Label object representation and manipulation with ITK

Gaëtan Lehmann[1]

August 4, 2008

[1]INRA, 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 components 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 label images and the objects within them in ITK. Those new classes have been used to implement several label images manipulation based on object attributes, as well as the binary and label specialization of some mathematical morphology filters based on the morphological reconstruction. This contribution comes with 65 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

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 label image. There are several way to create

such 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

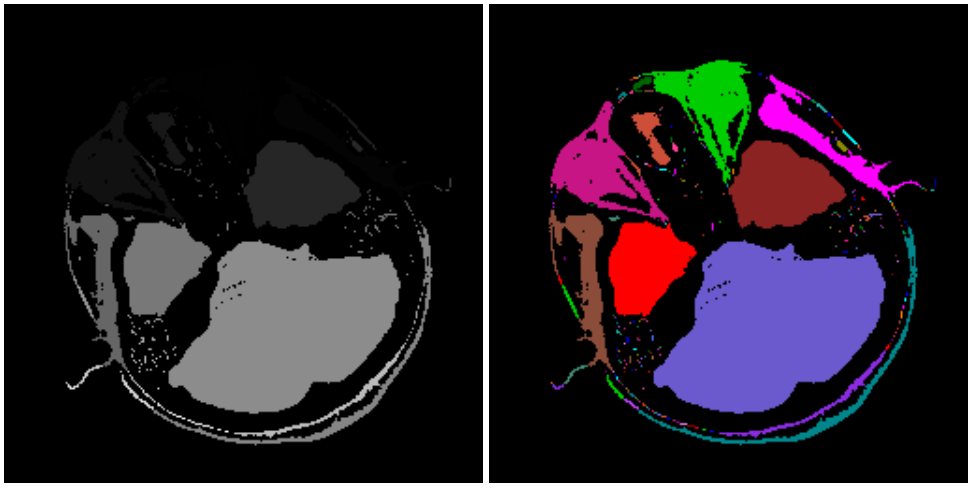
In that article, some terms will be cited very frequently. I will try to define them, in the context of the image analysis.

2.1 Label

A label is an identifier of something with the same caracteristics in the image. Those caracteristics 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 reprensent the label in memory, and its easy to find the next label by adding 1.

2.2 Label image

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



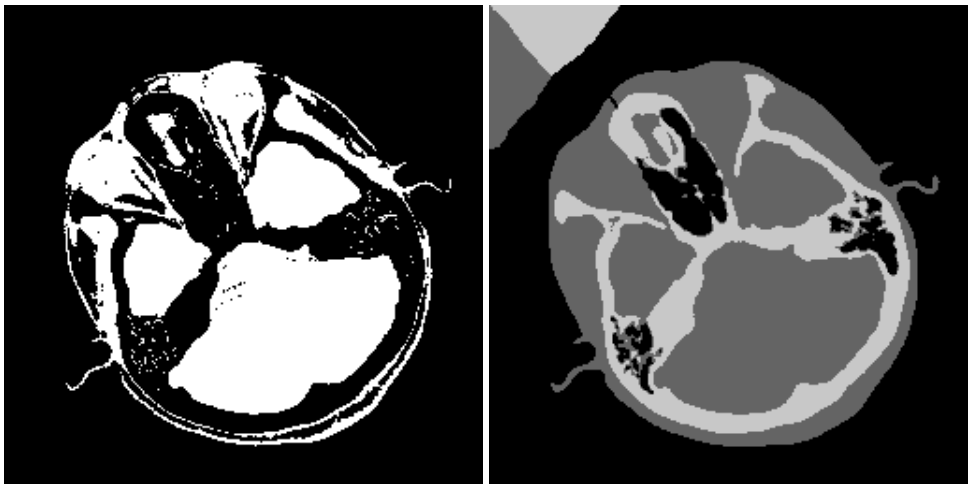
(a) A label Image. (b) The same image with colored labels

Figure 1: (a) the label image of connected components in Figure 2. (b) is the same image with labels colored with *itk::LabelToRGBImageFilter*.

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 label image can be considered as a binary image, and so, it let us manipulate a single object in a label image.



(a) A simple binary image. (b) A binary image, or a label image?

Figure 2: (a) is a simple binary image. Usually, the white pixel have the value 255, and the background the value 0. (b) contains 3 values (0, 100 and 200). The foreground value must be defined by the user, either 0, 100 or 200, and the values which are not the foreground are in the background.

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 label 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 of the class which implement 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 other 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 filters which are manipulating binary images are often prefixed with the word ”Binary”, to differenciate the grayscale versions 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 label 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 label 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, and it also store some values of the image, like its size, its spacing, etc.

4.1 itk::LabelMap

The *itk::LabelMap* class is in charge of managing the collection of label 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*. It has been chosen, to simplify the implementation and the tests, and because the feature is rarely useful in practice, to not implement the conditional background in the *itk::LabelMap* class[1]. All the images represented by a *itk::LabelMap* object have a background. If the user want to manipulate such an image with no background, he/she has to avoid the background label, for example by using a larger label type.

The *itk::LabelMap* provide a part of the API of the *itk::Image* class, and so can be manipulated as an image[2] in many cases. The performance can be very different however, because of the very different data structure used.

The *itk::LabelMap* is a templated class, which take a single parameter: the type of *label object* stored by that class. The dimension of the image is took from the *label 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 *label 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.

[1] The classes before the *revision 4* were implemented with the conditional background feature. All the related code has been removed bitween revision 3 and revision 4.

[2] It doesn’t support the itk::Image iterators though

No attribute – excepted the label – 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 label 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 label 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 label 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[3].

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.

The position in the images are given in index position when the position is the exact position of a pixel (for example, the position of the maximum value in the feature image) or in physical position when the position is given at a subpixel resolution (for example, the centroid). In both case, the position can easily be converted to the other representation with the *itk::LabelMap::TransformPhysicalPointToIndex()* and *itk::LabelMap::TransformIndexToPhysicalPoint()* methods.

