use some criteria which have not been used during the segmentation procedure, like the size of the object, the mean value of its pixels, etc., the attribute opening is often a very efficient way to enhance a segmentation. For example, after a thresholding of a grayscale image, the objects too small or too big to be of interest can be removed that way.
- A fixed number of objects can be kept, with the `KeepNObjectsLabelCollectionImageFilter` classes. They are chosen according to the value of their attribute. The user can choose to keep the ones with the highest, or with the lowest attribute values.
- The objects can be relabel, with the `RelabelLabelCollectionImageFilter` classes. The order of the label is dependant of the value of the attribute. Again, the user can choose to have the objects with the highest attribute value in the first labels, or to have the objects with the lowest attribute values in the first labels.

It can also be useful to simply get the attribute values associated with the objects. In that case, the classes provided in with that article can be used in place of `itk::LabelStatisticsImageFilter`, or to get some data about the shape or the position of the object.

Finally, it has been chosen to develop a specific filter for the morphological reconstruction. It would have been possible to implement the reconstruction with the most common case (build the object collection,

valuate the attributes filter the object, and rebuild the image), but in order to make it compatible with the *ShapeLabelObject* and *StatisticsLabelObject*, the reconstruction filter filters the collection directly, without setting an attribute in the objects ².

5.4 Generating an `itk::Image` from the `itk::LabelCollectionImage`

Once the manipulation of the objects is done, it can be useful to go back to a more classic *itk::Image*. Several classes are provided to do that:

- The *itk::LabelCollectionImageToLabelImageFilter* class simply convert a *itk::LabelCollectionImage* to a labeled image stored in a *itk::Image*.
- The *itk::LabelCollectionImageToBinaryImageFilter* put all the objects in the foreground of a binary image stored in a *itk::Image*. It is intended to be used with an image produced by the *itk::BinaryImageToLabelCollectionImageFilter*. The background values of the original image can also be restored by this filter.
- The *itk::LabelCollectionImageToMaskImageFilter* class can be used to mask an image with the objects of the *itk::LabelCollectionImage*.
- Finally, *itk::LabelCollectionImageToAttributeImageFilter* produce an *itk::Image* with the value of the attribute of the objects of the *itk::LabelCollectionImage*. This filter is mostly useful to have a global view of the attribute values in the image.

6 Prebuilt mini-pipeline filters

The general view of the previous section show a very common way to use those classes. To make easier to use, some prebuilt classes have been made, to perform the mini-pipeline:

- creation of the *itk::LabelCollectionImage* from an *itk::Image*,
- valuation of the attribute(s) of the objects,
- filtering of the *itk::LabelCollectionImage*,
- creation of an *itk::Image* from the filtered *itk::LabelCollectionImage*,

with a specific attribute.

Because the objects are often get from a labeled image or from a binary image, those filters have been made for binary, and labeled images.

6.1 Binary filters

- *itk::BinaryAttributeKeepNObjectsImageFilter*

²I'm not really pleased with that design though, and I'm thinking about reimplementing it using the generic attributes. It would have no impact on the binary filters API.

- *itk::BinaryAttributeOpeningImageFilter*
- *itk::BinaryShapeKeepNObjectsImageFilter*
- *itk::BinaryShapeOpeningImageFilter*
- *itk::BinaryStatisticsKeepNObjectsImageFilter*
- *itk::BinaryStatisticsOpeningImageFilter*

6.2 Label filters

- *itk::LabelAttributeKeepNObjectsImageFilter*
- *itk::LabelAttributeOpeningImageFilter*
- *itk::LabelShapeKeepNObjectsImageFilter*
- *itk::LabelShapeOpeningImageFilter*
- *itk::LabelStatisticsKeepNObjectsImageFilter*
- *itk::LabelStatisticsOpeningImageFilter*
- *itk::ShapeRelabelImageFilter*
- *itk::StatisticsRelabelImageFilter*

