ECE 6560 Final Project
Perona-Malik Anisotropic Diffusion and its Application in Image Denoising

Aanish Nair
gtID: 903700104
School of Electrical and Computer Engineering
Georgia Institute of Technology

1 Problem Statement

People constantly collect photographs for personal, professional, medical, or scientific needs, and there is a growing desire for higher quality and visually pleasing images. During acquisition, compression, and transmission, however, noise is introduced to the image, and it degrades. This makes post-processing and analysis more difficult.

Noise Removal techniques are employed to remove these noises that distort the image. This is a difficult process because different types of noise degrade the image, and it's also necessary to save as much detail and information in the image as possible. It's a significant problem to keep the sharpness of the image's edges, corners, and other sharp structures while smoothing the image.

High-frequency irrelevant information that is present in photographs is referred to be noise. Applying a Gaussian Low Pass Filter to the image is a straightforward way to reduce degradation and denoise it. The pixel values are diffused across the image with this filtering. This has the unintended effect of blurring out the edges of an image, which are likewise high-frequency yet significant information. Isotropic Diffusion is another name for this form of filtering, and it is ineffective for image denoising.

We're looking for a non-linear solution that smooths the inside of a region while preserving the edges in this project. Anisotropic Smoothing and Perona-Malik Anisotropic Denoising will be the subject of this report. The Finite Differences Method is used to create it. Section 2 will introduce the mathematical notion, Section 3 will deduce it, Section 4 will discretize it, and Section 5 will offer the experimental results, followed by a summary and discussion of what I've learnt.

2 Mathematical Concept

In this section, I will introduce the problem mathematically.

Let $\Omega \subset \mathbf{R}^2$ denote a subset of the plane and let $I(\cdot,t) : \Omega \to \mathbf{R}$ be a family of gray scale images. $I(\cdot,0)$ is the input image. Then, anisotropic diffusion is defined as

$$\frac{\partial I}{\partial t} = div(c(x,y,t)\nabla I) = \nabla c \cdot \nabla I + c(x,y,t)\Delta I$$

where Δ denotes the Laplacian, ∇ denotes the Gradient, $\mathbf{div}(\cdots)$ is the divergence operator and $c(x,y,t)$ is the diffusion coefficient.

$c(x,y,t)$ controls the rate of diffusion and is usually chosen as a function of the image gradient so as to preserve the edges in the image. [1] implemented their idea on anisotropic diffusion and proposed two functions for the diffusion coefficient:

$$c(||\nabla I||) = e^{-(||\nabla I||/K)^2} \tag{1}$$

and

$$c(||\nabla I||) = \frac{1}{1 + \left(\frac{||\nabla I||}{K}\right)^2} \tag{2}$$

the constant K controls the sensitivity to edges and is usually chosen experimentally or as a function of the noise in the image.

3 Derivation

I will derive the Partial Differential Equation (PDE) that I will utilize to solve the problem in this part.

Assume that we have an image I

$$I(x,y,t)$$

Noise is frequently manifested as a high-frequency signal. As a result of this, the energy function is increased. We want to find an energy function that is as low as possible. Let's start with a heat equation:

$$I_t = \Delta I = I_{xx} + I_{yy} \tag{3}$$

From the above, we develop the energy function of the form:

$$E(I) = \frac{1}{2}\int\int ||\nabla I||^2 dxdy \tag{4}$$

Now, let us consider the general form of the energy function in (4):

$$E(I) = \int_\Omega C(||\nabla I||)dxdy \tag{5}$$

where C : R $\rightarrow$ R and is increasing.

$$C(||\nabla I||) = L(I, I_x, I_y, x, y) \tag{6}$$

Here, L is called the Lagrangian. We can also look at C as a penalty. The higher the gradient, the greater the energy. The Euler-Langrange equation for (5) is given by:

$$L_I - \frac{\partial}{\partial x}L_{I_x} - \frac{\partial}{\partial y}L_{I_y} = 0 \tag{7}$$

We move forward by computing the individual terms in (7)

$$\begin{aligned} L_{I_x} &= \frac{\partial}{\partial x}C(||\nabla I||) \\ &= \frac{\partial}{\partial x}C(\sqrt{I_x^2 + I_y^2}) \\ &= C(\sqrt{I_x^2 + I_y^2})\frac{I_x}{\sqrt{I_x^2 + I_y^2}} \\ &= C'(||\nabla I||)\frac{I_x}{||\nabla I||} \end{aligned} \tag{8}$$

Similarly for L_{I_y} we find,

$$L_{I_y} = C'(||\nabla I||)\frac{I_y}{||\nabla I||} \tag{9}$$

Since $L = C(||\nabla I||) = C\left(\sqrt{I_x^2 + I_y^2}\right)$. Since it is independent of I it does not have a derivative with respect to I. Therefore,

$$L_I = 0 \tag{10}$$

The Gradient Descent PDE is defined as:

$$\nabla E = -I_t \tag{11}$$

Combining the equations (7), (8), (9), (10), (11), the Gradient Descent is:

$$\begin{aligned} I_t &= -L_I + \frac{\partial}{\partial x} L_{I_x} + \frac{\partial}{\partial y} L_{I_y} \\ &= \frac{\partial}{\partial x}\left(C'(||\nabla I||)\frac{I_x}{||\nabla I||}\right) + \frac{\partial}{\partial y}\left(C'(||\nabla I||)\frac{I_y}{||\nabla I||}\right) \\ &= \nabla \cdot \left(\frac{C'(||\nabla I||)}{||\nabla I||}\nabla I\right) \end{aligned} \tag{12}$$

Now that we've deduced the PDE (12) from the energy function, I'll turn my attention to Perona-Malik anisotropic diffusion. The diffusion coefficient is an edge-seeking function in this case.

The anisotropic diffusion equation is given by:

$$I_t = \nabla \cdot (C(||\nabla I||)\nabla I) \tag{13}$$

The edge seeking function $C(s)$ is chosen so that it satisfies two conditions (theoretically).

1. $\lim_{s\to\infty} = 1$ so that the rate of diffusion is high within uniform or inner regions.
2. $\lim_{s\to\infty} = 0$ so that there is zero diffusion along the boundaries.

An important property of the edge function is that it should have a zero value or a very insignificant value for the gradients that correspond to the edges of the image.

Example

- At the interior, we ideally should have $C(||\nabla I||) = 1$:

$$I_t = \nabla \cdot (1 * \nabla I) = \Delta I \tag{14}$$

- At the edges, we ideally should have $C(||\nabla I||) = 0$

$$I_t = \nabla \cdot (0 * \nabla I) = 0 \tag{15}$$

Now, because we don't know the image ahead of time, we have to figure out how to tell the difference between the image's edge and its content. We devise an edge-seeking algorithm. The gradient of an image is a good indicator of where the edges or content are located. The gradient of an image I is given by the formula:

$$||\nabla I|| = \sqrt{I_x^2 + I_y^2} \tag{16}$$

The beauty of the gradient of the image is that it is large at the edges and it is small at the content of the image.
Therefore, the anisotropic diffusion equation becomes:

$$I_t = C(||\nabla I||)\Delta I \tag{17}$$

The choice for the diffusion coefficient should such that when $||\nabla I||$ is large, the diffusion coefficient is small and when $||\nabla I||$ is small, the diffusion coefficient is large i.e., it is a monotonically decreasing function.