Both *itk::ShapeLabelObject* and *itk::StatisticsLabelObject* are templated classes. They take the same template parameters than the *itk::LabelObject* class. The two first template parameters of the *itk::AttributeLabelObject* class or the same than the ones of 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*.

[3] In the early stage of development, all the attributes were managed as in *itk::AttributeLabelObject*, and a set of 8 classes made to manipulate a single attribute were provided, leading to a huge number of classes.

- *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 *itk::Point¡ double, ImageDimension ¿*.
- *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 *itk::ImageRegion¡ ImageDimension ¿*.
- *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. A pixel on several borders (a pixel in a corner) is counted only one time, so the size on border can't be greater than the size of the object. This attribute is particulary useful to remove the objects which are touching too much the border. Its type is *unsigned long*.
- *PhysicalSizeOnBorder* is the physical size of the objects which are on the border of the image. In 2D, it is a distance, in 3D, a surface, etc. Contrary to the *PhysicalSize* attribute which is directly linked to the *Size*, this attribute is not directly linked to the *SizeOnBorder* attribute. This attribute is particulary useful to remove the objects which are touching too much the border. Its type is *double*.
- *FeretDiameter* is the diameter in physical units of the sphere which include all the object. The feret diameter is not computed by default, because of its high computation. Its type is *double*.
- *BinaryPrincipalMoments* contains the principal moments. Its type is *itk::Vector¡ double, ImageDimension ¿*.
- *BinaryPrincipalAxes* contains the principal axes of the object. Its type is *itk::Matrix¡ double, ImageDimension, ImageDimension ¿*.
- *BinaryElongation* is the elongation of the shape, computed as the ratio of the largest principal moment by the smallest principal moment. Its value is greater or equal to 1. Its type is *double*.
- *EquivalentRadius* is the equivalent radius of the hypersphere of the same size than the label object. The value depends on the image spacing. Its type is *double*.
- *EquivalentPerimeter* is the equivalent perimeter of the hypersphere of the same size than the label object. The value depends on the image spacing. Its type is *double*.
- *EquivalentEllipsoidPerimeter* is the size of the ellipsoid of the same size and the same ratio on all the axes than the label object. The value depends on the image spacing. Its type is *itk::Vector¡ double, ImageDimension ¿*.

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*itk::Index¡ ImageDimension ¿*.
- *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*itk::Index¡ ImageDimension ¿*.
- *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 obejct. Its type is *double*
- *CenterOfGravity* is the center of gravity of the object. Its type is *itk::Point¡ double ¿*.
- *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*.
- *PrincipalMoments* contains the principal moments. Its type is *itk::Vector¡ double, ImageDimension ¿*.
- *PrincipalAxes* contains the principal axes of the object. Its type is *itk::Matrix¡ double, ImageDimension, ImageDimension ¿*.
- *Elongation* is the elongation of the shape, computed as the ratio of the largest principal moment by the smallest principal moment. Its value is greater or equal to 1 Its type si *double*.
- *Histogram* is the histogram of the pixels covered by the label object in the feature image. Its type is *itk::Histogram¡ 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::LabelMap

The *itk::LabelMap* class provide some methods to fill the image ”by hand”, like the usual *SetPixel()* method. However, the most efficient way is to convert a label image or a binary image stored in an *itk::Image* to a *itk::LabelMap*, by using *itk::BinaryImageToLabelMapFilter* or *itk::LabelImageToLabelMapFilter*.

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::ShapeLabelMapFilter* to fill the attributes of the *itk::ShapeLabelObject*s,
- and *itk::StatisticsLabelMapFilter* to fill the attributes of the *itk::StatisticsLabelObject*s.

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

5.3 Manipulating the itk::LabelMap

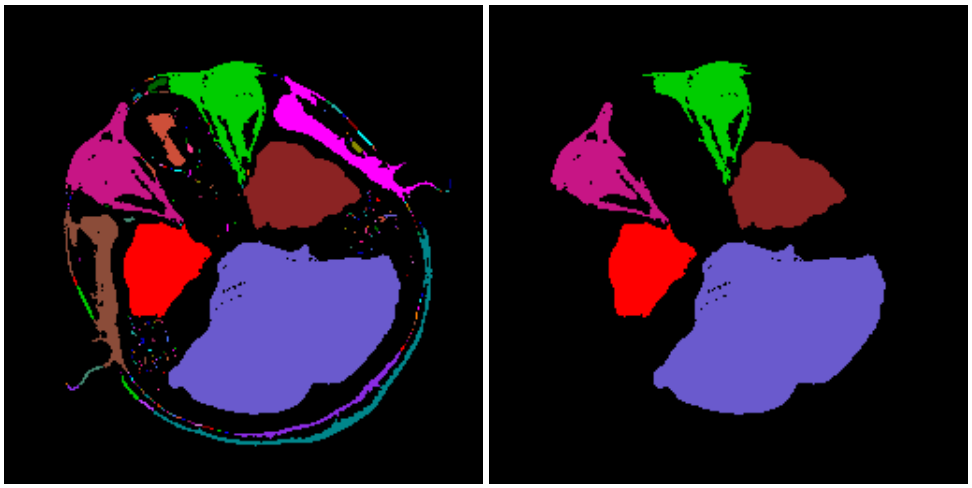
Once created and, optionally, valuated, several filters are provided to manipulate the *itk::LabelMap*:

- An opening can be performed with the *OpeningLabelMapFilter* 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. The class *AttributeSelectionLabelMapFilter* and its subclass *LabelSelectionLabelMapFilter* can be used to remove some objects based on their attribute value, even if the attribute type has no ordering property.
- A fixed number of objects can be kept, with the *KeepNObjectsLabelMapFilter* 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 relabeled, with the *RelabelLabelMapFilter* 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.
- The region covered by the *itk::LabelMap* can be changed with *itk::ChangeRegionLabelMapFilter* and its subclasses (*itk::CropLabelMapFilter*, *itk::PadLabelMapFilter*, *RegionFromReferenceLabelMapFilter* and *itk::AutoCropLabelMapFilter*).

