Course: Software Engineering (Project).
Project: Implement Gaussian and Mean Filters for Noise Removal

Supervision

Prof. Dr. Pech
&
Prof. Damir Dobric

Author

Md. Nazim Uddin
Matriculation Number: 1100022
Date: 29-05-2017

Contents

Abstract

In this paper, a basic but efficient algorithm has been used that capable to remove image noise. This complete noise removal procedure depends on Gaussian and Mean filter algorithm, that has less computational complexity. A novel algorithm for Gaussian noise estimation and removal is proposed by using 3x3, 5x5, 7x7 and 9x9 sub windows in which the test pixel appears. The standard deviation(STD) for all sub-windows are used to define reference STD(**σ**ref)and minimum(**σ**min) and maximum (**σ**max) standard deviations. The algorithm initially estimates the amount of noise corruption from the noise corrupted image. In the second stage, the center pixel is replaced by the mean value of the some of the surrounding pixels based on a threshold value. Noise removing with edge preservation and computational complexity are two conflicting parameters. Experimental results show the superior performance of Gaussian and Mean filter algorithm in .NET Core. This method removes Gaussian noise and the edges are better preserved with less computational complexity and this aspect makes it easy to implement in hardware.

Introduction

Noise having Gaussian-like distribution is very often encountered in acquired data. Gaussian noise is characterized by adding to each image pixel a value from a zero-mean Gaussian distribution. The zero-mean property of the distribution allows such noise to be removed by locally averaging pixel values. Conventional linear filters such as arithmetic mean filter and Gaussian filter smooth noises effectively but blur edges. The Gaussian smoothing operator is a 2-D convolution operator that is used to `blur' images and remove detail and noise. In this sense, it is like the mean filter, but it uses a different kernel that represents the shape of a Gaussian (`bell-shaped') hump. This kernel has some special properties. Since the goal of the filtering action is to cancel noise while preserving the integrity of edge and detail information, nonlinear approaches generally provide more satisfactory results than linear techniques. This information is necessary to perform the optimal choice of parameter values and/or threshold selections. Unfortunately, such information is very often not available in real time applications.

In the field of image processing, there have been many attempts to construct digital filters which have the qualities of noise attenuation and detail preservation. The Gaussian noise filtering technique it not a sufficient method to noise remove but a successful basic filtering method. Removing Gaussian noise involves smoothing the inside distinct region of an image. For this classical linear filter such as the Gaussian filter reduces noise efficiently but blur the edges significantly. Several researchers have attempted to generalize others standard filters those are seldom suitable for removing Gaussian noise. A nonlinear diffusion equations called as an anisotropic diffusion algorithm have been proposed for Gaussian noise removal.

Fundamental theory of Gaussian Filter

Let 'X' is an original image, 'A' is observed image, and a general discrete time model for image degradation can be expressed as

$A_{i,j} = X_{i,j} + \eta_{i,j}$

For i,j = 1,2….N, where Xi,j is original image pixels, ηi,j is additive Gaussian noise and Ai,j is the observed image. The objective of the restoration scheme is to recover the original image from the observed image. Here it is assumed the noise is normally distributed for a given mean and variance.

Let the noisy image is represented with A. the test pixel is located at (i,j), generally the 3 x 3 neighborhood is considered for normal filtering, whether corrupted or not. In our method we examined the 5 x 5 neighborhood of the test pixel in a different way. The 5 x 5 neighborhood is divided in to nine 3 x 3 sub-windows such that the test pixel appears in each of the sub-window. For each sub-window standard deviation (σ) [11, 12] is calculated. A reference standard deviation is decided as the median of the above sub-windows standard deviation (σi), i=1, 2 … 9, and two thresholds σmax and σmin are set and then average of the standard deviation (σavg) of nine sub windows whose standard deviation fall in the range [σmin, σmax] is calculated. This average standard deviation is now used to estimate whether the pixel under test is corrupted or not. This is done based on the difference between mean of the 3x3 neighborhood of the test pixel (i, j) and the pixel value itself. The test pixel is decided corrupted based on the above difference lies within the range [a, b], otherwise treated as uncorrupted. Where the range limits 'a' and 'b' are experimentally obtained as 0.5 x σavg and 0.5, respectively. This is repeated for the entire noisy image. The detailed procedure is explained in the section 3.

