Image Processing and Restoration Using Partial Differential Equations

Project Report

28.05.2020

Complex Variables and Partial Differential Equations (MAT2002)

Dr Bhupesh Datt Sharma

KRISHANU SHEKHAR

Abstract

Computers can analyze images and sequence of images contaminated by noise, or of inherent poor quality (something which matters a great deal in medical imaging). With the help of non-linear partial differential equations (PDEs), we will produce higher-quality images by enhancing its sharpness and filtering out noise. With the help of the Perona-Malik diffusion model, we will perform image smoothing in a selective manner. Geometrical equations of mean curvature flow will also play a vital role in the image processing application. The results will be shown in the forms of artificial and real 2D, and 3D images and a sequence of images.

Goal

The goal is to remove noise from 2-D and 3-D images using Perona-Malik Anisotropic Diffusion technique.

Introduction

The Heat Equation and the Perona-Malik Equation are diffusion equations commonly used for image processing because they use blurring to eliminate noise. While the Heat Equation is simpler and more symmetric than the Perona-Malik Equation, the latter does a better job of preserving key features, such as edges, in images.

We provide an introduction to the Perona-Malik diffusion model, describe and implement discretization schemes for it so that they can be used on real-world data, and observe its performance on images with artificially induced noise. We also present a curious edge-finding algorithm found while trying to implement Perona-Malik.

We conjecture as to why the edge-finding algorithm works and show samples of its output.

Diffusion Equations and Image Noise

When we take a photograph (or any make any measurement, for that matter) the data we generate will contain noise. Loosely speaking, noise is a random variation from the “true" value of the property we want to measure. For example, noise in a photograph could be random variation in the brightness of the image. Since noise is not reflective of the “true" value of the property we are measuring, it is generally considered bad.

It makes sense, then, that people are interested in ways to reduce noise. One common approach to noise reduction involves taking local, weighted averages of the measurement over its spatial domain (e.g. for a photograph we could take a local average of brightness). The idea is that by averaging out the measurement over small areas, we can account for any zero-sum random error in these areas. These approaches effectively simulate diffusion, the process by which heat or material density evens itself out over time. Perhaps unsurprisingly, they are described by diffusion equations, which are PDE that describe diffusive processes. We will focus our attention on the Perona-Malik Equation, a common diffusive PDE often used in image processing.

The Perona-Malik Equation

The Perona-Malik Equation was designed as an alternative to traditional Gaussian blurring with the goal of preserving more of the features of an image. Gaussian blurring, while an effective way to homogenize an image, does not do a good job of preserving features such as edges. Because it blurs equally across all parts of an image, edges are significant. In fact, since edges occur when local points have vastly different function values, edges are diffused extremely quickly with a Gaussian Blur; the extent to which a point's value changes under a blur is proportional to how much it differs from the point around it.

In order to combat this, Perona and Malik designed a modified Heat Equation in which the diffusion constant, D, varies in space according to the gradient of the image. in two dimensions, it looks like this:

$$u_t = D(\nabla u)(u_{xx} + u_{yy})$$

with D monotonically decreasing in ∇u. The idea is to make D large when the local function values are close together and small when local function values are far apart. This means that the equation will diffuse away edges (areas of a high gradient) slowly while di using interior regions (areas of a low gradient) quickly. Thus it will preserve edges while homogenizing within regions to eliminate noise.

In their original paper, Perona and Malik presented two possibilities for D(∇u), both parametrized by a constant K and having the property that they strictly decrease with ∇u. The first is an exponential decay, given by

$$D(\nabla u) = e^{-||\nabla u||^2/K^2}$$

and the second is a reciprocal given by

$$D(\nabla u) = \frac{1}{1 + \left(\frac{||\nabla u||}{K}\right)^2}.$$

Both have the property that they decay to zero as the gradient grows, however, Perona and Malik claimed that the former is better for high-contrast images while the latter performs well on images with large regions (i.e. not many edges near one another).