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.

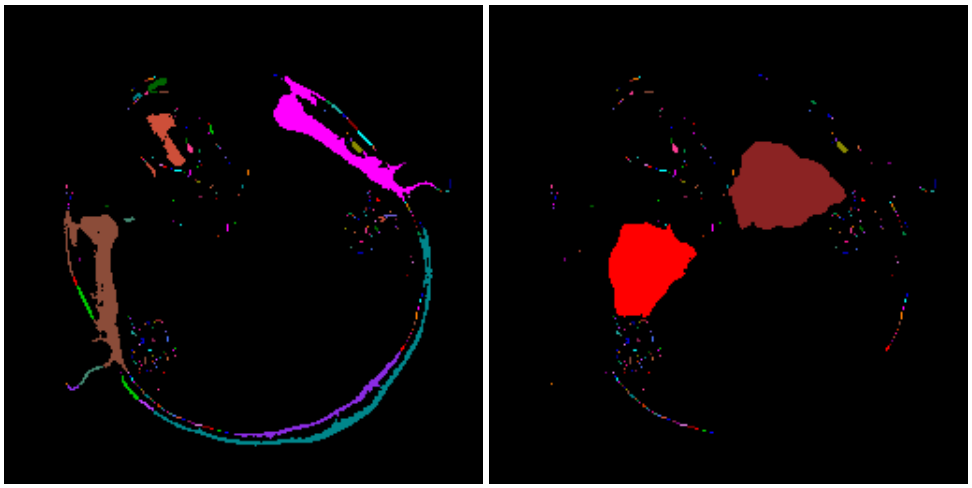
5.4 Generating an itk::Image from the itk::LabelMap

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:



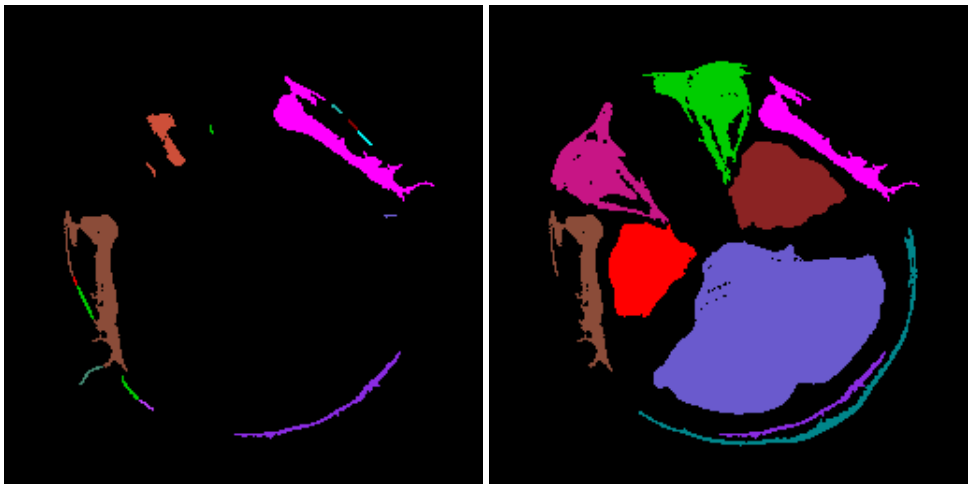
(a) A label image

(b) All the objects smaller than 1000 pixels removed



(c) All the objects greater than 1000 pixels removed

(d) All the objects with roundness smaller than 0.8 removed



(e) All the objects with elongation smaller than 10 removed

(f) All the objects with perimeter smaller than 100 removed

Figure 3: Some example of opening with different attribute and parameters. Note that the labels are kept unchanged in the output image.

- The *itk::LabelMapToLabelImageFilter* class simply convert a *itk::LabelMap* to a label image stored in a *itk::Image*.
- The *itk::LabelMapToBinaryImageFilter* 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::BinaryImageToLabelMapFilter*. The background values of the original image can also be restored by this filter.
- The *itk::LabelMapMaskImageFilter* class can be used to mask an image with the objects of the *itk::LabelMap*. With that filter, the image can be cropped to contain only the non-masked zone[4] , or the non-masked zone padded by a user defined number of pixels.
- The *itk::LabelMapToAttributeImageFilter* produce an *itk::Image* with the value of the attribute of the objects of the *itk::LabelMap*. This filter is mostly useful to have a global view of the attribute values in the image.
- The *LabelMapToRGBImageFilter* produce a color *itk::Image* with *itk::RGBPixel* as pixel type. The label objects are in color, as in Figure 1. This class is mostly useful for a quick visual validation without going outside ITK.
- Finally, the *LabelMapOvelayImageFilter* produce a color *itk::Image* with *itk::RGBPixel* as pixel type. The label objects are in color on top of a grayscale image, as in Figure 5. This class is mostly useful for a quick visual validation without going outside ITK.

6 Prebuilt mini-pipeline filters

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

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

with a specific attribute.

Also, when using the *itk::ShapeLabelObject* or the *itk::StatisticsLabelObject* class, we usually want them to be valuated. Some classes are provided to perform the mini-pipeline:

- creation of the *itk::LabelMap* from an *itk::Image*,
- valuation of the attributes of the objects

with the *itk::ShapeLabelObject* or the *itk::StatisticsLabelObject*.

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

[4] The code used to produce the output region based on the content of the image is partially copied from a contribution of Peter Cech http://www.vision.ee.ethz.ch/~pcech/itkAutoCropImageFilter/.