[1] presented coefficient functions (1) and (2), but I will focus my experiments on (2) and observe the effect of K on the result.

4 Discretization and Implementation

In this section, I will first start with the discretization of (13) and move on to the implementation details.

We can expand (13) as follows:

$$I_t = \nabla \cdot (C(||\nabla I||)\nabla I)$$

$$I_t = \nabla C(||\nabla I||)\nabla I + C(||\nabla I||)\nabla \cdot \nabla I \tag{18}$$
$$I_t = \nabla C(||\nabla I||)\nabla I + C(||\nabla I||)\Delta I$$

Using (2) in (18), we get:

$$I_t = \frac{\partial C(||\nabla I||)}{\partial x} I_x + \frac{\partial C(||\nabla I||)}{\partial y} I_y + \frac{1}{1 + \left(\frac{||\nabla I||}{K}\right)^2} \cdot (I_{xx} + I_{yy}) \tag{19}$$

We proceed to solve the individual terms in the above equation.

The term $\frac{\partial C(||\nabla I||)}{\partial x}$ is computed as follows:

$$\frac{\partial C(||\nabla I||)}{\partial x} = \frac{\partial}{\partial x} \frac{1}{1 + \left(\frac{||\nabla I||}{K}\right)^2}$$

$$\frac{\partial C(||\nabla I||)}{\partial x} = \frac{\partial}{\partial x}\frac{1}{1+\frac{I_x^2+I_y^2}{K^2}}$$

$$\frac{\partial C(||\nabla I||)}{\partial x} = -\frac{1}{\left(1+\frac{I_x^2+I_y^2}{K^2}\right)^2}\cdot\frac{\partial}{\partial x}\left(1+\frac{I_x^2+I_y^2}{K^2}\right)$$

$$\frac{\partial C(||\nabla I||)}{\partial x} = -\frac{1}{\left(1+\frac{I_x^2+I_y^2}{K^2}\right)^2}\cdot\frac{2I_xI_{xx}+2I_yI_{xy}}{K^2} \tag{20}$$

Now, the term $\frac{\partial C(||\nabla I||)}{\partial y}$ is given by:

$$\frac{\partial C(||\nabla I||)}{\partial y} = \frac{\partial}{\partial y}\frac{1}{1+\left(\frac{||\nabla I||}{K}\right)^2}$$

$$\frac{\partial C(||\nabla I||)}{\partial y} = \frac{\partial}{\partial y}\frac{1}{1+\frac{I_x^2+I_y^2}{K^2}}$$

$$\frac{\partial C(||\nabla I||)}{\partial y} = -\frac{1}{\left(1+\frac{I_x^2+I_y^2}{K^2}\right)^2}\cdot\frac{\partial}{\partial y}\left(1+\frac{I_x^2+I_y^2}{K^2}\right)$$

$$\frac{\partial C(||\nabla I||)}{\partial y} = -\frac{1}{\left(1+\frac{I_x^2+I_y^2}{K^2}\right)^2}\cdot\frac{2I_yI_{yy}+2I_xI_{xy}}{K^2} \tag{21}$$

From (19), (20) and (21), we have:

$$I_t = -\frac{1}{\left(1+\frac{I_x^2+I_y^2}{K^2}\right)^2}\cdot\frac{2I_xI_{xx}+2I_yI_{xy}}{K^2}I_x - \frac{1}{\left(1+\frac{I_x^2+I_y^2}{K^2}\right)^2}\cdot\frac{2I_yI_{yy}+2I_xI_{xy}}{K^2}I_y + \frac{1}{1|\left(\frac{||\nabla I||}{K}\right)^2}\cdot(I_{xx}+I_{yy})$$

$$I_t = \frac{K^2(I_{xx}+I_{yy})+(I_{xx}+I_{yy})(I_y^2-I_x^2)-4I_xI_yI_{xy}}{\left(K+\frac{I_x^2+I_y^2}{K}\right)^2} \tag{22}$$

We can use Forward Difference and Central Difference to compute the above partial derivatives.

The formulas for computation of the partial derivatives are as given below:

$$I_t(x,y,t) = \frac{I(x,y,t+\Delta t)-I(x,y,t)}{\Delta t}$$

$$I_x(x,y,t) = \frac{I(x+\Delta x,y,t)-I(x-\Delta x,y,t)}{2\Delta x}$$

$$I_y(x,y,t) = \frac{I(x,y+\Delta y,t)-I(x,y-\Delta x,t)}{2\Delta y}$$

$$I_{xx}(x,y,t) = \frac{I(x+\Delta x,y,t) - 2I(x,y,t) + I(x-\Delta x,y,t)}{\Delta x^2}$$

$$I_{yy}(x,y,t) = \frac{I(x,y+\Delta y,t) - 2I(x,y,t) + I(x,y-\Delta y,t)}{\Delta y^2}$$

Now that we have the formulas to calculate the partial derivatives, we have to find a way to dicretize the process so as to implement it on a computer. After doing a study of various approaches, I found the approach used in [2] reasonable which is called the **Four Nearest Neighbor Discretization** which takes into account all four neighbors around the pixel as opposed to the central difference that we did in class which took into consideration two neighbors in one direction.

The pixel value of an image at point (x,y) is updated in this discretization process with respect to its four closest neighbors, namely those that are above it, below it, to the left of it, and to the right of it.The formulas are given below and here the Δ operator signifies the difference:

$$\Delta I_N(x,y) = I(x-1,y) - I(x,y)$$

$$\Delta I_S(x,y) = I(x+1,y) - I(x,y)$$

$$\Delta I_W(x,y) = I(x,y-1) - I(x,y)$$

$$\Delta I_E(x,y) = I(x,y+1) - I(x,y)$$

For the coefficient function, [2] estimates the norm of the gradient of the image by the absolute value of the difference in the particular direction. The coefficient function in the different directions are discretized as follows:

$$C_N = C_N(||\nabla I||) \approx C(||\Delta I_N(x,y)||)$$

$$C_S = C_S(||\nabla I||) \approx C(||\Delta I_S(x,y)||)$$

$$C_W = C_W(||\nabla I||) \approx C(||\Delta I_W(x,y)||)$$

$$C_E = C_E(||\nabla I||) \approx C(||\Delta I_E(x,y)||)$$

Finally the update equation takes the form,

$$I^{t+1}(x,y) = I^t(x,y) + \lambda\left[C_N \cdot \Delta I_N^t(x,y) + C_S \cdot \Delta I_S^t(x,y) + C_W \cdot \Delta I_W^t(x,y) + C_E \cdot \Delta I_E^t(x,y)\right] \quad (23)$$

Here, $I^t(x,y)$ is the pixel value of the image at position (x,y) at time t and $I^{t+1}(x,y)$ is the pixel value of the image at position (x,y) at time $t+1$. λ is the stability of the numerical

scheme and $0 \leq \lambda \leq 1/4$.