7 Binary specialization of mathematical morphology filters

- *itk::BinaryClosingByReconstructionImageFilter*
- *itk::BinaryFillholeImageFilter*
- *itk::BinaryGrindPeakImageFilter*
- *itk::BinaryOpeningByReconstructionImageFilter*
- *itk::BinaryReconstructionByDilationImageFilter*
- *itk::BinaryReconstructionByErosionImageFilter*

8 Usage examples

8.1 Prebuilt pipelines

Binary shape opening

The source code is available in the file *binary_shape_opening.cxx*.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkSimpleFilterWatcher.h"

#include "itkBinaryShapeOpeningImageFilter.h"

int main(int argc, char * argv[])
{
    if( argc != 9 )
    {
        std::cerr << "usage: " << argv[0] << " input output foreground background lambda reverseOrdering co
        // std::cerr << " : " << std::endl;
        exit(1);
    }

    const int dim = 3;

    typedef itk::Image< unsigned char, dim > IType;

    typedef itk::ImageFileReader< IType > ReaderType;
    ReaderType::Pointer reader = ReaderType::New();
    reader->SetFileName( argv[1] );

    typedef itk::BinaryShapeOpeningImageFilter< IType > BinaryOpeningType;
    BinaryOpeningType::Pointer opening = BinaryOpeningType::New();
    opening->SetInput( reader->GetOutput() );
    opening->SetForegroundValue( atoi(argv[3]) );
    opening->SetBackgroundValue( atoi(argv[4]) );
    opening->SetLambda( atof(argv[5]) );
    opening->SetReverseOrdering( atoi(argv[6]) );
    opening->SetFullyConnected( atoi(argv[7]) );
    opening->SetAttribute( argv[8] );
    itk::SimpleFilterWatcher watcher(opening, "filter");

    typedef itk::ImageFileWriter< IType > WriterType;
    WriterType::Pointer writer = WriterType::New();
    writer->SetInput( opening->GetOutput() );
    writer->SetFileName( argv[2] );
    writer->Update();
    return 0;
}
```

Statistics relabel

The source code is available in the file *statistics_relabel.cxx*.

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkSimpleFilterWatcher.h"
```

```

#include "itkStatisticsRelabelImageFilter.h"

int main(int argc, char * argv[])
{
    if( argc != 8 )
    {
        std::cerr << "usage: " << argv[0] << " input input output background useBg reverseOrdering attribute"
        // std::cerr << " : " << std::endl;
        exit(1);
    }

    const int dim = 3;

    typedef itk::Image< unsigned char, dim > IType;

    typedef itk::ImageFileReader< IType > ReaderType;
    ReaderType::Pointer reader = ReaderType::New();
    reader->SetFileName( argv[1] );

    ReaderType::Pointer reader2 = ReaderType::New();
    reader2->SetFileName( argv[2] );

    typedef itk::StatisticsRelabelImageFilter< IType, IType > RelabelType;
    RelabelType::Pointer relabel = RelabelType::New();
    relabel->SetInput( reader->GetOutput() );
    relabel->SetFeatureImage( reader2->GetOutput() );
    relabel->SetBackgroundValue( atoi(argv[4]) );
    relabel->SetUseBackground( atoi(argv[5]) );
    relabel->SetReverseOrdering( atoi(argv[6]) );
    relabel->SetAttribute( argv[7] );
    itk::SimpleFilterWatcher watcher(relabel, "filter");

    typedef itk::ImageFileWriter< IType > WriterType;
    WriterType::Pointer writer = WriterType::New();
    writer->SetInput( relabel->GetOutput() );
    writer->SetFileName( argv[3] );
    writer->Update();
    return 0;
}

```

Label shape keep N objects

The source code is available in the file *label_shape_keep_n_objects.cxx*.

```

#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkSimpleFilterWatcher.h"

#include "itkLabelShapeKeepNObjectsImageFilter.h"