6.1 Binary filters

The filters to produce valuated attributes:

- *itk::BinaryImageToShapeLabelMapFilter*
- *itk::BinaryImageToStatisticsLabelMapFilter*

The filters to fully hide the usage of the label objects:

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

6.2 Label filters

The filters to produce valuated attributes:

- *itk::LabelImageToShapeLabelMapFilter*
- *itk::LabelImageToStatisticsLabelMapFilter*

The filters to fully hide the usage of the label objects:

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

7 Binary and label specialization of mathematical morphology filters

All the following filters are using a morphological reconstruction, implemented internally as an attribute opening with the *itk::AttributeLabelObject* class. Some binary filters are implemented:

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

and some label filters, useful when the label objects are connected, or when we want to avoid loosing the label by using a binary filter. Because the notion of reconstruction by erosion is difficult with labels, only a few filters are implemented:

- *itk::LabelReconstructionByDilationImageFilter*

8 Computation details

8.1 Binary image moments

Central image moments for grayscale images are usually computed as

$$Cm_{i,j} = \frac{S_{i,j}}{M} - Cg_i.Cg_j \quad (1)$$

$$S_{i,j} = \sum_{p \in D} (I(p).p_i.p_j) \quad (2)$$

where $S_{i,j}$ is the central moment, D is the domain of definition of the image I, $I(p)$ is the pixel value of the image at the position p, $0 \leq i < n$, $0 \leq i < n$, n is the image dimension, p_i is the physical position on the axis i, M is the total mass, Cg is the center of gravity. With binary images, $I(p)$ is either 0 if p is outside the object, or 1 if p is inside.

The complexity is $O(N_{P_I})$, where N_{P_I} is the number of pixels in the image.

With the run-length encoding of the binary objects, the complexity can be decreased to $O(N_{L_O})$, where N_{L_O} is the number of lines in the object.

$$\begin{aligned} S_{i,j} &= \sum_{p \in D} (I(p).p_i.p_j) \quad (3) \\ &= \sum_{p \in O} (p_i.p_j) \\ &= \sum_{L \in O} \sum_{p \in L} (p_i.p_j) \end{aligned}$$

Where O is a binary object in the image, and L is a line of the object O, encoded with the run-length encoding. In a line, p_i is a constant if $i > 0$, so, $S_{i,j}$ can be written:

$$S_{i,j} = \begin{cases} \sum_{L \in O} (l_L . p_i . p_j) & \text{if } i > 0 \text{ and } j > 0, \\ \sum_{L \in O} \left(p_i . \sum_{p \in L} p_0 \right) & \text{if } i > 0 \text{ and } j = 0, \\ \sum_{L \in O} \left(p_j . \sum_{p \in L} p_0 \right) & \text{if } i = 0 \text{ and } j > 0, \\ \sum_{L \in O} \sum_{p \in L} p_0^2 & \text{if } i = j = 0. \end{cases} \tag{4}$$

It is known that

$$\sum_{x=0}^{n} x = \frac{n(n+1)}{2} \tag{5}$$

$$\sum_{x=0}^{n} x^2 = \frac{n(n+1)(2n+1)}{6} \tag{6}$$

In order to use those formulae in the computation of $S_{i,j}$, the physical position has to be expanded in:

$$p_j = o_j + s_j i_j \tag{7}$$

where o_j is the origin of the line on the axis j, s_j is the spacing on the axis j, and i_j is the index on the axis j.

With equations 7, 5 and 6, it is easy to remove the loop in the computation of the sum of physical positions of the axis 0:

$$\begin{aligned} \sum_{p \in L} p_0 &= \sum_{i_0=0}^{l_L - 1} (o_0 + s_0 i_0) \\ &= l_L . o_0 + s_0 \sum_{i_0=0}^{l_L - 1} i_0 \\ &= l_L . o_0 + s_0 \left(\frac{(l_L - 1) l_L}{2} \right) \\ &= l_L \left(o_0 + \frac{s_0 (l_L - 1)}{2} \right) \end{aligned} \tag{8}$$

where l_L is the length of the line (in pixels), and in the sum of the square of the physical positions of the axis 0:

$$\begin{aligned}
\sum_{p\in L} p_0^2 &= \sum_{i_0=0}^{l_L-1} (o_0 + s_0 i_0)^2 \\
&= \sum_{i_0=0}^{l_L-1} (o_0^2 + s_0^2 i_0^2 + 2 o_0 s_0 i_0) \\
&= l_L.o_0^2 + s_0^2 \sum_{i_0=0}^{l_L-1} i_0^2 + 2 o_0 s_0 \sum_{i_0=0}^{l_L-1} i_0 \\
&= l_L.o_0^2 + s_0^2 \left(\frac{(l_L-1) l_L (2 l_L - 1)}{6} \right) + 2 o_0 s_0 \left(\frac{(l_L-1) l_L}{2} \right) \\
&= l_L \left(o_0^2 + s_0 (l_L - 1) \left(\frac{s_0 (2 l_L - 1)}{6} + o_0 \right) \right)
\end{aligned} \tag{9}$$

Finally, $S_{i,j}$, used in the computation of the central moments, is computed as:

$$S_{i,j} = \begin{cases}
\sum\limits_{L\in O} (l_L.p_i.p_j) & \text{if } i > 0 \text{ and } j > 0, \\
\sum\limits_{L\in O} \left(p_i.l_L \left(o_0 + \frac{s_0 (l_L - 1)}{2} \right) \right) & \text{if } i > 0 \text{ and } j = 0, \\
\sum\limits_{L\in O} \left(p_j.l_L \left(o_0 + \frac{s_0 (l_L - 1)}{2} \right) \right) & \text{if } i = 0 \text{ and } j > 0, \\
\sum\limits_{L\in O} \left(l_L \left(o_0^2 + s_0 (l_L - 1) \left(\frac{s_0 (2 l_L - 1)}{6} + o_0 \right) \right) \right) & \text{if } i = j = 0.
\end{cases} \tag{10}$$

8.2 Roundness

The computation works with any image dimension, and so use the definition of volume and area of an hypersphere in any dimension.

$$V_n(r) = \frac{\pi^{\frac{n}{2}} r^n}{\Gamma(\frac{n}{2} + 1)} \tag{11}$$

where V_n is the volume of the hypersphere n is the image dimension, and r is the radius of the hypersphere.