According to [2], it is feasible to demonstrate that the discretized scheme still meets the maximum (and minimum) principle, regardless of the gradient approximation used.

This can be shown directly from equation (23) using the fact that $\lambda \in [0, 1/4]$, and $c \in [0, 1]$. Let us define the following.

The maximum of the neighbors of $I(x,y)$ at iteration t:

$$I^t_{max}(x,y) \doteq \max[(I, I_N, I_S, I_E, I_W)^t(x,y)]$$

The minimum of the neighbors of $I(x,y)$ at iteration t:

$$I^t_{min}(x,y) \doteq \min[(I, I_N, I_S, I_E, I_W)^t(x,y)]$$

We can prove that

$$I^t_{min}(x,y) \leq I^{t+1}(x,y) \leq I^t_{max}(x,y)$$

The above equation proves that there are no local maxima or minima possible in the interior of the discretized scale-space.

$$I^{t+1}(x,y) = I^t(x,y) + \lambda\,[C_N \cdot \Delta I^t_N(x,y) + C_S \cdot \Delta I^t_S(x,y) + C_W \cdot \Delta I^t_W(x,y) + C_E \cdot \Delta I^t_E(x,y)]$$

$$I^{t+1}(x,y) = I^t(x,y)\left(1 - \lambda\,(C_N + C_S + C_W + C_E)^t(x,y)\right) + \lambda\,[C_N \cdot I^t_N(x,y) + C_S \cdot I^t_S(x,y) + C_W \cdot I^t_W(x,y) + C_E \cdot I^t_E(x,y)]$$

$$I^{t+1}(x,y) \leq I^t_{max}(x,y)\left(1 - \lambda\,(C_N + C_S + C_E + C_W)^t(x,y)\right) + \lambda I^t_{max}(C_N + C_S + C_E + C_W)^t(x,y)$$

Therefore,

$$I^{t+1}(x,y) \leq I^t_{max}(x,y)$$

Similarly, we obtain,

$$I^{t+1}(x,y) \geq I^t_{min}(x,y)\left(1 - \lambda\,(C_N + C_S + C_E + C_W)^t(x,y)\right) + \lambda I^t_{min}(x,y)(C_N + C_S + C_W + C_E)^t(x,y) = I^t_{min}(x,y)$$

The algorithm would start from a noisy image and then we perform the diffusion process by updating the pixel values in the image according to the update equation (23) for a specified number of iterations to eventually denoise the image.

5 Experiments and Results

I will be performing my experiments on a grayscale image. I will begin by explaining the noises that I add to the image followed by an introduction to the evaluation metrics that I will be using and then I will proceed with my experiments to study the effect of K on (17).

5.1 Types of Noises

5.1.1 Gaussian Noise

Gaussian Noise is a type of statistical noise that has a probability density function that is equal to that of a normal distribution, also known as a Gaussian distribution. To generate the noise, a random Gaussian function is mixed with the image function. Since this type of noise forms in amplifiers or detectors, it is also called as electronic noise.

5.1.2 Salt and Pepper Noise

Salt and Pepper Noise introduces both extremely bright (value = 255) and extremely dark (value = 0) pixels all across the image randomly. Since this type of noise randomly drops the original data values where random white and dark points are introduced, it is also called as Data Drop Noise.

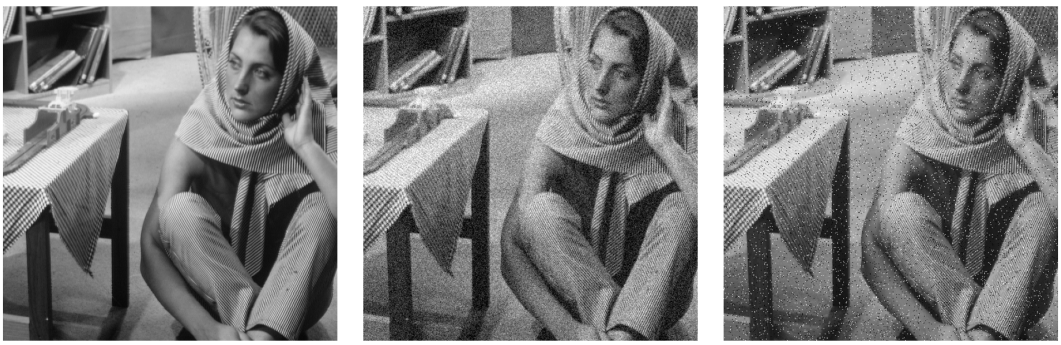

Figure 1: The Original image, Gaussian Noise added to the image, Salt and Pepper Noise added to the image

5.2 Evaluation Metrics

5.2.1 Edge Detection

Since I want to evaluate how well the denoising technique retains edge information, edge detection should be one of the evaluation metrics. This can be done by convolving two kernels which can be used to extract both the vertical and horizontal edges. We get the edge information for the whole image by adding both the edges.

$$\text{HorizontalEdge}(I_x) = I * K_x$$

$$\text{VerticalEdge}(I_y) = I * K_y$$

$$\text{Edge}(I_x + I_y) = \text{HorizontalEdge} + \text{VerticalEdge}$$

where,

$$K_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

K_x and K_y are also known as Sobel Kernels.

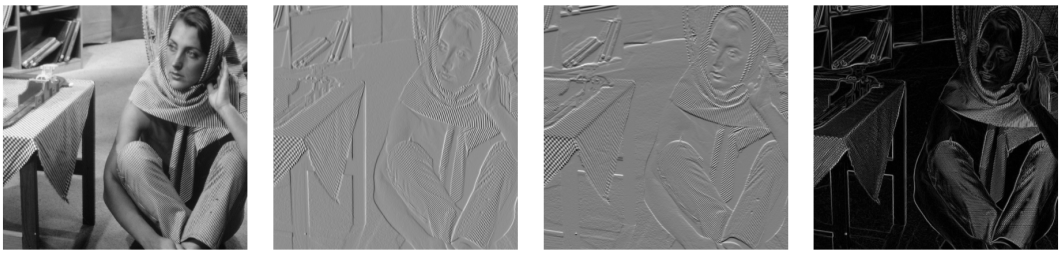

Figure 2: The Original Image, Horizontal Edges, Vertical Edges and the Edge Detection result

5.2.2 PSNR

The Peak Signal-To-Noise Ratio (PSNR) is the ratio between the maximum possible power of a signal and the power of the noise that affects its representation. It can be defined in terms of Mean Squared Error (MSE) by:

$$PSNR(\text{dB}) = 20\log_{10}\left(\frac{MAXVAL}{MSE}\right) \tag{24}$$

where, *MAXVAL* is the maximum pixel value of the image and

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I(i,j) - K(i,j))^2 \tag{25}$$

Here, the imaage I is of dimension mxn and K is the noisy approximation of the image.

5.3 Perona-Malik Anisotropic Diffusion

I will now present results of the Perona-Malik Anisotropic Diffusion. I will first run the PDE on the image with Gaussian noise and then proceed to run the PDE on the image with Salt and Pepper Noise.

For the Gaussian Noise, I run the PDE for 80 iterations with K $= 0.1$ and $\lambda = 0.1$.