```

```

int main(int argc, char * argv[])
{
    if( argc != 7 )
    {
        std::cerr << "usage: " << argv[0] << " input output background nb reverseOrdering attribute" << std::endl;
        // std::cerr << " : " << std::endl;
        exit(1);
    }

    const int dim = 3;

    typedef itk::Image< unsigned char, dim > IType;

    typedef itk::ImageFileReader< IType > ReaderType;
    ReaderType::Pointer reader = ReaderType::New();
    reader->SetFileName( argv[1] );

    typedef itk::LabelShapeKeepNObjectsImageFilter< IType > LabelOpeningType;
    LabelOpeningType::Pointer opening = LabelOpeningType::New();
    opening->SetInput( reader->GetOutput() );
    opening->SetBackgroundValue( atoi(argv[3]) );
    opening->SetNumberOfObjects( atoi(argv[4]) );
    opening->SetReverseOrdering( atoi(argv[5]) );
    opening->SetAttribute( argv[6] );
    itk::SimpleFilterWatcher watcher(opening, "filter");

    typedef itk::ImageFileWriter< IType > WriterType;
    WriterType::Pointer writer = WriterType::New();
    writer->SetInput( opening->GetOutput() );
    writer->SetFileName( argv[2] );
    writer->Update();
    return 0;
}

```

Binary fill holes

The source code is available in the file *binary_fillhole.cxx*.

```

#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkCommand.h"
#include "itkSimpleFilterWatcher.h"

#include "itkLabelObject.h"
#include "itkLabelCollectionImage.h"
#include "itkBinaryFillholeImageFilter.h"

int main(int argc, char * argv[])
{
    if( argc != 5 )

```

```

{
  std::cerr << "usage: " << argv[0] << " input output conn fg" << std::endl;
  // std::cerr << " : " << std::endl;
  exit(1);
}

const int dim = 2;

typedef itk::Image< unsigned char, dim > IType;

typedef itk::ImageFileReader< IType > ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
reader->Update();

typedef itk::BinaryFillholeImageFilter< IType > I2LType;
I2LType::Pointer reconstruction = I2LType::New();
reconstruction->SetInput( reader->GetOutput() );
reconstruction->SetFullyConnected( atoi(argv[3]) );
reconstruction->SetForegroundValue( atoi(argv[4]) );
// reconstruction->SetBackgroundValue( atoi(argv[5]) );
itk::SimpleFilterWatcher watcher(reconstruction, "filter");

typedef itk::ImageFileWriter< IType > WriterType;
WriterType::Pointer writer = WriterType::New();
writer->SetInput( reconstruction->GetOutput() );
writer->SetFileName( argv[2] );
writer->Update();
return 0;
}

```

8.2 LabelObject and LabelCollectionImage manipulation

AttributeLabelObject

The *AttributeLabelObject* let the user specify the type of the attribute he wants to use, and thus is the good choice to implement a new attribute.

The source code is available in the file *generic_attribute.cxx*.

First we include the headers of the class we will use, and parse the command line.

```

#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"

#include "itkAttributeLabelObject.h"
#include "itkLabelCollectionImage.h"

#include "itkLabelImageToLabelCollectionImageFilter.h"

#include "itkAttributeKeepNObjectsLabelCollectionImageFilter.h"
#include "itkAttributeOpeningLabelCollectionImageFilter.h"
#include "itkAttributeRelabelLabelCollectionImageFilter.h"

```

```
#include "itkLabelCollectionImageToAttributeImageFilter.h"
#include "itkLabelCollectionImageToLabelImageFilter.h"

int main(int argc, char * argv[])
{
    if( argc != 10 )
    {
        std::cerr << "usage: " << argv[0] << " label input attr keep open relabel bg lambda nb" << std::endl;
        // std::cerr << " : " << std::endl;
        exit(1);
    }
}
```

Declare the dimension used, and the type of the image for input and output.

```
const int dim = 2;
typedef unsigned char PType;
typedef itk::Image< PType, dim > IType;
```

