Filtering with ITK
Some notes

February 15, 2020

Contents

List of Figures

List of Tables

List of Equations

Listings

1. What is this session about?

In this session we would be practising different filtering capabilities provided by ITK. We will refer a lot to the chapter on "Filtering" you can find in the ITK software guide [1, 2] (`InsightSoftwareGuide-Book2.pdf`)[1]. Filters are processing objects that accept one or more images as input and return one or more images as output. Firstly, we will practice some filters that try to adapt different pixel types between images, together with filters that additionally adapt the intensity range for the output image.

As we have seen in a previous theory lecture, the concept of neighborhood is frequently used to define the operations (filters) applied in the spatial domain. This way we used the following formula:

$$g(x, y) = T[f(x, y)]$$

where $f(x, y)$ represents the input image, $g(x, y)$ the output image,and T is an operator on f defined over a neighborhood of point $(x, y) \in \mathbb{Z}^2$. Neighborhood is frequently used by filters which compute every output pixel value using the pixel values in a neighborhood of the current pixel. To understand how ITK treats this concept we will practice the mean and median filter.

Noise is inherently related to medical image. In the last section we will practice different filters that ITK provides to reduce the signal-to-noise ratio in images.

1.1. Learning outcomes

- To know how to use several intensity mapping filters.
- To practice some basic neighborhood filters.
- To understand the basic mathematics about smoothing filters, how to use them for noise reduction, and what kind of images are each of them useful for.
- To know how edge preserving smoothing filters might reduce noise but maintaining image edges.

2. Casting and intensity mapping

Type conversion (casting) is a primary task for the ITK user. We need to anticipate the type we will use by taking into account the data file and the filters we will apply to, specifically the type for the pixels of the input image. The CastImageFilter (See section 'Casting and Intensity Mapping' on the *guide1* for further details) simply casts the type of each pixel of the input image to the pixel type of the output image. This filter cannot be considered as spatial filtering because it does not perform any operation on the intensity of the pixel, but it can change the intensity of the pixels due to truncation and round errors. See the code below and try with the image `example.png`.

```
typedef itk::Image<float,2>  FloatImageType;
typedef itk::Image<unsigned char,2>  UnsignedCharImageType;

// Declare an image file reader of type float and create an instance of it
typedef itk::ImageFileReader<FloatImageType> ReaderType;
ReaderType::Pointer reader = ReaderType::New();

// Declare a cast filter with input image of type float and output image of
// type unsigned char, and create an instance of it
```

[1] From now on we will be referring the ITK guide as *guide1* and *guide2* for `InsightSoftwareGuide-Book1.pdf` and `InsightSoftwareGuide-Book2.pdf`, respectively.

```
typedef itk::CastImageFilter<FloatImageType,UnsignedCharImageType> CastFilterType;
CastFilterType::Pointer castFilter = CastFilterType::New();
```

Listing 1: Casting of an image which has been read using a `float` type to a `unsigned char` type.

Figure 1 shows the file `example.png` taken from itk.org website. The image 1(a) shows the image read as a float image and the image 1(b) shows the image resulting from a cast to `unsigned char` type.

(a) Image stored using a float type. (b) Image stored using an unsigned char type.

Figure 1: Cast filter example.

The `RescaleIntensityImageFilter` applies a linear transformation to each pixel of the input image. The minimum and maximum values automatically extracted from the input image by the filter are scaled using a minimum and maximun values provided by the user. The linear transformation is as follows:

$$outPixel = (inPixel - inMIN) \cdot \frac{outMAX - outMIN}{inMAX - inMIN} + outMIN$$

The `ShiftScaleImageFilter` also applies a linear transformation to the intensity of each pixel of the input image, but the transformation is specified by the user using two parameters: a multiplying factor and a value to be added (shift). This can be expressed as

$$outPixel = (inPixel + shift) \cdot scale$$

The parameters of the linear transformation applied by the `NormalizeImageFilter` are computed internally such that the statistical distribution of gray levels in the output image have zero mean and a variance of one. This intensity correction is particularly useful in registration applications as a preprocessing step to the evaluation of mutual information metrics. The linear transformation applied by `NormalizeImageFilter` is given as

$$outPixel = \frac{inPixel - mean}{\sqrt{variance}}$$

Task 1

Write a program that reads an image from a file and displays the result of applying the following filters to the input image: `RescaleIntensityImageFilter`, `ShiftScaleImageFilter`, and `NormalizeImageFilter`. Try this program with the input image `BrainProtonDensitySlice256x256.png`. You may read the corresponding source code of each filter in '`ITK_DIR/Examples/Filtering`' to get a clue about how to program this task.