The Gaussian distribution in 1-D has the form

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

where **σ** is the standard deviation of the distribution. We have also assumed that the distribution has a mean of zero (*i.e.* it is centered on the line *x*=0). The distribution is illustrated in Figure

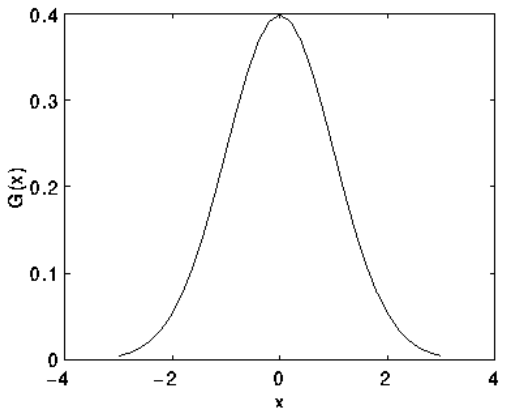


Fig: 1-D Gaussian distribution with mean 0 and **σ** =1

In 2-D, an isotropic (*i.e.* circularly symmetric) Gaussian has the form

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

This distribution is shown in Figure below

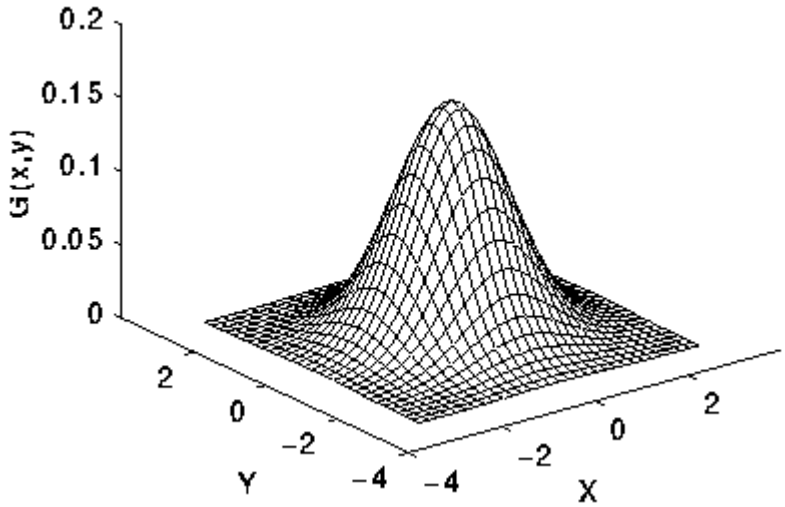


Fig: 2-D Gaussian distribution with mean (0,0) and **σ** =1

The idea of Gaussian smoothing is to use this 2-D distribution as a `point-spread' function, and this is achieved by convolution. 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. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point. Figure 3 shows a suitable integer-valued convolution kernel that approximates a Gaussian with a **σ** of 1.0. It is not obvious how to pick the values of the mask to approximate a Gaussian. One could use the value of the Gaussian at the center of a pixel in the mask, but this is not accurate because the value of the Gaussian varies non-linearly across the pixel. We integrated the value of the Gaussian over the whole pixel (by summing the Gaussian at 0.001 increments). The integrals are not integers: we rescaled the array so that the corners had the value 1. Finally, the 273 is the sum of all the values in the mask.

Remove Gaussian noise

If the pixel is found corrupted then a filter is invoked. The corrupted pixel is replaced with a new value obtained from the following formula.

$x_{new}(i, j)= [\mu\text{-}0.5x\ \sigma avg\ ]$

Where x_{new} is the new value for the pixel position represented by (i, j), μ is the mean of the 3x3 central sub window, σavg is the average standard deviation defined in the previous section.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Fig: Discrete approximation to Gaussian function with **σ** =1.0

Once a suitable kernel has been calculated, then the Gaussian smoothing can be performed using standard convolution methods. The convolution can in fact be performed quickly since the equation for the 2-D isotropic Gaussian shown above is separable into *x* and *y* components. Thus 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 Gaussian is in fact the *only* completely circularly symmetric operator which can be decomposed in such a way.)

Figure 4 shows the 1-D *x* component kernel that would be used to produce the full kernel shown in Figure 3 (after scaling by 273, rounding and truncating one row of pixels around the boundary because they mostly have the value 0. This reduces the 7x7 matrix to the 5x5 shown above.). The *y* component is the same but is oriented vertically.

A further way to compute a Gaussian smoothing with a large standard deviation is to convolve an image several times with a smaller Gaussian. While this is computationally complex, it can have applicability if the processing is carried out using a hardware pipeline.