The AttributeLabelObject class take 3 template parameters: the 2 ones from the LabelObject class, and the type of the attribute associated with each node. Here we have chosen a double. We then declares the type of the LabelCollectionImage with the type of the label object.

```
typedef itk::AttributeLabelObject< unsigned long, dim, double > L0Type;
typedef itk::LabelCollectionImage< L0Type > LCIType;
```

We read the input images.

```
typedef itk::ImageFileReader< IType > ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );

ReaderType::Pointer reader2 = ReaderType::New();
reader2->SetFileName( argv[2] );
```

And convert the label image to a LabelCollectionImage.

```
typedef itk::LabelImageToLabelCollectionImageFilter< IType, LCIType > I2LType;
I2LType::Pointer i2l = I2LType::New();
i2l->SetInput( reader->GetOutput() );
i2l->SetBackgroundValue( atoi(argv[7]) );
```

The next step is made outside the pipeline model, so we call Update() now.

```
i2l->Update();
reader2->Update();
```

Now we will valuate the attribute. The attribute will be the mean of the pixels values in the 2nd image. Note that the StatisticsLabelObject can give us that value, without having to code that by hand - that's an example.

Lets begin by declaring the iterator for the objects in the image, and get the object container, to reuse it later.

```
LCIType::LabelObjectType::const_iterator it;
LCIType::Pointer labelCollection = i2l->GetOutput();
const LCIType::LabelObjectType & labelObjectContainer = labelCollection->GetLabelObjectType();
```

Now iterate over all the objects in the image.

```
for( it = labelObjectContainer.begin(); it != labelObjectContainer.end(); it++ )
{
```

The label is there if we need it, but it can also be found at `labelObject->GetLabel()`.

```
const PType & label = it->first;
LOTType * labelObject = it->second;
```

Init the variables used for the computation.

```
double mean = 0;
unsigned long size = 0;
```

Create the iterator for the lines, and iterate over them

```
LOTType::LineContainerType::const_iterator lit;
LOTType::LineContainerType lineContainer = labelObject->GetLineContainer();

for( lit = lineContainer.begin(); lit != lineContainer.end(); lit++ )
{
    const LCIType::IndexType & firstIdx = lit->GetIndex();
    const unsigned long & length = lit->GetLength();

    size += length;
}
```

Then iterate over all the pixels in the line, and get the pixel values in the feature image to compute their mean.

```
long endIdx0 = firstIdx[0] + length;
for( LCIType::IndexType idx = firstIdx; idx[0]<endIdx0; idx[0]++ )
{
    mean += reader2->GetOutput()->GetPixel( idx );
}
```

Complete the computation of the mean, and set it as attribute value for the current object.

```
mean /= size;
labelObject->SetAttribute( mean );
```

The `LabelObject` class provides a `Print()` method to display its ivars.

```
labelObject->Print( std::cout );
{
```

Now that the objects have their attribute, we are free to manipulate them with the common filters, or by hand. The default accessor (AttributeLabelObject) is the right one when using AttributeLabelObject so we don't have to specify it. A different one can be used if needed though.

```
typedef itk::AttributeKeepNObjectsLabelCollectionImageFilter< LCIType > KeepType;
KeepType::Pointer keep = KeepType::New();
keep->SetInput( labelCollection );
keep->SetReverseOrdering( true );
keep->SetNumberOfObjects( atoi(argv[9]) );
```

Prevent the filter to run in place, so the input image is not modified.

```
keep->SetInPlace( false );

typedef itk::AttributeOpeningLabelCollectionImageFilter< LCIType > OpeningType;
OpeningType::Pointer opening = OpeningType::New();
opening->SetInput( labelCollection );
opening->SetLambda( atof(argv[8]) );
keep->SetInPlace( false );

typedef itk::AttributeRelabelLabelCollectionImageFilter< LCIType > RelabelType;
RelabelType::Pointer relabel = RelabelType::New();
relabel->SetInput( labelCollection );
keep->SetInPlace( false );
```