$$\Gamma\left(\frac{n}{2} + 1\right) = \begin{cases}
\left(\frac{n}{2}\right)! & \text{if } n \text{ is even}, \\
\sqrt{\pi} \frac{n!!}{2^{(n+1)/2}} & \text{if } n \text{ is odd}.
\end{cases} \tag{12}$$

$n!!$ is the *double factorial*, defined as:

$$n!! = \begin{cases}
1 & \text{if } n < 2, \\
n(n-2)!! & \text{if } n \geq 2.
\end{cases} \tag{13}$$

$$A_n(r) = \frac{n V_n}{r} \tag{14}$$

where A_n is the area of the hypersphere.

$$R = \frac{A_n(r)}{a} \tag{15}$$

where R is the roundness, a is the measured area of the object[5], and the r is the radius of an hypersphere with the same volume than the object, computed using equation 11.

8.3 Pixel's neighborhood

The run-length encoding does not allow an easy access to the neighbors of a given pixel. If the neighborhood of all the pixels must be accessed, it is much easier to convert the itk::LabelMap to a *itk::Image* with the *itk::LabelMapToLabelImageFilter*, and use that image to access the neighbors. That's what is done in *itk::ShapeLabelMapFilter* to compute the maximum Feret diameter, and the perimeter estimation.

9 Usage examples

9.1 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();
```

[5] More details about perimeter estimation will be published in another article.

```
  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 attribut
    // 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->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 obejcts

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" << st
    // 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 "itkLabelMap.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;
}
```

9.2 LabelObject and LabelMap manipulation

AttributeLabelObject

The *itk::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 "itkLabelMap.h"

#include "itkLabelImageToLabelMapFilter.h"

#include "itkAttributeKeepNObjectsLabelMapFilter.h"
#include "itkAttributeOpeningLabelMapFilter.h"
#include "itkAttributeRelabelLabelMapFilter.h"

#include "itkLabelMapToAttributeImageFilter.h"
#include "itkLabelMapToLabelImageFilter.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 LabelMap with the type of the label object.

```
typedef itk::AttributeLabelObject< unsigned long, dim, double > LabelObjectType;
typedef itk::LabelMap< LabelObjectType > LabelMapType;
```

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 LabelMap.

```
typedef itk::LabelImageToLabelMapFilter< IType, LabelMapType > 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.

```
LabelMapType::LabelObjectContainerType::const_iterator it;
LabelMapType::Pointer labelMap = i2l->GetOutput();
const LabelMapType::LabelObjectContainerType & labelObjectContainer = labelMap->GetLabelObjectContair
```

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;
  LabelObjectType * 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

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

for( lit = lineContainer.begin(); lit != lineContainer.end(); lit++ )
  {
  const LabelMapType::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( LabelMapType::IndexType idx = firstIdx; idx[0]<endIdx0; idx[0]++)
    {
    mean += reader2->GetOutput()->GetPixel( idx );
    }
  }
```

Complete the compuation of the mean, and set it as attibute 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 wright one when using AttributeLabelObject so we don't have to specify it. A different one can be used if needed though.

```
typedef itk::AttributeKeepNObjectsLabelMapFilter< LabelMapType > KeepType;
KeepType::Pointer keep = KeepType::New();
keep->SetInput( labelMap );
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::AttributeOpeningLabelMapFilter< LabelMapType > OpeningType;
OpeningType::Pointer opening = OpeningType::New();
opening->SetInput( labelMap );
opening->SetLambda( atof(argv[8]) );
opening->SetInPlace( false );

typedef itk::AttributeRelabelLabelMapFilter< LabelMapType > RelabelType;
RelabelType::Pointer relabel = RelabelType::New();
relabel->SetInput( labelMap );
relabel->SetInPlace( false );
```

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

```
  typedef itk::LabelMapToAttributeImageFilter< LabelMapType, IType > A2IType;
  A2IType::Pointer a2i = A2IType::New();
  a2i->SetInput( labelMap );
```

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

```
  typedef itk::LabelMapToLabelImageFilter< LabelMapType, 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;
}
```

Custom attribute accessor

This example shows how to use the LabelMap classes to remove all the object with a bounding size smaller (or greater) than a given value on the z axis. The attribute we are interested is already computed by *ShapeLabelMapFilter* and stored in *ShapeLabelObject*. It is not usable as is however, because it is part of a multicomponent attribute: *Region*.

To be able to use it with the standard classes, we have to provide an accessor which can be used by the opening filter.

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

First include the classes we'll use

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

#include "itkLabelImageToShapeLabelMapFilter.h"
```

```
#include "itkAttributeOpeningLabelMapFilter.h"
#include "itkLabelMapToLabelImageFilter.h"
```

Now we can declare the custom accessor type, which will be used by the opening filter.

```
template< class TLabelObject >
class ITK_EXPORT LastDimesionRegionSizeObjectAccessor
{
public:
```

The declaration of *AttributeValueType* is mandatory. It is used internally in the opening filter to define the type of the lambda value. *AttributeValueType* should be the same as the type returned by the operator() method.

```
  typedef unsigned long AttributeValueType;
```

operator() is the core of the accessor. It takes a label object as parameter, and return the attribute of interest. Some computations can be done inside the accessor method, but they should be as fast as possible, because *operator()* may be called many time by some filters. If an attribute value takes time to compute, it should rather be computed once for all and stored in a new attribute. In that case, there is no computation at all, only an access to a value hidden in a multicomponent attribute, so things are very fast.

```
  inline const AttributeValueType operator()( const TLabelObject * labelObject )
    {
    return labelObject->GetRegion().GetSize()[TLabelObject::ImageDimension-1];
    }
};
```

Now the main program. First the usual validation of the number of arguments.

```
int main(int argc, char * argv[])
{

  if( argc != 6 )
    {
    std::cerr << "usage: " << argv[0] << " input output bg lambda reverse" << std::endl;
    // std::cerr << "  : " << std::endl;
    exit(1);
    }
```

Let's declare the dimension used, and the type of the input image

```
  const int dim = 3;
  typedef unsigned char PType;
  typedef itk::Image< PType, dim > IType;
```

We read the input image.