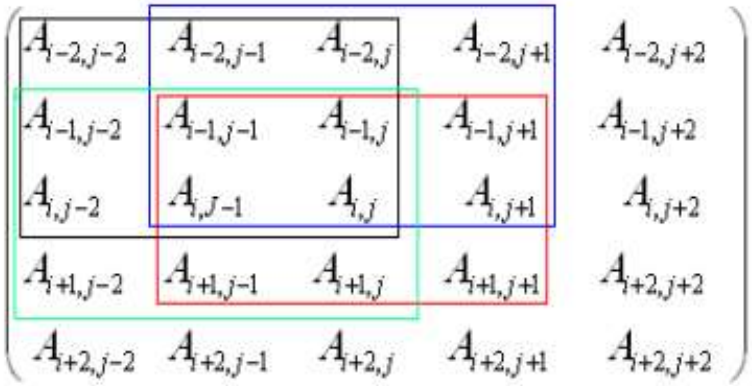
The Gaussian filter not only has utility in engineering applications. It is also attracting attention from computational biologists because it has been attributed with some amount of biological plausibility, *e.g.*some cells in the visual pathways of the brain often have an approximately Gaussian response.

Algorithm

1. Consider a 5 x 5 test window AT from the noisy image as:

$$A_T = \begin{pmatrix} A_{i-2,j-2} & A_{i-2,j-1} & A_{i-2,j} & A_{i-2,j+1} & A_{i-2,j+2} \\ A_{i-1,j-2} & A_{i-1,j-1} & A_{i-1,j} & A_{i-1,j+1} & A_{i-1,j+2} \\ A_{i,j-2} & A_{i,J-1} & A_{i,j} & A_{i,j+1} & A_{i,j+2} \\ A_{i+1,j-2} & A_{i+1,j-1} & A_{i+1,j} & A_{i+1,j+1} & A_{i+2,j+2} \\ A_{i+2,j-2} & A_{i+2,j-1} & A_{i+2,j} & A_{i+2,j+1} & A_{i+2,j+2} \end{pmatrix}$$

2. Divide this window into 3 x 3 sub-windows such that the test pixel should appear in each of the sub-windows. Nine such sub-windows are possible and four of them as shown below.



3. For each 3x3 sub-window calculate the standard deviation, σi, i=1, 2 ……N where N is maximum number of the sub-windows, for this paper it is equal to 9.
4. Set reference standard deviation, (σref), as median of σi, i=1, 2 ……N.
5. Set σmin = k1x σref.
6. Set σmax = k2 x σref.
7. Calculate average (σavg) of the standard deviations σi, i=1, 2 ……N whose standard deviation lies in the range [σmin, σmax].
8. This σavg is used as a parameter to decide whether the test pixel is corrupted or not.

The above process is repeated by sliding 5x5 window one step forward row wise and then column wise to cover the entire image.

Both filters attenuate high frequencies more than low frequencies, but the mean filter exhibits oscillations in its frequency response. The Gaussian on the other hand shows no oscillations. In fact, the shape of the frequency response curve is itself (half a) Gaussian. So, by choosing an appropriately sized Gaussian filter we can be confident about what range of spatial frequencies are still present in the image after filtering, which is not the case of the mean filter. This has consequences for some edge detection techniques, as mentioned in the section on zero crossings. (The Gaussian filter also turns out to be very like the optimal smoothing filter for edge detection under the criteria used to derive the Canny edge detector.

Following image has been used to demonstrate Gaussian filtering process.

which has been corrupted by 1% salt and pepper noise (*i.e.* individual bits have been flipped with probability 1%). The image became

shows the result of Gaussian smoothing (using the same convolution as above). Compare this with the original

Notice that much of the noise still exists and that, although it has decreased in magnitude somewhat, it has been smeared out over a larger spatial region. Increasing the standard deviation continues to reduce/blur the intensity of the noise, but also attenuates high frequency detail (*e.g.* edges) significantly, as shown in pictures.

Fundamental theory of Mean Filter

The mean filter is a simple sliding-window spatial filter that replaces the center value in the window with the average (mean) of all the pixel values in the window. The window, or kernel, is usually square but can be any shape. An example of mean filtering of a single 3x3 window of values is shown below.

Unfiltered values

5	3	6
2	1	9
8	4	7

5 + 3 + 6 + 2 + 1 + 9 + 8 + 4 + 7 = 45
45 / 9 = 5

Mean filtered

*	*	*
*	5	*
*	*	*

The idea of mean filtering is simply to replace each pixel value in an image with the mean ('average') value of its neighbors, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is usually thought of as a convolution filter. Like other convolutions it is based around a kernel, which represents the shape and size of the neighborhood to be sampled when calculating the mean. Often a 3×3 square kernel is used, as shown in Figure 1, although larger kernels (*e.g.* 5×5 squares) can be used for more severe smoothing. (Note that a small kernel can be applied more than once to produce a similar but not identical effect as a single pass with a large kernel.)