3. Neighborhood filters: Mean and Median filters

As we have seen in theory classes, the concept of neighborhood is frequently used in spatial filtering for computing every output pixel value using the pixel values in a neighborhood of the current input pixel. We define an operator over a neighborhood that computes an output value for the currently visited pixel based upon the values of the pixels in the neighborhood.

The *mean filter* (`MeanImageFilter` in ITK[2]) is commonly used for *noise reduction*. The filter computes the statistical mean of the neigborhood of the current pixel and assigns this value to the output pixel. The mean filter is sensitive to the presence of outliers in the neigborhood and ITK allows the user to set the size of the neighborhood. The size of the neighborhood is defined along every dimension by passing a `SizeType` object with the corresponding values. The value in each dimension indicates how many pixel wide we will take from the current pixel to set the neighborhood. For example, if we set $size[0] = 1, size[1] = 1$ in a 2D image we would be specifying a 3×3 neighborhood of pixels, and if we set $size[0] = 2, size[1] = 2$ we would be specifying a 5×5 neighborhood, i.e. a 5×5 pixel mask.

The *median filter* is a most robust approach for noise reduction than mean filter. In fact, this filter is particularly useful when working with *salt-and-pepper* noise because is not as sensitive as the mean filter to the presence of outliers in the neighborhood. The median filter computes the value of each output pixel as the statistical median of the neighborhood of the current input pixel. As in the case of mean filter, the user can set the size of the neighborhood using a SizeType object.

Task 2

Write a program that reads an image from a file and displays the result of applying two mean filters with neighborhood size of 3×3 and 5×5, respectively; and two median filters with the same neighborhood sizes. Try the program with the following images: `saltAndPepperNoise1.jpg`, `saltAndPepperNoise2.jpg`, and `gaussianNoise.jpg`. You may read the source code `MeanImageFilter.cxx` and `MedianImageFilter.cxx` placed at '`ITK_DIR/Examples/Filtering`' to understand the use of both filters.

4. Smoothing filters

Previous section has shown that salt and pepper noise can be effectively removed by applying the median filter, but we have seen that real noise is not so easy to remove (cf. task 2). The signal to noise ratio that appears in the different medical imaging modalities is a serious trouble in order to process these images. Next, we are going to describe several methods to reduce noise on images (For further reading see section 'Smoothing Filters' in the *guide2*).

4.1. Blurring

The effect of blurring an image is to attenuate high spatial frequencies and in these frequencies is precisely where noise goes on. Different filters attenuate frequencies in different ways but one of

[2] You can see the filters in the section 'Neighborhood Filters' of the *guide1*.

the most commonly used filters is the Gaussian. The ITK toolkit provides two implementations of Gaussian smoothing. Remember from theory classes that the gaussian filter is represented by the following equation:

$$G(x, \sigma, \mu) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{\frac{x-\mu}{\sigma^2}}$$

where μ represents the centroid of the function and σ represents the standard deviation[3] and thus the width of the function, i.e. the *width of the kernel*.

Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we can perform the convolution. This is done by taking advantage of the separability of the Gaussian kernel, therefore a one-dimensional Gaussian function is discretized on a convolution kernel. Finally, the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then convolving with another 1-D Gaussian in the y direction.

- The **Discrete Gaussian** filter computes the convolution of the input image with a Gaussian kernel. The size of the kernel is extended until there are enough discrete points in the Gaussian to ensure that a user-provided maximum error is not exceeded. Since the size of the kernel is unknown a priori, it is necessary to impose a limit to its growth. The user can thus provide a value to be the maximum admissible size of the kernel. The filter requires the user to provide a value for the variance associated with the Gaussian kernel. The method `SetVariance()` is used for this purpose. For details about how to use this filter read the code in `DiscreteGaussianImageFilter.cxx`.
- The **Binomial Blurring** filter computes a nearest neighbor average along each dimension. The process is repeated a number of times, as specified by the user. In principle, after a large number of iterations the result will approach the convolution with a Gaussian. The number of repetitions is set with the `SetRepetitions()` method and, as usual in a linear iterative method, the computation time will increase linearly with the number of repetitions selected. For details about how to use this filter read the code in `BinomialBlurImageFilter.cxx`.
- The **Recursive Gaussian IRR**[4] filter solves the problem related to the width of the discretized gaussian kernel when the standard deviation is large. This is due to the large amount of computation per pixel because of the neighborhood taken into account. This filter implements an approximation of convolution with the Gaussian and its derivatives by using *IIR filters*. In practice this filter requires a constant number of operations for approximating the convolution, regardless of the σ value. For details about how to use this filter read the code in `SmoothingRecursiveGaussianImageFilter.cxx` and the ITK *guide2*, section on Smoothing Filters, concretely Blurring.

Task 3

Write a program that reads an image from a file and displays the result of applying the three Gaussian filters presented above. Try the program with the following images: `saltAndPepperNoise1.jpg`, `saltAndPepperNoise2.jpg`, and `gaussianNoise.jpg`. Compare the resulting images with the images from **Task 2**.