Figure 3: Result of PDE on image with Gaussian Noise at time steps 0 and 20

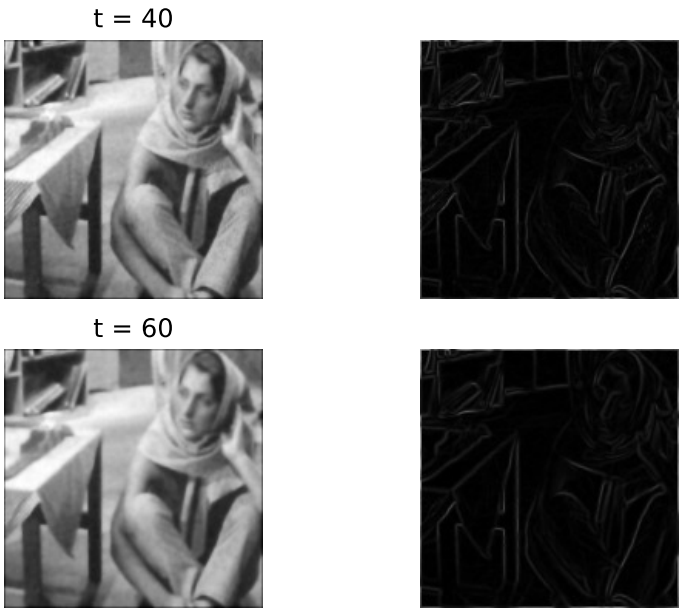

Figure 4: Result of PDE on image with Gaussian Noise at time steps 40 and 60

Figure 5: Result of PDE on image with Gaussian Noise at time step 80

Here, we can see that the PDE does a good job of removing the noise at a time step of 20 itself. Also we notice that the edges are also well preserved as compared to the Gaussian Low Pass Filter.

For the Salt and Pepper Noise, I run the PDE for 160 iterations with K $= 0.1$ and $\lambda = 0.1$.

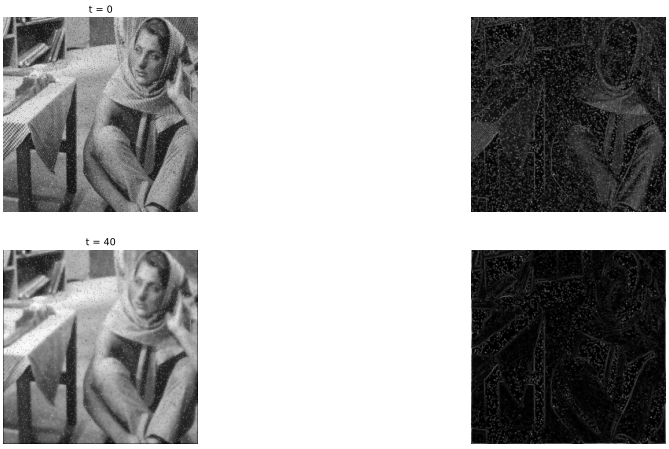

Figure 6: Result of PDE on image with Salt and Pepper Noise at time steps 0 and 40

Figure 7: Result of PDE on image with Salt and Pepper Noise at time steps 80 and 120

Figure 8: Result of PDE on image with Salt and Pepper Noise at time step 160

Here, we can see that the PDE is not able to remove the noise to a great extent at time step 40 but it eventually does denoise the image.

Let us observe the result by increasing K to 0.2.

Figure 9: Result of PDE on image with Salt and Pepper Noise at time steps 0 and 40

Figure 10: Result of PDE on image with Salt and Pepper Noise at time steps 80 and 120

Figure 11: Result of PDE on image with Salt and Pepper Noise at time step 160

Here, we can see that the PDE does a better job of denoising the image at an earlier time step as compared to the PDE with K = 0.1. This result demonstrates the effect of K and how it determines the sensitivity and speed of the PDE. Although we obtain good results faster with a higher K value, we are also prone to losing edge information more faster. Therefore, determining a good value of K is an experiment worth conducting.

5.4 Finding the optimal K value

From previous results, we get an idea on the relationship between K and the results of Perona-Malik Anisotropic Diffusion. We now wan to find the optimal value of K for both the Gaussian Noise and Salt and Pepper Noise.
The parameters that I set to test the effect of K on the diffusion results are 80 iterations for Gaussian Noise and 160 iterations for Salt and Pepper Noise. I keep the λ value as 0.1 and I sweep the values for K from 0.01 to 0.6. It is worth noting that these results are particular to these parameter values and the choice of image.

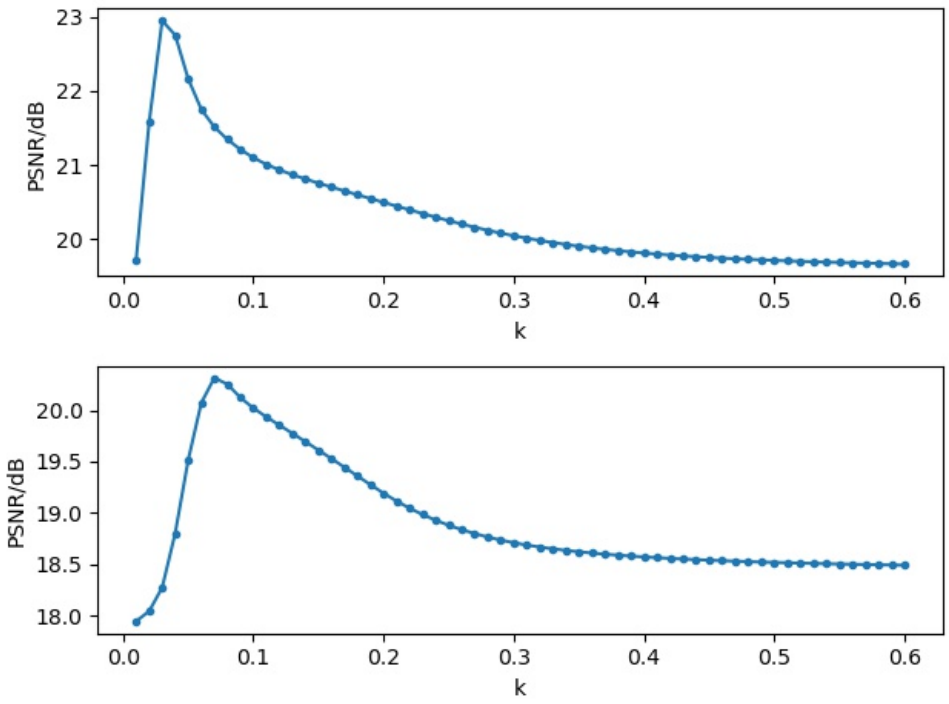

Figure 12: Variation of PSNR with the value of K for Gaussian and Salt and Pepper Noise

From Fig. 12, we can see that the optimal value for Gaussian Noise is 0.03 while the optimal value for Salt and Pepper Noise is 0.07. This result may mean that we need a higher K value to denoise Salt and Pepper Noise as compared to Gaussian Noise.

In the process of conducting the above experiments, I found it interesting to explore other diffusion functions or other denoising techniques. In the following sections, I will present my experimental results with another diffusion function and the FFT based Denoising technique.