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

And convert it to a *LabelMap*, with the shape attributes computed. We use the default label object type provided by *LabelImageToShapeLabelMapFilter*.

```
  typedef itk::LabelImageToShapeLabelMapFilter< IType > I2LType;
  I2LType::Pointer i2l = I2LType::New();
  i2l->SetInput( reader->GetOutput() );
  i2l->SetBackgroundValue( atoi(argv[3]) );
```

The opening filter is declared with our custom accessor type as second template argument, so it will be able to user our custom attribute to make the opening.

```
  typedef LastDimesionRegionSizeObjectAccessor< I2LType::LabelObjectType > AccessorType;
  typedef itk::AttributeOpeningLabelMapFilter< I2LType::OutputImageType, AccessorType > OpeningType;
  OpeningType::Pointer opening = OpeningType::New();
  opening->SetInput( i2l->GetOutput() );
  opening->SetLambda( atof(argv[4]) );
  opening->SetReverseOrdering( atof(argv[5]) );
  itk::SimpleFilterWatcher watcher(opening, "filter");
```

The label map is then converted back to an label image.

```
  typedef itk::LabelMapToLabelImageFilter< I2LType::OutputImageType, IType > L2IType;
  L2IType::Pointer l2i = L2IType::New();
  l2i->SetInput( opening->GetOutput() );
```

Write the result to the disk.

```
  typedef itk::ImageFileWriter< IType > WriterType;
  WriterType::Pointer writer = WriterType::New();
  writer->SetInput( l2i->GetOutput() );
  writer->SetFileName( argv[2] );
  writer->Update();
```

Finally, print all the label objects after the opening, to check everything has been done right.

```
  opening->GetOutput()->PrintLabelObjects();
  std::cout << "Number of objects after the opening: " << opening->GetOutput()->GetNumberOfLabelObjects

  return 0;
}
```

9.3 Reading attribute values

In that example, we will read a binary image, and get some of attributes about the obejcts 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 "itkLabelMap.h"
#include "itkBinaryImageToLabelMapFilter.h"
#include "itkShapeLabelMapFilter.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 StatisticsLabelObejct).

```
  typedef unsigned long LabelType;
  typedef itk::ShapeLabelObject< LabelType, dim > LabelObjectType;
  typedef itk::LabelMap< LabelObjectType > LabelMapType;
```

convert the image in a collection of objects

```
  typedef itk::BinaryImageToLabelMapFilter< ImageType, LabelMapType > ConverterType;
  ConverterType::Pointer converter = ConverterType::New();
  converter->SetInput( reader->GetOutput() );
  converter->SetForegroundValue( 200 );
```

and valuate the attributes with the dedicated filter: ShapeLabelMapFilter

```
  typedef itk::ShapeLabelMapFilter< LabelMapType > 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. *itk::BinaryImageToLabelMapFilter* produces consecutives labels, so a simple *for* loop will do the job.

```
  LabelMapType::Pointer labelMap = converter->GetOutput();
  for( unsigned int label=1; label<=labelMap->GetNumberOfLabelObjects(); label++ )
    {
    // we don't need a SmartPointer of the label object here, because the reference is kept in
    // in the label map.
    const LabelObjectType * labelObject = labelMap->GetLabelObject( label );
    std::cout << label << "\t" << labelObject->GetPhysicalSize() << "\t" << labelObject->GetCentroid()
    }

  return 0;
}
```

9.4 The mask features

The *itk::LabelMapMaskImageFilter* class let the user mask a part of an *itk::Image* with the objects of a *itk::LabelMap*. It can also crop the image to contain only the masked region.

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

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

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

#include "itkLabelObject.h"
#include "itkLabelMap.h"
#include "itkLabelImageToLabelMapFilter.h"
#include "itkLabelMapMaskImageFilter.h"


int main(int argc, char * argv[])
{

  if( argc != 9 )
    {
    std::cerr << "usage: " << argv[0] << " labelImage input output label bg neg crop cropBorder" << st
    // std::cerr << "  : " << std::endl;
    exit(1);
    }
```

the filters are able to work in any dimension. Lets choose 3, so the program can be tested on 2D and 2D image.

```
  const int dim = 3;
```

declare the input image type

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

and the label object type to use. The input image is a label image, so the type of the label can be the same type than the pixel type. itk::LabelObject is chosen, because only the mask feature is tested here, so we don't need any attribute.

```
  typedef itk::LabelObject< unsigned char, dim > LabelObjectType;
  typedef itk::LabelMap< LabelObjectType > LabelMapType;
```

read the label image and the input image to be masked.

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

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

convert the label image to a label collection image.

```
  typedef itk::LabelImageToLabelMapFilter< ImageType, LabelMapType> I2LType;
  I2LType::Pointer i2l = I2LType::New();
  i2l->SetInput( reader->GetOutput() );
```

then mask the image. Two inputs are required (the label collection image, and the image to be masked). The label used to mask the image is passed with the *SetLabel()* method. The background in the output image, where the image is masked, is passed with *SetBackground()*. The user can choose to mask the image outside the label object (that's the default behavior), or inside the label object with the chosen label, by calling *SetNegated()*. Finally, the image can be cropped to the masked region, by calling *SetCrop(true)*, or to a region padded by a border, by calling both *SetCrop()* and *SetCropBorder()*. The crop border defaults to 0, and the image is not cropped by default.

```
  typedef itk::LabelMapMaskImageFilter< LabelMapType, ImageType > MaskType;
  MaskType::Pointer mask = MaskType::New();
  mask->SetInput( i2l->GetOutput() );
  mask->SetFeatureImage( reader2->GetOutput() );
  mask->SetLabel( atoi(argv[4]) );
  mask->SetBackgroundValue( atoi(argv[5]) );
  mask->SetNegated( atoi(argv[6]) );
  mask->SetCrop( atoi(argv[7]) );
  MaskType::SizeType border;
  border.Fill( atoi(argv[8]) );
  mask->SetCropBorder( border );
  itk::SimpleFilterWatcher watcher6(mask, "filter");
```