Throughout this discussion, we use the term edge, despite the fact that we do not have a formal definition of what constitutes an edge. However, we know qualitatively that edges occur when the gradient of an image is large. Moreover, we consider an edge “stronger" or perhaps “more important" if the gradient is especially large. Thus the idea of making the diffusion constant decrease as the gradient grows makes sense; the more “edge-like" or “stronger" an edge, the less we want to diffuse it away. Or, thought of another way, the longer we want it to take to diffuse the edge away. (Given sufficient time, the Perona-Malik Equation will homogenize an image entirely, just as a Gaussian Blur would.)

The Perona-Malik Equation:

$$u_t = \nabla \cdot \left(g\left(|\nabla u|\right) \nabla u \right),$$

where $g(s)$ is a smooth nonincreasing function with $g(0) = 1$, $g(s) \geq 0$ and $g(\infty) = 0$.

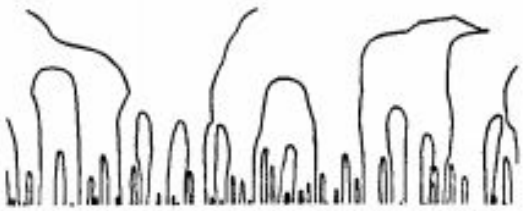


Fig. 2. Position of the edges (zeros of the Laplacian with respect to x) through the linear scale space of Fig. 1 (adapted from Witkin [21]).

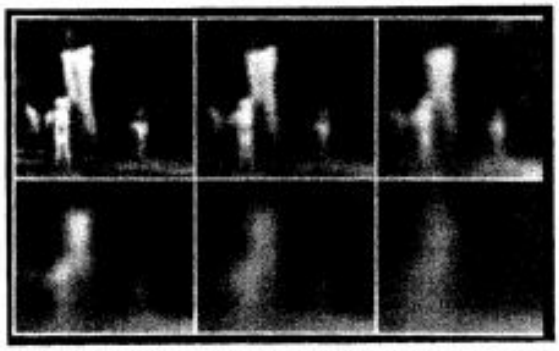


Fig. 3. Scale-space (scale parameter increasing from top to bottom, and from left to right) produced by isotropic linear diffusion (0, 2, 4, 8, 16, 32 iterations of a discrete 8 nearest-neighbor implementation. Compare to Fig. 12.

Consider the anisotropic diffusion equation

$$I_t = div\left(c(x, y, t)\nabla I\right) = c(x, y, t)\Delta I + \nabla c \cdot \nabla I \quad (3)$$

where we indicate with *div* the divergence operator, and with ∇ and Δ respectively the gradient and Laplacian operators, with respect to the space variables. It reduces to the isotropic heat diffusion equation $I_t = c\Delta I$ if $c(x, y, t)$ is a constant. Suppose that at the time (scale) t, we knew the locations of the region boundaries appropriate for that scale. We would want to encourage smoothing *within* a region in preference to smoothing *across* the boundaries. This could be achieved by setting the conduction coefficient to be 1 in the interior of each region and 0 at the boundaries. The blurring would then take place separately in each region with no interaction between regions. The region boundaries would remain sharp.

B. Edge Enhancement

With conventional low-pass filtering and linear diffusion the price paid for eliminating the noise, and for performing scale space, is the blurring of edges. This causes their detection and localization to be difficult. An analysis of this problem is presented in [4].

Edge enhancement and reconstruction of blurry images can be achieved by high-pass filtering or running the diffusion equation backwards in time. This is an ill-posed problem, and gives rise to numerically unstable computational methods, unless the problem is appropriately constrained or reformulated [9].

We will show here that if the conduction coefficient is chosen to be an appropriate function of the image gradient we can make the anisotropic diffusion enhance edges while running *forward* in time, thus enjoying the stability of diffusions which is guaranteed by the maximum principle.

We model an edge as a step function convolved with a Gaussian. Without loss of generality, assume that the edge is aligned with the y axis.

The expression for the divergence operator simplifies to

$$\operatorname{div}\left(c(x, y, t)\nabla I\right) = \frac{\partial}{\partial x}\left(c(x, y, t)I_x\right).$$

We choose c to be a function of the gradient of I: $c(x, y, t) = g(I_x(x, y, t))$ as in (4). Let $\phi(I_x) \doteq g(I_x) \cdot I_x$ denote the flux $c \cdot I_x$.