Image smoothing refers to any image-to-image transformation designed to smoothen or flatten an image by reducing the rapid pixel-to-pixel variation in grey levels. Smoothing may be

accomplished by applying an averaging mask that computes a weighted sum of the pixel grey levels in a neighborhood and replaces the center pixel with that grey level. The image is blurred and its brightness retained as the mask coefficients are all-positive and sum to one. The mean filter is one of the most basic smoothing filters. Mean filtering is usually thought of as a convolution operation as the mask is successively moved across the image until every pixel has been covered. Like other convolutions it is based around a kernel, which represents the shape

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \qquad \frac{1}{n^2}\begin{bmatrix}1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\\ \vdots & \vdots & \ddots & \vdots\\ 1 & 1 & \cdots & 1\end{bmatrix}$$

(a) (b)

Fig: (a) 3 × 3 mean filter; (b) general n × n mean filter

Fig: Effect of mean filtering: (a) original crowd image; (b) Blurred image with n = 7 and (c) 11.

and size of the neighborhood to be sampled when calculating the mean. Larger kernels are used when more severe smoothing is required. Fig. (a) shows a mean mask for a 3 × 3 window, while a more general n × n mask is shown in Fig. (b). Variations on the mean filter include threshold averaging, wherein smoothing is applied subject to the condition that the center pixel grey level is changed only if the difference between its original value and the average value is greater than a preset threshold. This causes the noise to be smoothed with a less blurring in image detail. It must be noted here that the smoothing operation is equivalent to low-pass filtering as it eliminates edges and regions of sudden grey level change by replacing the center pixel grey level by the neighborhood average. It effectively eliminates pixel grey levels that are unrepresentative of their surroundings. Noise, due to its spatial decorrelations, generally has a higher spatial frequency spectrum than the normal image components. Hence, a simple low-pass filter can be very effective in noise cleaning. Smoothing filters thus find extensive use in blurring and noise

removal. Blurring is usually a preprocessing step bridging gaps in lines or curves, helping remove small unwanted detail before the extraction of relevant larger objects. Figs. (b) and (c) show the effect of averaging for various window sizes n (7 and 11). The mean filter is an important image processing tool and finds use in Gaussian noise reduction, blurring before thresholding to eliminate small detail, bridging gaps in broken characters for improved machine perception in OCRs, cosmetic processing of human faces images to reduce fine skin lines and blemishes, etc. The mean is also used as a derived or texture feature in image segmentation process. Let us now look at the basic implementation of this mean filter.

The basic method

Consider an L level image F (P × Q), whose pixel grey levels are stored in a 2-D array, say, data[P][Q] such that data [0][0] and data [P − 1][Q − 1] contains the first and last pixels, respectively [2]. To apply the mean-filter to this image over a rectangular neighborhood window m × n (m & n are odd positive integers), the image must be appropriately padded by some method like replication of boundary rows and columns [1] at the four sides of the input image. The average grey level of a pixel data[row][col] over an m × n neighborhood is then calculated as follows,

```
For row = 0 to P - 1

     For col = 0 to Q - 1

          Sum = Sum of m × n pixel grey levels in the
          neighborhood centered at data[row][col] new
          data[row][col] = Sum/ (m × n)

     End

End
```

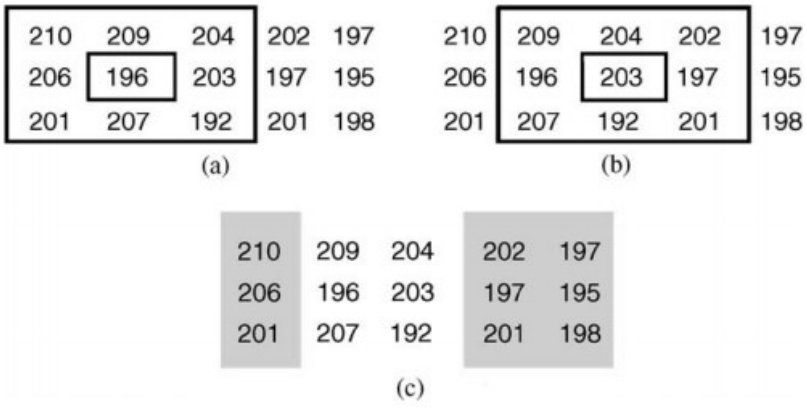


Fig: (a) Neighborhood around 196; (b) neighborhood around 203; (c) un-shaded portion represents 6 out of 9 common pixels.