Finally, save the output image.

```
  typedef itk::ImageFileWriter< ImageType > WriterType;
  WriterType::Pointer writer = WriterType::New();
  writer->SetInput( mask->GetOutput() );
  writer->SetFileName( argv[3] );
  writer->Update();

  return 0;
}
```

9.5 A full python example

In that example, we want to:

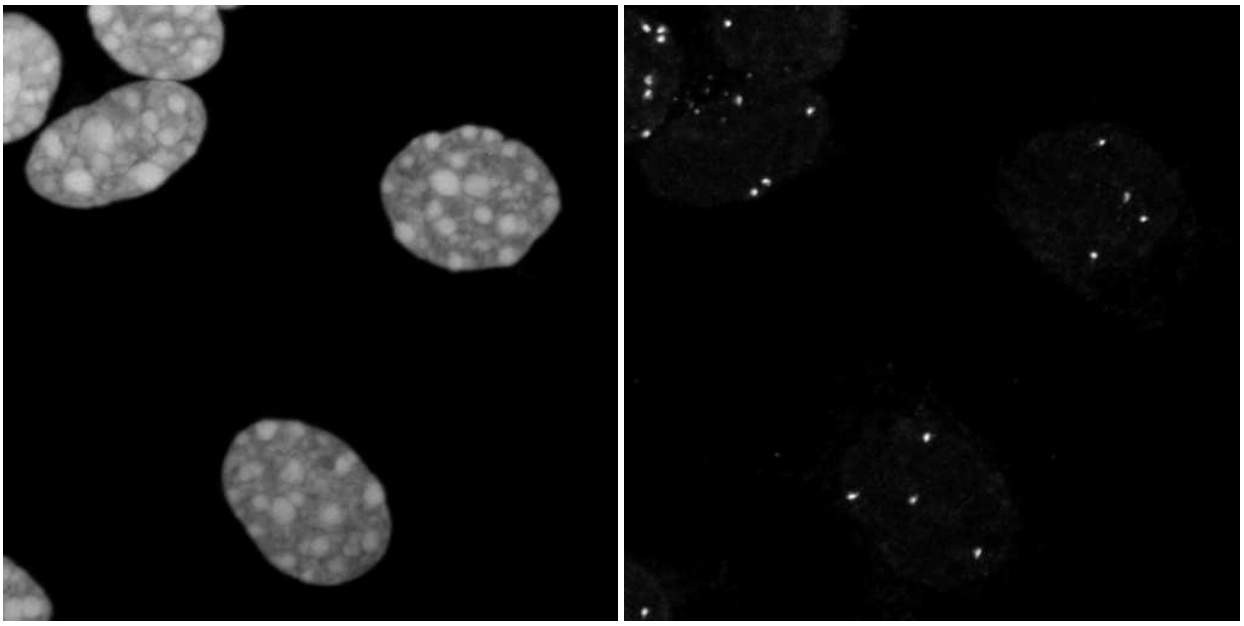
- find the nuclei in the first image
- find the spots insice the nucleus in the second image
- get the mean value in the nucleus, in the zone of each spot.

The source code is available in example.py.

Lets begin with the usual *import*s.

```
import itk, sys
itk.auto_progress()
```

Then declare the type we will use, as in *C++*.



(a) Nucleus (b) Spots

Figure 4: The input images.

```
Dimension = 2
PixelType = itk.UC
ImageType = itk.Image[ PixelType, Dimension ]

DistancePixelType = itk.F
DistanceImageType = itk.Image[ DistancePixelType, Dimension ]

RGBPixelType = itk.RGBPixel[PixelType]
RGBImageType = itk.Image[ RGBPixelType, Dimension ]

LabelObjectType = itk.StatisticsLabelObject[itk.UL, Dimension]
LabelMapType = itk.LabelMap[LabelObjectType]
```

read the image of the nucleus

```
nuclei = itk.ImageFileReader[ImageType].New(FileName="images/noyaux.png")
```

perform a simple binarization. Note that the Otsu filter does not use the same convention as usual: the white part is outside.

```
otsu = itk.OtsuThresholdImageFilter[ImageType, ImageType].New(nuclei, OutsideValue=255,
    InsideValue=0)
```

The nuclei are not separated. We split them with a watershed.

```
maurer = itk.SignedMaurerDistanceMapImageFilter[ImageType, DistanceImageType].New(otsu)
watershed = itk.MorphologicalWatershedImageFilter[DistanceImageType, ImageType].New(maurer,
    Level=60, MarkWatershedLine=False)
mask = itk.MaskImageFilter[ImageType, ImageType, ImageType].New(watershed, otsu)
```

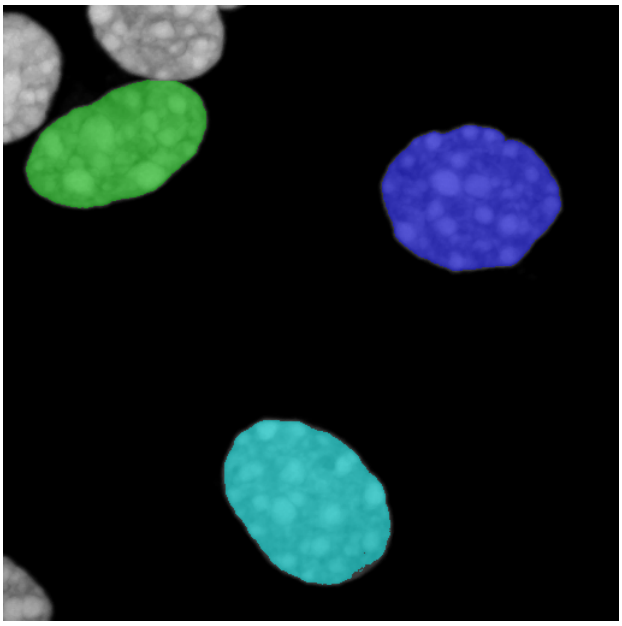


Figure 5: The segmented nuclei. The too small objects and the ones on the border have been excluded.