Then the 1-D version of the diffusion equation (3) becomes

$$I_t = \frac{\partial}{\partial x}\phi(I_x) = \phi'(I_x) \cdot I_{xx}. \tag{5}$$

We are interested in looking at the variation in time of the slope of the edge: $\partial/\partial t(I_x)$. If $c(\cdot) > 0$ the function $I(\cdot)$ is smooth, and the order of differentiation may be inverted:

$$\begin{aligned}\frac{\partial}{\partial t}(I_x) = \frac{\partial}{\partial x}(I_t) &= \frac{\partial}{\partial x}\left(\frac{\partial}{\partial x}\ \phi(I_x)\right) \\ &= \phi'' \cdot I_{xx}^2 + \phi' \cdot I_{xxx}.\end{aligned} \tag{6}$$

Proof of the Maximum Principle

Call $\boldsymbol{A}$ an open bounded set of $\mathbb{R}^n$ (in our case $\boldsymbol{A}$ is the plane of the image, a rectangle of $\mathbb{R}^2$), and $\boldsymbol{T} = (a, b)$ an interval of $\mathbb{R}$. Let $\boldsymbol{D}$ be the open cylinder of $\mathbb{R}^{n+1}$ formed by the product $\boldsymbol{D} = \boldsymbol{A} \times \boldsymbol{T} = \{(x, t) : x \in \boldsymbol{A}, t \in \boldsymbol{T}\}$. Call $\partial \boldsymbol{D}$ the boundary of $\boldsymbol{D}$, $\overline{\boldsymbol{D}}$ its closure, and $\partial_T \boldsymbol{D}$, $\partial_S \boldsymbol{D}$, and $\partial_B \boldsymbol{D}$ the top, side, and bottom portions of $\partial \boldsymbol{D}$:

$$\partial_T \boldsymbol{D} = \{(x, t) : x \in \boldsymbol{A}, t = a\}$$

$$\partial_S \boldsymbol{D} = \{(x, t) : x \in \partial \boldsymbol{A}, t \in \boldsymbol{T}\}$$

$$\partial_B \boldsymbol{D} = \{(x, t) : x \in \boldsymbol{A}, t = b\}$$

and, for convenience, call $\partial_{SB} \boldsymbol{D}$ the side-bottom boundary:

$$\partial_{SB} \boldsymbol{D} = \partial_S \boldsymbol{D} \cup \partial_B \boldsymbol{D}.$$

The following theorems hold.

Theorem: Consider a function $f : \mathbb{R}^{n+1} \to \mathbb{R}$ that is continuous on $\overline{\boldsymbol{D}}$, and twice differentiable on $\boldsymbol{D} \cup \partial_T \boldsymbol{D}$. If f satisfies the differential inequality

$$C(x, t) f_t - c(x, t) \triangle f - \nabla c \cdot \nabla f \leq 0 \qquad (20)$$

on $\boldsymbol{D}$, with $C : \mathbb{R}^{n+1} \to \mathbb{R}^+$ continuous on $\overline{\boldsymbol{D}}$, and differentiable on $\boldsymbol{D} \cup \partial_T \boldsymbol{D}$, then it obeys the maximum principle, i.e., the maximum of f in $\overline{\boldsymbol{D}}$ is reached on the bottom-side boundary $\partial_{SB} \boldsymbol{D}$ of $\boldsymbol{D}$:

$$\max_{\overline{\boldsymbol{D}}} f = \max_{\partial_{SB} \boldsymbol{D}} f.$$

Procedure

According to the paper by Jitendra Malik and Pietro Perona published in 1990 (http://image.diku.dk/imagecanon/material/PeronaMalik1990.pdf), a python function using the **NumPy** library. was developed that smoothes images for noise reduction.

```
import numpy as np
```

```
def
PeronaMalik_Smoother(image,K,LAMBDA,gfunction,nIterations,convert_to_grays
cale=True):
```

The edges are preserved, and this can be considered as the first step toward image segmentation.