5.5 A different Diffusion Coefficient

[3] proposed a different edge stopping function for anisotropic diffusion that is given as follows:

$$C(||\nabla I||) = \frac{1}{1+\left(\frac{||\nabla I||}{K}\right)^{\alpha(||\nabla I||)}} \tag{26}$$

where,

$$\alpha(||\nabla I||) = 2 - \frac{2}{1+\left(\frac{||\nabla I||}{K}\right)^{2}} \tag{27}$$

The results of the PDE with the new diffusion coefficient are shown below.

Figure 13: Result of PDE with new diffusion coefficient on image with Gaussian Noise at time steps 0 and 20

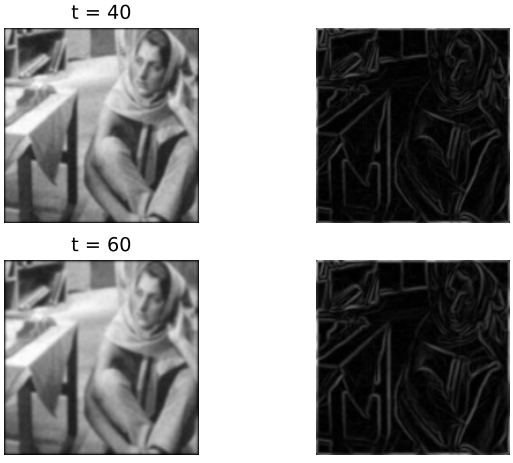

Figure 14: Result of PDE with new diffusion coefficient on image with Gaussian Noise at time steps 40 and 60

Figure 15: Result of PDE with new diffusion coefficient on image with Gaussian Noise at time step 80

Figure 16: Result of PDE with new diffusion coefficient on image with Salt and Pepper Noise at time steps 0 and 40

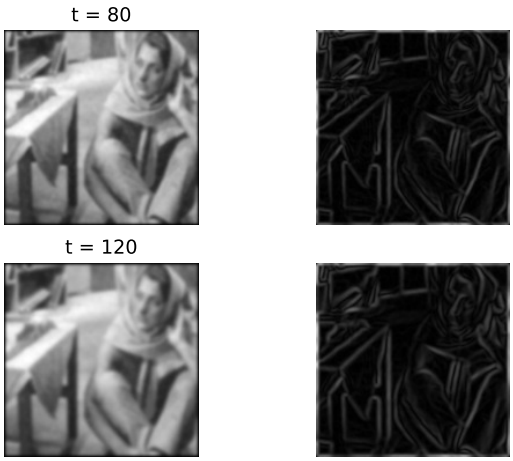

Figure 17: Result of PDE with new diffusion coefficient on image with Salt and Pepper Noise at time steps 80 and 120

Figure 18: Result of PDE with new diffusion coefficient on image with Salt and Pepper Noise at time step 160

Here the PDE is run with the same parameters as in the previous section. On comparing with Fig. 3 and Fig. 13 we can see the diffusion coefficient in [3] does not improve much on the previous one on the image with Gaussian Noise. But there is a difference when it comes to the Salt and Pepper Noise where the PDE with diffusion coefficient 26 completes denoising at time step 40 with K = 0.1 which could not be done with the previous diffusion coefficient. This effect is also evident when comparing the PSNR of the two different diffusion coefficients.

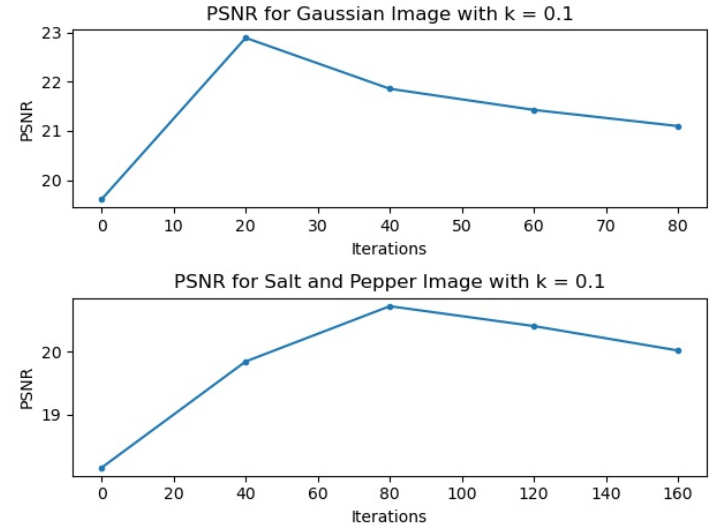

Figure 19: Plot of PSNR vs iterations using Perona-Malik Diffusion Coefficient on Gaussian Noise (Top) and Salt and Pepper Noise (Bottom)

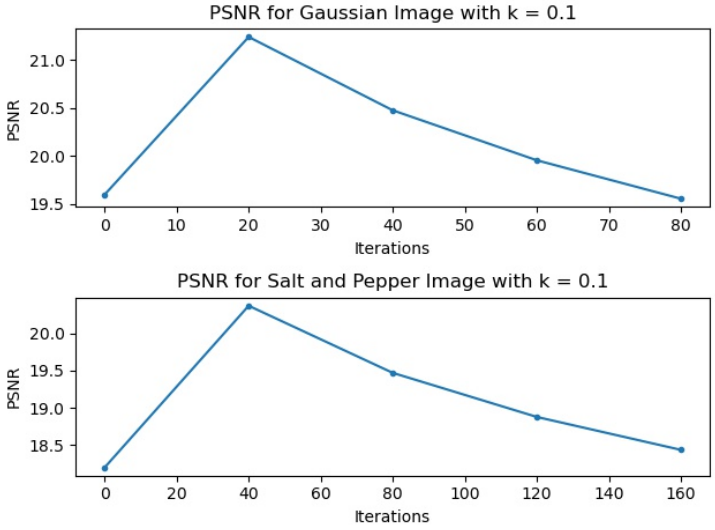

Figure 20: Plot of PSNR vs iterations using Wei's Diffusion Coefficient on Gaussian Noise (Top) and Salt and Pepper Noise (Bottom)

We can see on comparing Fig. 19 and Fig. 20, the PSNR results on the Gaussian Noise image are identical between both diffusion coefficients. But, when it comes to Salt and Pepper Noise, we see that the diffusion coefficient proposed by [3] reaches a high PSNR at earlier number of iterations as compared to [1]. We also see that the PSNR drops sharply while using Wei's diffusion coefficient since the diffusion on the image is done and the PDE then proceeds to denoise the useful edges.

5.6 Other Diffusion Techniques

In this section I will explore other diffusion techniques and compare the result to that of Perona-Malik Anisotropic Diffusion.

5.6.1 Performance of Gaussian Blur

To start with, I will present the results of Gaussian Blur on the noisy images. The Gaussian kernel has a variance (σ) of 5.