Here new data[P][Q] is a 2-D array containing the computed averages. This basic method, though simple to implement is inefficient due to unnecessary re-computations in terms of both additions and divisions. The required number of additions is P×Q×m×n, while that of divisions is P×Q. Thus, both are in direct relation to the image size and increases proportionately with it. A major redundancy stems from the fact that the sum over a neighborhood is repeatedly computed as the mask is shifted from one pixel to the next. As the mask is shifted by only one pixel at a time, we need only add that new pixels and subtract the old ones instead of computing the whole sum all over again . Consider the 3 × 3 neighborhood window around the pixel 196 as shown in Fig. (a). When this neighborhood is shifted right to the next pixel, 203 we get Fig.(b). The pixels common to both these neighborhoods are shown un-shaded in Fig. (c) and their sum can be carried forward to the next neighborhood without the need for re-computation. The algorithm for a fore mentioned addition reduction procedure is summarized below

```
For row = 0 to P – 1

     col = 0

     Sum = Sum of m × n pixel grey levels in the neighborhood
     centered at data[row][col]

     new data [row][col] = Sum/(m × n)

     For col = 1 to Q – 1

          Sum = Sum – m number of pixel grey levels in column
          (col – n/2)

          Sum = Sum + m number of pixel grey levels in column
          (col + n/2) [n/2 is the integer part of n by 2] new
          data [row][col] = Sum/ (m × n)

     End

End
```

At the beginning of every new row, the sum of m × n pixel grey levels are computed. This total neighborhood pixel grey level summation is carried out P times as against the P × Q times in the normal case. The required number of additions is now reduced to P × m × n + P × Q × m × 2, which is almost equal to P × Q × m × 2. The proposed column-summation store-and-fetch method [9,10] will reduce this number drastically.

Division reduction

Division, being a floating-point operation, contributes most significantly towards the computation time in any filtering algorithm. Thus, it is of primary importance to try and reduce their number. Let us consider the innate redundancy in the basic method about divisions, by considering a 3 × 3 neighborhood-averaging example. The possible range of summation of 9 grey levels, (considering an 8-bit grey level image), is from 0 to 2295 (255 × 9) as shown in Fig. 5. It may be noted here that in case of 7-bit images we only need to store the averages from 0 to 1143 (127 × 9). Thus, if all pixels are black, the summation is 0, while if all pixels are white the summation is 2295, with each sum having a corresponding average as shown in Fig. 6. For example, when the sum of 9 pixel grey levels is between 0 and 8, the average is 0 (considering the floor function for simplicity). Similarly, for a sum in between 2286 and 2294, the average is 254. So, we need only store these averages in an array.

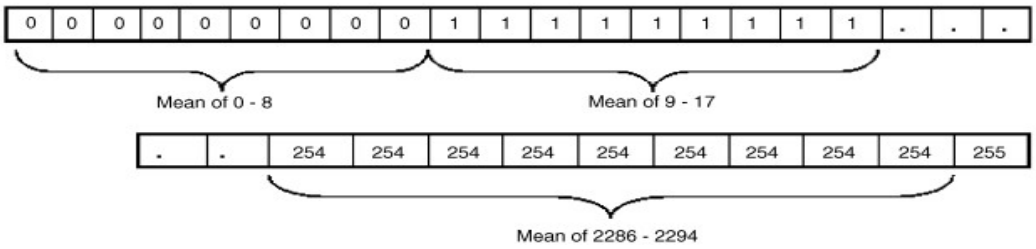


Fig: Storing averages in an array with the sum as index.

(say mean []) of length 2296, (255 × m × n + 1), and access them using the sum of neighborhood pixel grey levels as the array index. The mean storing algorithm is as follows,

```
j = k = 0
For i = 0 to 255 × m × n
     If k = m × n
          k = 0
          j = j + 1
```

```
    End
     mean[i] = j
     k = k + 1
End
```

In this case the number of divisions is reduced to 255 × m × n + 1. This number may be reduced to zero by using multiple additions, as they are far less computationally intensive. The final algorithm is as follows

```
For row = 0 to (P – 1)
    col = 0 Store sum of pixel grey levels of columns 0,
    1,...,n – 1 in s[0], s[1],...,s[n – 1]
    Sum = s[0] + s[1]+···+ s[n – 1]
    new data[row][col] = mean[Sum]
    j = 0
    For col = 1 to (Q – 1)
        Sum = Sum – s[j ]
        s[j ] = Sum of pixel grey levels in column (col + n/2)
        Sum = Sum + s[j ]
        new data[row][col] = mean[Sum]
        j = j + 1
        If j > = n
        j = 0
        End
    End
End
```

It is identical to the one in Section 4.1, except for the lines marked bold, where the method of mean storage has been changed to one of direct-access. Going back to our original example, a 1024 × 1024 image with a neighborhood of 7 × 7 now requires around 9 × 106 additions and 0 divisions instead of the original 50 × 106 additions and 1 × 106 divisions.