The attribute values can be put directly in a classic image.

```
typedef itk::LabelCollectionImageToAttributeImageFilter< LCIType, IType > A2IType;
A2IType::Pointer a2i = A2IType::New();
a2i->SetInput( labelCollection );
```

Or the label collection can be converted back to a label image, or to a binary image (not shown here)

```
typedef itk::LabelCollectionImageToLabelImageFilter< LCIType, IType > L2IType;
L2IType::Pointer l2i = L2IType::New();
```

Finally, write the results

```
typedef itk::ImageFileWriter< IType > WriterType;
WriterType::Pointer writer = WriterType::New();

writer->SetInput( a2i->GetOutput() );
writer->SetFileName( argv[3] );
writer->Update();

writer->SetInput( l2i->GetOutput() );

l2i->SetInput( keep->GetOutput() );
writer->SetFileName( argv[4] );
writer->Update();

l2i->SetInput( opening->GetOutput() );
```

```

writer->SetFileName( argv[5] );
writer->Update();

l2i->SetInput( relabel->GetOutput() );
writer->SetFileName( argv[6] );
writer->Update();

return 0;
}

```

8.3 Reading attribute values

In that example, we will read a binary image, and get some of attributes about the objects contained in that image. The source code is available in the file *attribute_values.cxx*.

First include the classes we'll use

```

#include "itkImageFileReader.h"
#include "itkShapeLabelObject.h"
#include "itkLabelCollectionImage.h"
#include "itkBinaryImageToLabelCollectionImageFilter.h"
#include "itkShapeLabelCollectionImageFilter.h"

int main(int, char * argv[])
{
    const int dim = 2;

```

then declare the type of the input image

```

typedef unsigned char PixelType;
typedef itk::Image< PixelType, dim > ImageType;

```

read the input image

```

typedef itk::ImageFileReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );

```

define the object type. Here the ShapeLabelObject type is chosen in order to read some attribute related to the shape of the objects (by opposition to the content of the object, with the StatisticsLabelObject).

```

typedef unsigned long LabelType;
typedef itk::ShapeLabelObject< LabelType, dim > LabelObjectType;
typedef itk::LabelCollectionImage< LabelObjectType > LabelCollectionType;

```

convert the image in a collection of objects

```

typedef itk::BinaryImageToLabelCollectionImageFilter< ImageType, LabelCollectionType > ConverterType;
ConverterType::Pointer converter = ConverterType::New();

```

```
converter->SetInput( reader->GetOutput() );
converter->SetForegroundValue( 200 );
```

and evaluate the attributes with the dedicated filter: `ShapeLabelCollectionImageFilter`

```
typedef itk::ShapeLabelCollectionImageFilter< LabelCollectionType > ShapeFilterType;
ShapeFilterType::Pointer shape = ShapeFilterType::New();
shape->SetInput( converter->GetOutput() );
```

update the shape filter, so its output will be up to date

```
shape->Update();
```

then we can read the attribute values we're interested in. `BinaryImageToLabelCollectionImageFilter` produces consecutive labels, so a simple for loop will do the job.

```
LabelCollectionType::Pointer collection = shape->GetOutput();
for( int label=1; label<collection->GetNumberOfObjects(); label++ )
{
    LabelObjectType::Pointer labelObject = collection->GetLabelObject( label );
    std::cout << label << "\t" << labelObject->GetPhysicalSize() << "\t" << labelObject->GetCentroid()
}

return 0;
}
```

9 Threading support

When possible, the filters provided with that contribution have been multithreaded. Some of them however, are not (easily) threadable (the `KeepNObjects` and `Relabel` filters), are shouldn't get any performance improvement in a threaded version (the `Opening` filters).