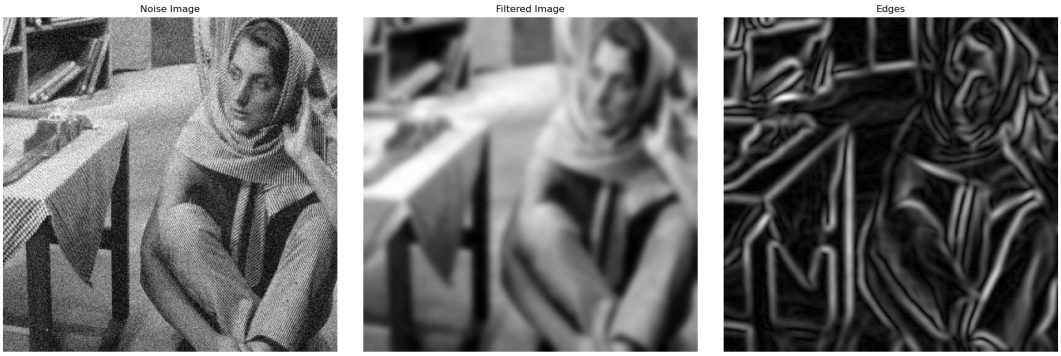

Figure 21: Result of Gaussian Blur on image with Gaussian Noise

Figure 22: Result of Gaussian Blur on image with Salt and Pepper Noise

From Fig. 21 and Fig. 22, we can see that the edge are blurred although the noise has been removed. Hence, we can say that Gaussian Blur or Gaussian Low Pass Filtering is not effective for Image Denoising.

5.6.2 FFT Base Image Denoising

A noisy image can be transformed from the spatial to the frequency domain with the help of the Fast Fourier Transform (FFT). This transformed image is then filtered with the help of High Pass Filters (HPF) and Low Pass Filters (LPF). The high pass filtered image is subjected to an inverse FFT where thresholding is applied to preserve the sharpness in the image. The low pass filter is subjected to an inverse FFT which is then passed through a PWL filter to enhance image smoothness. The combination of the resulting images gives us our denoised result. The code for this part is referenced from [4].

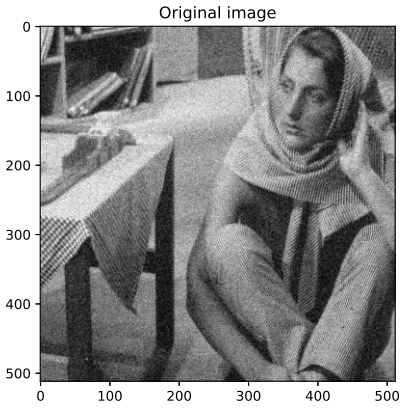

Figure 23: Noisy Image for FFT approach

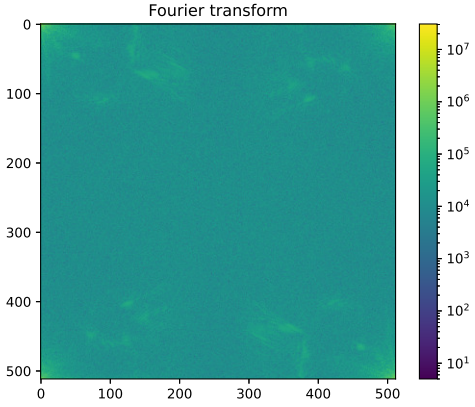

Figure 24: Fourier Transform of the Noisy Image

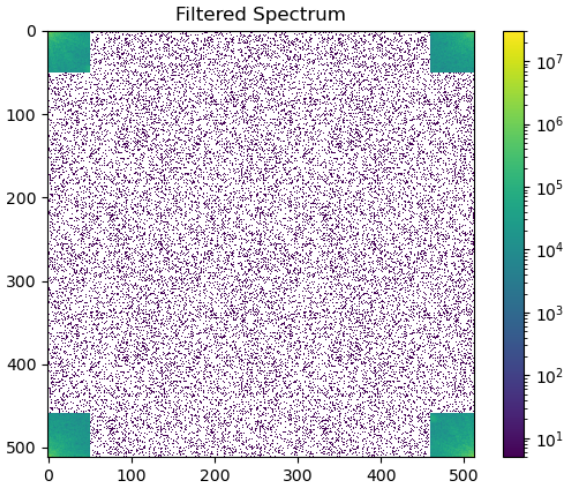

Figure 25: Filtering the Noise in the Image

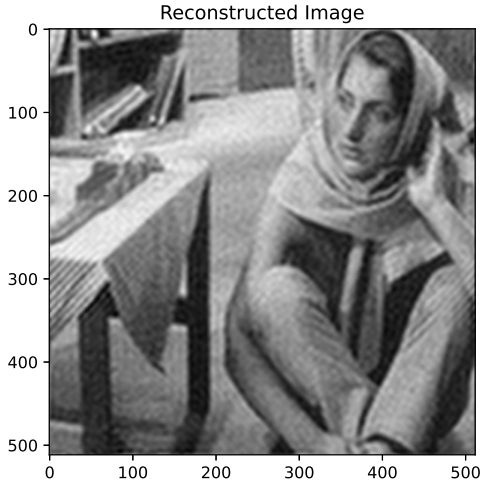

Figure 26: Denoised Image

We can see from the results that the FFT based approach although reduces the noise in the image, there still some artifacts in the result and it also does not retain the sharpness of the image as much as the PDE based denoising method does.

5.6.3 Deep Learning Based Approach

Although this is not in the scope of the project, I was interested to see how a Deep Learning Based Denoising approach would work as compared to a Partial Differential Equation approach. I will not get into much detail about the approach. [5] is a transfomer model that mitigates the shortcomings of a CNN (limited receptive field and inadaptability to input

(a) Gaussian Noise Image

(b) Salt and Pepper Image

Figure 27: Noisy images that are input to the model

images). As compated to other transformer models, this one is an efficient one and is named Restoration-Transformer (Restormer).

It achieves state-of-the-art results on several image restoration tasks, including image deraining, single-image motion deblurring, defocus deblurring (single-image and dual-pixel data), and image denoising (Gaussian grayscale/color denoising, and real image denoising).

These are the images that I use to test the model. You can test out the model with your own images at `https://colab.research.google.com/drive/1C2818h7KnjNv4R1sabe14_AYL7lWhmu6?usp=sharing`

The results from the model are given below.

Figure 28: Denoised Result for Gaussian Noise image using Deep Learning

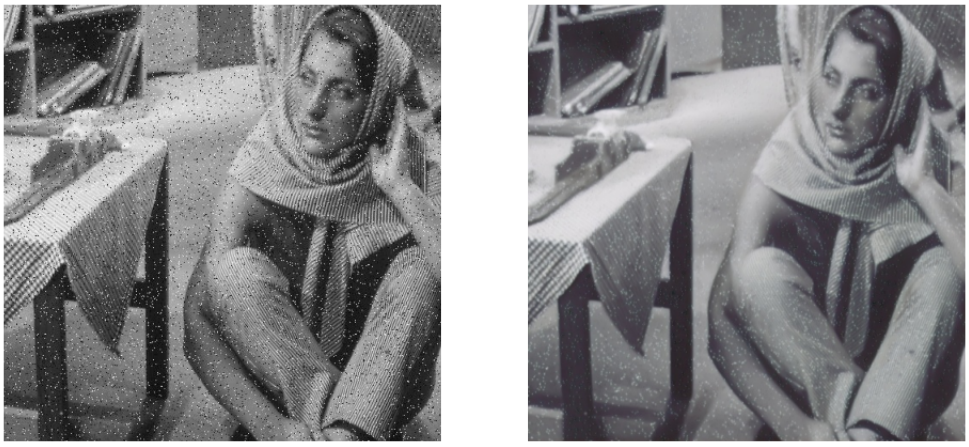

Figure 29: Denoised Result for Salt and Pepper Noise image using Deep Learning

We can see that the Deep Learning method does well on the image with Gaussian Noise, but when it comes to Salt and Pepper Noise, there are some artifacts that remain in the result which shows that Deep Learning methods may denoise the image more "globally" while PDEs and their "local" behaviour might be more suitable to such applications.