Algorithm Implementation

C# Code

Main code

```
using GaussianAndMeanFilter;
using LearningFoundation;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Xunit;

namespace UnitTest
{
    public class UnitTest
    {
        /// <summary>
        /// Unit test for Mean Filter
        /// </summary>
        [Fact]
        public void MeanF()
        {
            LearningApi lApi = new LearningApi();
            lApi.UseActionModule<double[,,], double[,,]>((input, ctx) =>
            {
                Bitmap myBitmap = new
Bitmap($"{Directory.GetCurrentDirectory()}/TestPicture/fce5noi3.gif");

                double[,,] data = new double[myBitmap.Width, myBitmap.Height, 3];

                for (int x = 0; x < myBitmap.Width; x++)
                {
                    for (int y = 0; y < myBitmap.Height; y++)
                    {
                        Color pixelColor = myBitmap.GetPixel(x, y);

                        data[x, y, 0] = pixelColor.R;
                        data[x, y, 1] = pixelColor.G;
                        data[x, y, 2] = pixelColor.B;
                    }
                }
                return data;
            });

            lApi.AddModule(new MeanFilter());

            double[,,] result = lApi.Run() as double[,,];
```

```
            Assert.True(result != null);

            Bitmap blurBitmap = new Bitmap(result.GetLength(0), result.GetLength(1));

            for (int x = 0; x < result.GetLength(0); x++)
            {
                for (int y = 0; y < result.GetLength(1); y++)
                {
                    Color pixelColor = Color.FromArgb((int)result[x, y, 0],
(int)result[x, y, 1], (int)result[x, y, 2]);

                    blurBitmap.SetPixel(x, y, pixelColor);
                }
            }

            Image img = (Image)blurBitmap;

            img.Save($"{ Directory.GetCurrentDirectory()}/OutPutPic/Mean.jpg");
        }

        /// <summary>
        /// Unit test for Gaussian Filter
        /// </summary>
        [Fact]
        public void Gaussian()
        {
            LearningApi lApi = new LearningApi();
            lApi.UseActionModule<double[,,], double[,,]>((input, ctx) =>
            {
                Bitmap myBitmap = new
Bitmap($"{Directory.GetCurrentDirectory()}/TestPicture/fce5noi3.gif");

                double[,,] data = new double[myBitmap.Width, myBitmap.Height, 3];

                for (int x = 0; x < myBitmap.Width; x++)
                {
                    for (int y = 0; y < myBitmap.Height; y++)
                    {
                        Color pixelColor = myBitmap.GetPixel(x, y);

                        data[x, y, 0] = pixelColor.R;
                        data[x, y, 1] = pixelColor.G;
                        data[x, y, 2] = pixelColor.B;
                    }
                }
                return data;
            });

            lApi.AddModule(new GaussianFilter());

            double[,,] result = lApi.Run() as double[,,];

            Assert.True(result != null);

            Bitmap blurBitmap = new Bitmap(result.GetLength(0), result.GetLength(1));

            for (int x = 0; x < result.GetLength(0); x++)
```

```
            {
                for (int y = 0; y < result.GetLength(1); y++)
                {
                    Color pixelColor = Color.FromArgb((int)result[x, y, 0],
(int)result[x, y, 1], (int)result[x, y, 2]);

                    blurBitmap.SetPixel(x, y, pixelColor);
                }
            }

            Image img = (Image)blurBitmap;

            img.Save($"{ Directory.GetCurrentDirectory()}/OutPutPic/Gaussian.jpg");
        }

        /// <summary>
        /// Unit test for Gaussian and Mean Filter
        /// </summary>
        [Fact]
        public void GaussianAndMean()
        {
            LearningApi lApi = new LearningApi();
            lApi.UseActionModule<double[,,], double[,,]>((input, ctx) =>
            {
                Bitmap myBitmap = new
Bitmap($"{Directory.GetCurrentDirectory()}/TestPicture/fce5noi3.gif");

                double[,,] data = new double[myBitmap.Width, myBitmap.Height, 3];

                for (int x = 0; x < myBitmap.Width; x++)
                {
                    for (int y = 0; y < myBitmap.Height; y++)
                    {
                        Color pixelColor = myBitmap.GetPixel(x, y);

                        data[x, y, 0] = pixelColor.R;
                        data[x, y, 1] = pixelColor.G;
                        data[x, y, 2] = pixelColor.B;
                    }
                }
                return data;
            });

            lApi.AddModule(new GaussianFilter());

            lApi.AddModule(new MeanFilter());

            double[,,] result = lApi.Run() as double[,,];

            Assert.True(result != null);

            Bitmap blurBitmap = new Bitmap(result.GetLength(0), result.GetLength(1));

            for (int x = 0; x < result.GetLength(0); x++)
            {
                for (int y = 0; y < result.GetLength(1); y++)
                {
```

```
                    Color pixelColor = Color.FromArgb((int)result[x, y, 0],
(int)result[x, y, 1], (int)result[x, y, 2]);

                    blurBitmap.SetPixel(x, y, pixelColor);
                }
            }

            Image img = (Image)blurBitmap;

            img.Save($"{
Directory.GetCurrentDirectory()}/OutPutPic/GaussianAndMean.jpg");
        }
    }
}
```