5. Edge Preserving Smoothing

The problem of noise reduction filters presented so far is that they are not adaptive. Features such as edges are not preserved, indeed they are smoothed. These filters cannot adapt neither the size of

[3] In statistics, when we consider the Gaussian probability density function it is called the standard deviation σ, and the square of it, σ^2, the variance.

[4] Infinite Impulse Response

the filter nor its elements' values to local characteristics of images. Perona and Malik introduced an alternative to linear-filtering that they called *anisotropic diffusion*.

Anisotropic diffusion formulation includes a variable conductance term that, in turn, depends on the differential structure of the image. Thus, the variable conductance can be formulated to limit the smoothing at "edges" in images, as measured by high gradient magnitude. ITK implements two variants of anisotropic diffusion:

- **Gradient Anisotropic Diffusion** filter implements an N-dimensional version of the classic Perona-Malik anisotropic diffusion equation for scalar-valued images. The conductance term for this implementation is chosen as a function of the gradient magnitude of the image at each point, reducing the strength of diffusion at edge pixels. This filter requires three parameters: the number of iterations to be performed, the time step and the conductance parameter used in the computation of the level set evolution. These parameters are set using the methods `SetNumberOfIterations()`, `SetTimeStep()` and `SetConductanceParameter()` respectively. Typical values for the time step are 0.25 in 2D images and 0.125 in 3D images. The number of iterations is typically set to 5; more iterations result in further smoothing and will increase the computing time linearly. For details about how to use this filter read the code in `GradientAnisotropicDiffusionImageFilter.cxx`.
- **Curvature Anisotropic Difusion** filter performs anisotropic diffusion on an image using a modified curvature diffusion equation (MCDE). MCDE does not exhibit the edge enhancing properties of classic anisotropic diffusion, which can under certain conditions undergo a "negative" diffusion, which enhances the contrast of edges. Qualitatively, MCDE is less sensitive to contrast than classic Perona-Malik style diffusion, and preserves finer detailed structures in images. This filter requires three parameters: the number of iterations to be performed, the time step used in the computation of the level set evolution and the value of conductance. These parameters are set using the methods `SetNumberOfIterations()`, `SetTimeStep()` and `SetConductance()` respectively. Typical values for the time step are 0.125 in 2D images and 0.0625 in 3D images. The number of iterations can be usually around 5, more iterations will result in further smoothing and will increase the computing time linearly. The conductance parameter is usually around 3.0. For details about how to use this filter read the code in `CurvatureAnisotropicDiffusionImageFilter.cxx`.
- **Curvature Flow** filter performs edge-preserving smoothing in a similar fashion to the classical anisotropic diffusion. The filter uses a level set formulation where the iso-intensity contours in an image are viewed as level sets, where pixels of a particular intensity form one level set. The level set function is then evolved under the control of a diffusion equation. Areas of high curvature will diffuse faster than areas of low curvature. Hence, small jagged noise artifacts will disappear quickly, while large scale interfaces will be slow to evolve, thereby preserving sharp boundaries between objects. The CurvatureFlow filter requires two parameters: the number of iterations to be performed and the time step used in the computation of the level set evolution. These two parameters are set using the methods `SetNumberOfIterations()` and `SetTimeStep()` respectively. Typical values for the time step are 0.125 in 2D images and 0.0625 in 3D images. The number of iterations can be usually around 10, more iterations will result in further smoothing and will increase the computing time linearly. Edge-preserving behavior is not guaranteed by this filter. Some degradation will occur on the edges and will increase as the number of iterations is increased. For details about how to use this filter read the code in `CurvatureFlowImageFilter.cxx`.

Task 4

Write a program that reads an image from a file and displays the result of applying the three edge-preserving filters presented above. Try the program with the following images: `saltAndPepperNoise1.jpg`, `saltAndPepperNoise2.jpg`, and `gaussianNoise.jpg`. Compare the resulting images with the images from tasks **Task 2** y **Task 3**.

Exercise

Take as input a noisy image and try to obtain a noise-free output image by combining several filters in a pipeline. Write a report on the different results you have obtained concerning the different types of images you has provided to your programs including smoothing filters and edge preserve smoothing filters, together with the different parameter values you have used. Please, illustrate the results showing the images you have obtained in each pipeline, detailing the parameters you have tried out.

References

[1] Johnson, H. J., McCormick, M. M., and Ibanez, L. *The ITK Software Guide Book 1: Introduction and Development Guidelines-Volume 1 4th ed.*, 2017.

[2] Johnson, H. J., McCormick, M. M., and Ibanez, L. *The ITK Software Guide Book 2: Design and Functionality-Volume 2 4th ed.*, 2017.