5.7 Comparison

PSNR Comparison across various methods					
	Perona Malik	**Wei**	**Gaussian Blur**	**FFT**	**Restormer (Deep Learning)**
PSNR on Gaussian Noise	20.9	21.5	21	23	23.9
PSNR on Salt and Pepper Noise	18.3	20.5	21	22.8	23.4

Table 1: PSNR of different algorithms on Gaussian and Salt and Pepper Noise

We also make the observation that the Perona-Malik anisotropic diffusion has a better result on Gaussian Noise as compared to Salt and Pepper Noise. This similar trend is observed in all other methods except Gaussian Blur owing to its isotropic diffusion. This can lead us to make the conclusion that Salt and Pepper noise is much difficult to denoise as compared to Gaussian Noise. We also observe that anisotropic diffusion with [3]'s diffusion coefficient does much better that the diffusion coefficient proposed by [1].

From the table above, we can see the PSNR of various denoising algorithms on the images with Gaussian and Salt and Pepper Noise. We observe that the FFT based denoising techniques shows a better result on the objective measure (PSNR), but it does not look as good on observation. This same observation applies to the Gaussian Blur results.

So as to prove the point that anisotropic diffusion retains the structure of the image (edge information), I will compare the methods with SSIM being the metric of comparison.

SSIM (Structural similarity) is a method that predicts the perceived quality of images. It measures the similarity of two images. It is a full reference metric i.e., the measurement or prediction of image quality is based on an initial distortion free image as reference [6]. As compared to PSNR, SSIM compares the luminance, contrast and structure between images.

SSIM Comparison across various methods					
	Perona Malik	**Wei**	**Gaussian Blur**	**FFT**	**Restormer (Deep Learning)**
SSIM on Gaussian Noise	0.6681	0.6034	0.5175	0.5685	0.7043
SSIM on Salt and Pepper Noise	0.3595	0.5586	0.5179	0.5885	0.6493

Table 2: SSIM of different algorithms on Gaussian and Salt and Pepper Noise

The above table presents the SSIM of the different methods on Gaussian Noise and Salt and Pepper Noise images. We can see that anisotropic diffusion does a better job as compared to FFT and Gaussian Blur in preserving the structure in the image. We can also see that the Perona-Malik diffusion does not do well on Salt and Pepper Noise with $k = 0.1$. We also see an improvement when we move to [3]'s diffusion coefficient.

6 Conclusion

In conclusion, in this project I implemented Perona-Malik anisotropic diffusion and conducted experiments to evaluate the effect of the parameter K which controls the PDEs senstitivity to the edges of the image.

I started with the problem statement about denoising images, why it is difficult and how Perona-Malik diffusion is useful in denoising images. In section 2, I introduced the problem statement in a mathematical way. In section 3, I derived the PDE from the energy function and continues to discretize it in section 4. In section 5, I conducted my experiments and presented the results from those experiments.

I believe I learnt a lot through this project. I now have a better understanding of anisotropic diffusion in specific Perona-Malik anisotropic diffusion. I understadn that the Perona-Malik diffusion coefficient is like an edge seeking function that directs us to the area of interest. I also gained an understanding of how the parameter K controls the sensitivity to the edges of the image. Though my experiments, I also understand that the parameter differs on different images and that finding the optimal K is indeed a task worth importance. I also understand now how anisotropic diffusion performs better in retaining image structure. On hindsight, I would have loved to extend the project and explore the interplay of PDEs and Neural networks. There are recent advances that focus on this area. [7] proposes a PDE-guided method for image denoising using small data.

I would like to thank Professor Yezzi and I am glad to have taken this class with him. His style of teaching and enthusiasm towards the world of PDEs engaged me during each lecture and this class helped me gain a lot of information which I hope to use in my career.

References

[1] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[2] P. Perona, T. Shiota, and J. Malik, "Anisotropic diffusion," in *Geometry-driven diffusion in computer vision*. Springer, 1994, pp. 73–92.

[3] G. W. Wei, "Generalized perona-malik equation for image restoration," *IEEE Signal processing letters*, vol. 6, no. 7, pp. 165–167, 1999.

[4] "Image denoising using fft," https://scipy-lectures.org/intro/scipy/auto_examples/solutions/plot_fft_image_denoise.html, accessed: 04-21-2022.

[5] S. W. Zamir, A. Arora, S. Khan, M. Hayat, F. S. Khan, and M.-H. Yang, "Restormer: Efficient transformer for high-resolution image restoration," in *CVPR*, 2022.

[6] Wikipedia contributors, "Structural similarity — Wikipedia, the free encyclopedia," 2022, [Online; accessed 25-April-2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Structural_similarity&oldid=1073788800

[7] J. Jeon, P. Kim, B. Jang, and Y. Kim, "Pde-guided reservoir computing for image denoising with small data," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 31, no. 7, p. 073103, 2021.

[8] J. Weickert, *Anisotropic diffusion in image processing*. Teubner Stuttgart, 1998, vol. 1.

[9] V. Kamalaveni, R. A. Rajalakshmi, and K. Narayanankutty, "Image denoising using variations of perona-malik model with different edge stopping functions," *Procedia Computer Science*, vol. 58, pp. 673–682, 2015.

[10] S. Gopinathan, R. Kokila, and P. Thangavel, "Wavelet and fft based image denoising using non-linear filters." *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 5, no. 5, 2015.

[11] Wikipedia contributors, "Anisotropic diffusion — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Anisotropic_diffusion&oldid=1033347785, 2021, [Online; accessed 16-April-2022].