Mean Filter Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using LearningFoundation;

namespace GaussianAndMeanFilter
{
    public class MeanFilter : IPipelineModule<double[,,], double[,,]>
    {
        /// <summary>
        /// Inherited from IPipeLine module
        /// </summary>
        /// <param name="data"></param>
        /// <param name="ctx"></param>
        /// <returns></returns>
        public double[,,] Run(double[,,] data, IContext ctx)
        {
            return filter(data);
        }

        /// <summary>
        /// Implementing a Mean filter matrix of (3x3) dimention
        /// </summary>
        /// <param name="data"></param>
        /// <returns></returns>
        private double[,,] filter(double[,,] data)
        {
            double[,,] result = data;

            for (int x = 1; x < data.GetLength(0)-1; x++)
            {
                for (int y = 1; y < data.GetLength(1)-1; y++)
                {
                    //Reading Red value first row of the 3x3 matrix
                    var prev11R = data[x - 1, y - 1, 0];
                    var prev12R = data[x , y - 1, 0];
```

```
                    var prev13R = data[x + 1, y - 1, 0];

                    //Reading Red value sencond row of the 3x3 matrix
                    var prev21R = data[x - 1, y, 0];
                    var prev22R = data[x , y, 0];
                    var prev23R = data[x + 1, y, 0];

                    //Reading Red value third row of the 3x3 matrix
                    var prev31R = data[x - 1, y+1, 0];
                    var prev32R = data[x, y+1, 0];
                    var prev33R = data[x + 1, y+1, 0];

                    //Reading Green value first row of the 3x3 matrix
                    var prev11G = data[x - 1, y - 1, 1];
                    var prev12G = data[x, y - 1, 1];
                    var prev13G = data[x + 1, y - 1, 1];

                    //Reading Green value sencond row of the 3x3 matrix
                    var prev21G = data[x - 1, y, 1];
                    var prev22G = data[x, y, 1];
                    var prev23G = data[x + 1, y, 1];

                    //Reading Green value third row of the 3x3 matrix
                    var prev31G = data[x - 1, y + 1, 1];
                    var prev32G = data[x, y + 1, 1];
                    var prev33G = data[x + 1, y + 1, 1];

                    //Reading Blue value first row of the 3x3 matrix
                    var prev11B = data[x - 1, y - 1, 2];
                    var prev12B = data[x, y - 1, 2];
                    var prev13B = data[x + 1, y - 1, 2];

                    //Reading Blue value sencond row of the 3x3 matrix
                    var prev21B = data[x - 1, y, 2];
                    var prev22B = data[x, y, 2];
                    var prev23B = data[x + 1, y, 2];

                    //Reading Blue value third row of the 3x3 matrix
                    var prev31B = data[x - 1, y + 1, 2];
                    var prev32B = data[x, y + 1, 2];
                    var prev33B = data[x + 1, y + 1, 2];

                    //Calculating the mean value
                    double avgR = (prev11R + prev12R + prev13R + prev21R + prev22R +
prev23R + prev31R + prev32R + prev33R)/9;
                    double avgG = (prev11G + prev12G + prev13G + prev21G + prev22G +
prev23G + prev31G + prev32G + prev33G) / 9;
                    double avgB = (prev11B + prev12B + prev13B + prev21B + prev22B +
prev23B + prev31B + prev32B + prev33B) / 9;

                    //Replacing the original pixel value by the calculated value
                    result[x, y, 0] = avgR;
                    result[x, y, 1] = avgG;
                    result[x, y, 2] = avgB;
                }
            }

            return result;
```

```
        }
    }
}
```

Gaussian Filter Code

```
using LearningFoundation;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace GaussianAndMeanFilter
{
    public class GaussianFilter : IPipelineModule<double[,,], double[,,]>
    {
        /// <summary>
        ///
        /// </summary>
        /// <param name="data"></param>
        /// <param name="ctx"></param>
        /// <returns></returns>
        public double[,,] Run(double[,,] data, IContext ctx)
        {
            return filter(data);
        }

        /// <summary>
        /// Implementing a Gaussian filter matrix of (3x3) dimention
        /// </summary>
        /// <param name="data"></param>
        /// <returns></returns>
        private double[,,] filter(double[,,] data)
        {
            double[,,] result = data;

            for (int x = 1; x < data.GetLength(0) - 1; x++)
            {
                for (int y = 1; y < data.GetLength(1) - 1; y++)
                {
                    //Red value first row
                    var prev11R = data[x - 1, y - 1, 0];
                    var prev12R = data[x, y - 1, 0];
                    var prev13R = data[x + 1, y - 1, 0];