And now switch to the label map representation, and compute the attribute values

```
stats = itk.LabelImageToStatisticsLabelMapFilter[ImageType, ImageType, LabelMapType].New(mask,
    nuclei)
```

drop the objects too small to be a nucleus, and the ones on the border

```
size = itk.ShapeOpeningLabelMapFilter[LabelMapType].New(stats,Attribute='Size',
    Lambda=100)
border = itk.ShapeOpeningLabelMapFilter[LabelMapType].New(size, Attribute='SizeOnBorder',
    Lambda=10, ReverseOrdering=True)
```

Reoder the labels. The objects with the highest mean are the first ones.

```
relabel = itk.StatisticsRelabelLabelMapFilter[LabelMapType].New(border, Attribute='Mean')
```

for visual validation:

```
overlay = itk.LabelMapOverlayImageFilter[LabelMapType, ImageType, RGBImageType].New(relabel,
    nuclei)
itk.write(overlay, "nuclei-overlay.png")
```

Now, the spots:

```
spots = itk.ImageFileReader[ImageType].New(FileName="images/spots.png")
```

Mask the spot image to keep only the nucleus zone. The rest of the image is cropped, excepted a border of 2 pixels

```
maskSpots = itk.LabelMapMaskImageFilter[LabelMapType, ImageType].New(relabel, spots, Label=1,
    Crop=True, CropBorder=2)
```

A simple thresholding:

```
th = itk.BinaryThresholdImageFilter[ImageType, ImageType].New(maskSpots, LowerThreshold=110)
```

Now swith to the label map representation, and compute the attribute values. This time, the input image is not a label image, but a binary one.

```
sstats = itk.BinaryImageToStatisticsLabelMapFilter[ImageType, ImageType, LabelMapType].New(th,
    nuclei)
```

we know there are 4 spots in the nubleus, so keep the 4 biggest spots. The other attributes are also usable - we may have chosen to keep the 4 brightest spots for example.

```
skeep = itk.ShapeKeepNObjectsLabelMapFilter[LabelMapType].New(sstats, Attribute='Size',
    NumberOfObjects=4)
```

Reoder the labels. The bigger objects first.

```
srelabel = itk.StatisticsRelabelLabelMapFilter[LabelMapType].New(skeep, Attribute='Size')
```

Finally, display the values we are interested in:

- the nucleus number,
- the spot position,
- the mean value in the nucleus in the spot zone.

```
print "nuclei", "x", "y", "mean"

for nl in range(1, relabel.GetOutput().GetNumberOfLabelObjects()+1):
  maskSpots.SetLabel(nl)
  srelabel.UpdateLargestPossibleRegion()
  labeCollection = srelabel.GetOutput()

  for l in range(1, labeCollection.GetNumberOfLabelObjects()+1):
    lo = labeCollection.GetLabelObject(l)
    print nl, lo.GetCentroid()[0], lo.GetCentroid()[1], lo.GetMean()
```

10 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), or shouldn't get any performance improvement in a threaded version (the *Opening* filters).

The *itk::BinaryImageToLabelMapFilter* class is a slight modification of the Richard Beare's *itk::ConnectedComponentImageFilter*, and have also been threaded to get the best of the performances on a multicore system.

nucleus	x	y	mean
1	117.925925926	146.111111111	188.185185185
1	154.25	87.4166666667	126.416666667
1	107.666666667	155.125	122.0
1	95.2380952381	78.2857142857	121.0
2	417.631578947	158.736842105	132.894736842
2	431.277777778	177.388888889	131.222222222
2	390.117647059	207.588235294	96.8235294118
2	396.8	113.666666667	113.2
3	251.148148148	358.814814815	105.037037037
3	189.333333333	407.888888889	111.074074074
3	293.72	454.8	95.48
3	239.888888889	411.111111111	135.222222222

Table 1: Output of the python example.

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

Because the *itk::LabelMap* 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 *itk::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 *itk::LabelMapFilter*, or *itk::InPlaceLabelMapFilter*, 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.

11 In place filtering

All the filters which are taking a *itk::LabelMap* as input, and are producing a *itk::LabelMap* as output, are implemented as a subclass of *InPlaceLabelMapFilter* 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 *InPlaceLabelMapFilter* and implement the *virtual void ThreadedGenerateData(LabelObjectType * labelObject)*, to get easy thread support [6], 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.

12 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 both stable and unstable WrapITK, and have been fully tested with python.

[6] see the previous section

13 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 knowledge of the iterator design. Any help on that point is welcome.

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 *itk::AttributeLabelObject* instead of a *itk::ShapeLabelObject* or a *itk::StatisticsLabelObject*.

The converters from/to image are provided, but it may be useful to have the converters from/to other objects representations:

- spatial objects,
- meshs,
- structuring elements.

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

Note that some names have been changed in revision 3. To fix your sources, the following commands can be used in your source directory (order is important):

```
perl -e 's/LabelCollectionImage/LabelMap/g' -pi *
perl -e 's/GetNumberOfObjects/GetNumberOfLabelObjects/g' -pi *
perl -e 's/PrintObjects/PrintLabelObjects/g' -pi *
perl -e 's/LabelMapToMaskImageFilter/LabelMapMaskImageFilter/g' -pi *
```

14 Conclusion

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

15 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 *itk::SpatialObject* class as base class of the *itk::LabelObject* class.

I thank Dr Pierre Adenot and MIMA2 confocal facilities (http://mima2.jouy.inra.fr) for providing the 3D test image. I thank Dr Maria Ballester for providing the image used in the python example.

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.