Appendices

The appendix contains the code snippets that I used to run the experiments in the project. The repository can be found at `https://github.com/nairaanish/ECE6560_Final_Project`

```
def noise_addition(noise, img):
    '''
    This function takes a noise type and an image as input arguments
    and returns
    a noisy image with the type of noise added to it.
    '''
    if noise == "gaussian":
        noisy_image = random_noise(img, 'gaussian', seed = None, clip =
        True)
        return noisy_image
    elif noise == 'saltpepper':
        noisy_image = random_noise(img, 's&p', seed = None, clip =
        True)
        return noisy_image

```

Snippet1:`noise_addition` function to add noise to an image

```
def edge_detection(img):
    '''
    This function takes an image an input and filters the horizontal
    and vertical
    edges of the image and puts them together to obtain the edges of
    the image
    '''
    vertical_edges = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]],
    np.int32)
    horizontal_edges = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]],
    np.int32)
    dx_image = ndimage.filters.convolve(img / 255, vertical_edges)
    dy_image = ndimage.filters.convolve(img / 255, horizontal_edges)
    image_edges = np.hypot(dx_image, dy_image)
    return image_edges

```

Snippet2:`edge_detection` function to detect edges in an image

```
def gaussian_blur_edges(noise_img):
    '''
    This function takes in the noisy image as input and applies
    Gaussian Blur
    with a Gaussian kernel with sigma = 5 and returns the result.
    '''
    print('\nApplying the Gaussian Blur')

    noisy_image = np.array(Image.open(noise_img).convert('L'))

    gauss_img = ndimage.gaussian_filter(noisy_image, sigma = 5)

    imageio.imwrite('PATH_TO_RESULT', gauss_img)

    print('\nEdge Detection')

    gauss_img_edges = edge_detection(gauss_img)

    imageio.imwrite('PATH_TO_RESULT', gauss_img_edges)

    figure = plt.figure()
    plt.gray()
    plt.axis('off')

    ax1 = figure.add_subplot(131)
    plt.title("Noisy Image")
    plt.axis('off')

    ax2 = figure.add_subplot(132)
    plt.title('Filtered Image')
    plt.axis('off')

    ax3 = figure.add_subplot(133)
    plt.title('Edges')
    plt.axis('off')

    ax1.imshow(noisy_image)
    ax2.imshow(gauss_img)
    ax3.imshow(gauss_img_edges)

    figure.tight_layout()
    plt.show()
```

Snippet3:`gaussian_blur_edges` function to apply gaussian blur to a noisy image

```python
def coeff(I, k):
    '''
    This function calculates the diffusion coefficient for Perona-Malik
    anisotropic diffusion given inputs the image segment and the parameter k
    '''
    return 1 / (1 + ((I/k) ** 2))

def diffuse(img, log_f, iter, k, lmb = 0.01):
    '''
    This function performs Perona-Malik anisotropic diffusion on the
    input image for the input number of iterations using the 4 nearest neighbor discretization scheme.
    '''
    image = np.array(Image.open(img).convert('L')) / 255
    new_image = np.zeros(image.shape, dtype=image.dtype)

    result = [image]

    for t in range(iter):
        I_Up = image[:-2, 1:-1] - image[1:-1, 1:-1]
        I_Down = image[2:, 1:-1] - image[1:-1, 1:-1]
        I_Right = image[1:-1, 2:] - image[1:-1, 1:-1]
        I_Left = image[1:-1, :-2] - image[1:-1, 1:-1]

        new_image[1:-1, 1:-1] = image[1:-1, 1:-1] + lmb * (
            coeff(I_Up, k) * I_Up +
            coeff(I_Down, k) * I_Down +
            coeff(I_Right, k) * I_Right +
            coeff(I_Left, k) * I_Left
        )

        image = new_image

        if (t+1) % log_f == 0:
            result.append(image.copy())

    return result
```

Snippet4:Perona_Malik.py functions to perform Perona-Malik diffusion

```
def alpha(grad_I, k):
    '''
    This function calculates the exponential term in Wei's diffusion coefficient given the gradient of the image segment and the parameter k.
    '''
    return 2 - (2 / 1 + ((grad_I/k) ** 2))

def coeff_guo(I, k):
    '''
    This function calculates wei's diffusion coefficient give the image segment and
    the parameter k
    '''
    return 1 / (1 + ((np.linalg.norm(I)/k) ** alpha(np.linalg.norm(I), k)))

def diffuse_new(img, log_f, iter, k, lmb = 0.01):
    '''
    This function performs anisotropic diffusion using wei's diffusion coefficient
    on the input image for the input number of iterations using the 4 nearest neighbor discretization scheme.
    '''
    image = np.array(Image.open(img).convert('L')) / 255
    new_image = np.zeros(image.shape, dtype=image.dtype)

    result = [image]

    for t in range(iter):
        I_Up = image[:-2, 1:-1] - image[1:-1, 1:-1]
        I_Down = image[2:, 1:-1] - image[1:-1, 1:-1]
        I_Right = image[1:-1, 2:] - image[1:-1, 1:-1]
        I_Left = image[1:-1, :-2] - image[1:-1, 1:-1]

        new_image[1:-1, 1:-1] = image[1:-1, 1:-1] + lmb * (
            coeff_guo(I_Up, k) * I_Up +
            coeff_guo(I_Down, k) * I_Down +
            coeff_guo(I_Right, k) * I_Right +
            coeff_guo(I_Left, k) * I_Left
        )

        image = new_image

```

```
        if (t+1) % log_f == 0:
            result.append(image.copy())

    return result
```

Snippet5:Wei.py functions to perform diffusion using Wei's diffusion coefficient

```
def PSNR(img, ref):
    '''
    This function calculates the PSNR given the image and the reference
    ↪ image
    '''
    mse = np.mean((img - ref) ** 2)

    if mse == 0:
        return 100

    max_val = 1.0

    return 20 * math.log10(max_val / math.sqrt(mse))
```

Snippet6:PSNR function to calculate the PSNR

```
import numpy as np
import matplotlib.pyplot as plt
import cv2

im = plt.imread('results/barbara_gaussian.jpg')
plt.figure()
plt.imshow(im, plt.cm.gray)
plt.title('Original image')
plt.show()

from scipy import fftpack
im_fft = fftpack.fft2(im)

# Show the results

def plot_spectrum(im_fft):
    '''
    Function to plot the frequency spectrum of the input image
    '''
    from matplotlib.colors import LogNorm
    # A logarithmic colormap
    plt.imshow(np.abs(im_fft), norm=LogNorm(vmin=5))
```

```
    plt.colorbar()

plt.figure()
plot_spectrum(im_fft)
plt.title('Fourier transform')
plt.show()

# In the lines following, we'll make a copy of the original spectrum and
# truncate coefficients.

# Define the fraction of coefficients (in each direction) we keep
keep_fraction = 0.1

# Call ff a copy of the original transform. Numpy arrays have a copy
# method for this purpose.
im_fft2 = im_fft.copy()

# Set r and c to be the number of rows and columns of the array.
r, c = im_fft2.shape

# Set to zero all rows with indices between r*keep_fraction and
# r*(1-keep_fraction):
im_fft2[int(r*keep_fraction):int(r*(1-keep_fraction))] = 0

# Similarly with the columns:
im_fft2[:, int(c*keep_fraction):int(c*(1-keep_fraction))] = 0

plt.figure()
plot_spectrum(im_fft2)
plt.title('Filtered Spectrum')
plt.show()

# Reconstruct the denoised image from the filtered spectrum, keep only the
# real part for display.
im_new = fftpack.ifft2(im_fft2).real

plt.figure()
plt.imshow(im_new, plt.cm.gray)
# plt.title('Reconstructed Image')
plt.show()
cv2.imwrite('FFT_result.jpg', im_new)
```

Snippet6:`fft_denoise.py` functions to perform diffusion using FFT