                    //Red value sencond row
                    var prev21R = data[x - 1, y, 0];
                    var prev22R = data[x, y, 0];
                    var prev23R = data[x + 1, y, 0];

                    //Red value third row
                    var prev31R = data[x - 1, y + 1, 0];
                    var prev32R = data[x, y + 1, 0];
```

```
                    var prev33R = data[x + 1, y + 1, 0];

                    //Green value first row
                    var prev11G = data[x - 1, y - 1, 1];
                    var prev12G = data[x, y - 1, 1];
                    var prev13G = data[x + 1, y - 1, 1];

                    //Green value sencond row
                    var prev21G = data[x - 1, y, 1];
                    var prev22G = data[x, y, 1];
                    var prev23G = data[x + 1, y, 1];

                    //Green value third row
                    var prev31G = data[x - 1, y + 1, 1];
                    var prev32G = data[x, y + 1, 1];
                    var prev33G = data[x + 1, y + 1, 1];

                    //Blue value first row
                    var prev11B = data[x - 1, y - 1, 2];
                    var prev12B = data[x, y - 1, 2];
                    var prev13B = data[x + 1, y - 1, 2];

                    //Blue value sencond row
                    var prev21B = data[x - 1, y, 2];
                    var prev22B = data[x, y, 2];
                    var prev23B = data[x + 1, y, 2];

                    //Blue value third row
                    var prev31B = data[x - 1, y + 1, 2];
                    var prev32B = data[x, y + 1, 2];
                    var prev33B = data[x + 1, y + 1, 2];

                    //Calculating new pixel value
                    double avgR = (prev11R*1 + prev12R*2 + prev13R*1 + prev21R*2 +
prev22R*4 + prev23R*2 + prev31R*1 + prev32R*2 + prev33R*1) / 16;
                    double avgG = (prev11G * 1 + prev12G * 2 + prev13G * 1 + prev21G * 2
+ prev22G * 4 + prev23G * 2 + prev31G * 1 + prev32G * 2 + prev33G * 1) / 16;
                    double avgB = (prev11B * 1 + prev12B * 2 + prev13B * 1 + prev21B * 2
+ prev22B * 4 + prev23B * 2 + prev31B * 1 + prev32B * 2 + prev33B * 1) / 16;

                    //Replacing the original pixel value by the calculated value
                    result[x, y, 0] = avgR;
                    result[x, y, 1] = avgG;
                    result[x, y, 2] = avgB;
                }
            }

            return result;
        }
    }
}
```

Class and Function used

UnitTest.cs:

In this class there are three user defined functions named `MeanF()`, `Gaussian()` and `GaussianAndMean()` are used to implement the mean filter, Gaussian filter and Gaussian and Mean Filter respectively. An input image is converted to bitmap by using ActionModule bitmap data is ported to algorithm. After filtering, algorithm returns a bitmap data which is converted to image and saved.

MeanFilter.cs :

In this class, based on the pixels of an input image matrix new pixel values are calculated with a 3x3 kernel of mean filter.

GaussianFilter.cs :

In this class, based on the pixels of an input image matrix new pixel values are calculated with a 3x3 kernel of Gaussian filter.

Conclusion

The fast Gaussian and Mean filtering technique presented here has successfully reduced the time requirements for smoothing operations with mean filters, especially for large images. This implementation reduces the number of additions to approximately 1/nth of the original number, where n×n is the neighborhood size and completely eliminates the division operation by store and-fetch methods very efficiently. The gain in the performance and usefulness of this method has been demonstrated for different images.

The various applications of Gaussian and Mean filtering, as described in detail at the beginning of this paper can all benefit tremendously from this improved implementation. Noise removal by threshold averaging in remote-sensing applications is one such important example. The effect of this filtering technique will be tremendous in such an application. This method can easily be extended to higher bit-level grayscale images or color images.

Reference

1. https://www.cs.auckland.ac.nz/courses/compsci373s1c/PatricesLectures/Gaussian%20Filtering_1up.pdf
2. https://www.markschulze.net/java/meanmed.html
3. http://www1.inf.tu-dresden.de/~ds24/lehre/bvme_ss_2013/ip_03_filter.pdf
4. http://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm
5. http://www.cns.nyu.edu/pub/eero/simoncelli96c.pdf
6. http://www.ripublication.com/irph/ijece/ijecev5n1__3.pdf
7. https://pdfs.semanticscholar.org/df57/2b11b245267686aba59a564fd56f472cf845.pdf
8. http://www.intelligence.tuc.gr/~petrakis/courses/computervision/filtering.pdf