The `BinaryImageToLabelCollectionImageFilter` class is a slight modification of the Richard Beare's `ConnectedComponentImageFilter`, and thus, has not been threaded. It should however be possible to increase its performance that way.

The classical thread architecture is used when the input image is an `Image`: the image is splitted in several regions (one per thread), and each thread work on its own region.

Because the `LabelCollectionImage` image is not an array of pixels, it can't be splitted that way. Instead, several threads are created, and try to take an object in the collection. If they get one, they process that object individually, and try to get another one when the object is processed. If no object can be get, the thread ends. A `FastMutexLock` is used to ensure that only one thread take an object at a time.

For the developer, the usage of the threading support is made very simple, by subclassing `LabelCollectionImageFilter`, or `InPlaceLabelCollectionImageFilter`, and implementing the method `virtual void ThreadedGenerateData(LabelObjectType * labelObject)` in the new class. This method only has to process the `labelObject` passed in parameter. All the threading code and mutex lock management is already implemented. The mutex lock remain accessible if the subclass need to use it, as the `m_LabelObjectContainerLock` ivar.

10 In place filtering

All the filters which are taking a *LabelCollectionImage* as input, and are producing a *LabelCollectionImage* as output, are implemented as a subclass of *InPlaceLabelCollectionImageFilter* and thus are running in place by default.

The use can modify this behavior with the *SetInPlace(bool)*, *InPlaceOn()*, and *InPlaceOff()* methods, as with the usual *InPlaceImageFilter*.

To use that feature, a developer only have to subclass *InPlaceLabelCollectionImageFilter* and implement the *virtual void ThreadedGenerateData(LabelObjectType * labelObject)*, to get easy thread support³, or the *virtual void GenerateData()* if the filter is not threadable. In that last case, the only image to manipulate is the one get with the *GetOutput()* method, which is the input image if the filter runs in place, or a copy of the input image if the filter is not running in place.

11 Wrappers support

All the classes provided with that article, excepted the most generic ones made to help the developer to implement some new features, can be used with WrapITK, and have been fully tested with python.

12 Known bugs and future work

To fit the ITK style, some iterators should be implemented to be able to iterate over all the

- objects,
- lines,
- or pixels

of an image, starting from

- an image,
- an object,
- or a line.

Doing that require a good knownledge of the iterator design. Any help on that point is welcome.

Also, more attributes will be implemented, like the perimeter estimation, the usual image moments, etc. It may be useful to implement the most commonly used opening, keep N objects and relabel transforms in a more efficient way, by using an *AttributeLabelObject* instead of a *ShapeLabelObject* or a *StatisticsLabelObject*.

The computation of the moments is partially implemented. The help of someone with a good knowledge of that subject would be very helpful to complete that work. The moments for the binary shape should also be implemented.

All the *Shape* filters are currently able to use the Feret diameter only in 2D images. For the other dimensions, the feret diameter is not computed, and so always get a value of 0. The computation of the Feret diameter should be forced if the user want to use it.

The *BinaryImageToLabelCollectionImageFilter* class should be threaded to get the best of that filter on multiprocessors systems.

Finally, all the binary and label filters should be implemented as a subclass of *InPlaceImageFilter*.

³see the previous section

13 Conclusion

ITK is currently lacking a good way to manipulate the binary objects. With that contribution I hope to have mostly filled that lack.

14 Acknowledgments

I thank Richard Beare for his suggestion to use the run length encoding to represent the binary objects, and Julien Jomier for his help for the choice to *not* use the *SpatialObject* class as base class of the *LabelObject* class.

I thank Dr Pierre Adenot and MIMA2 confocal facilities (<http://mima2.jouy.inra.fr>) for providing the 3D test image. I am grateful to the INRA MIGALE bioinformatics platform (<http://migale.jouy.inra.fr>) for providing the computational resources used for the timing tests.

References

- [1] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, <http://www.itk.org/ItkSoftwareGuide.pdf>, 2003.