This method preserves edges while smoothing non-edge regions. Inter-region blurring is minimized while intra-region blurring is encouraged.

Variables:

- **image**: the input image
- **K**: parameter for diffusion sensitivity. Could set this manually, or determine by Canny noise estimator histogram.
- **LAMBDA**: parameter for diffusion sensitivity. Should be 0 < LAMBDA < 1/4
- **gfunction**: 'Exponential' or 'Cauchy'. The function to use for calculating diffusion coefficients.

These 2 functions are used in the original paper:

- **nIterations**: number of iterations to perform (more iterations -> more blurring). Images of all iterations are saved in an array for viewing.
- **convert_to_grayscale**: True or False. If True, converts a colour image to grayscale. If False, applies diffusion independently to each colour channel.

We print out the shape so the user can decide if it has the right number of colour channels, etc. For instance, to let the user know if there are 4 channels (if not expecting alpha).

```
print ('Image shape: ', image.shape)
```

We can optionally convert multichannel image to grayscale (if RGBA, will also average over A, so this is why the shape is printed above). If the image is already 2 dimensional, is already assumed grayscale, so warn the user.

```
if convert_to_grayscale == True:
```

```
        try:
            image = np.mean(image,axis=2)
        except IndexError:
            raise IndexError('Image is already 2D, so is assumed grayscale
already. Check shape.')
```

Get the number of colour channels of image [1 for grayscale, 3 for RGB, 4 for RGBA]. Assuming the image shape is height x width for grayscale or height x width x Nchannels for colour. But not expecting more than 3 dimensions.

```
 nChannels = 1 if image.ndim == 2 else image.shape[2]
    print ('nChannels',nChannels)
```

In the case of a grayscale image, to make things easier later, just make the grayscale image have the 3rd axis of length 1.

```
if nChannels == 1:
        image = np.expand_dims(image,axis=2)
```

4D Container array of all iterations of image diffusion

```
image_stack = np.expand_dims(image,axis=0)
```

Do **nIterations** of diffusion

```
 for i in range(nIterations):
        if i % 10 == 0:
            print ('Starting iteration {0} of {1}'.format(i,nIterations))
        image_t = np.zeros(image.shape)
        for channel in range(nChannels):
            temp = image_stack[-1][:,:,channel]
```

Following equation 8 in the paper: calculate nearest neighbour differences to an approximate gradient of image intensity.

```
            vert_diff = np.diff(temp,axis=0)
```

```
            horiz_diff = np.diff(temp,axis=1)
            nanrow =
np.expand_dims(np.nan*np.ones(vert_diff.shape[1]),axis=0)
            nancol =
np.expand_dims(np.nan*np.ones(horiz_diff.shape[0]),axis=0).T
            grad_S = np.vstack((vert_diff,nanrow))
#NaN on bottom row
            grad_N = np.vstack((nanrow,-vert_diff))
#NaN on top row, and negated diffs since going opposite direction from
np.diff() default
            grad_E = np.hstack((horiz_diff,nancol))
#NaN on right column
            grad_W = np.hstack((nancol,-horiz_diff))
#NaN on left column, and negated diffs since going opposite direction from
np.diff() default
```

Following equation 10 in the paper: calculate conduction coefficients

Technically, the coefficients should be more appropriately be evaluated at the halfway point between pixels, not at the pixels themselves.

But this is more complicated for approximately the same results (according to authors). So use the same values for gradients as above.

```
if gfunction == 'Exponential':
                c_S = np.exp(-(grad_S/K)**2)
                c_N = np.exp(-(grad_N/K)**2)
                c_E = np.exp(-(grad_E/K)**2)
                c_W = np.exp(-(grad_W/K)**2)
```

```
            if gfunction == 'Cauchy':
                c_S = 1./(1.+(grad_S/K)**2)
                c_N = 1./(1.+(grad_N/K)**2)
                c_E = 1./(1.+(grad_E/K)**2)
                c_W = 1./(1.+(grad_W/K)**2)

            #Examine the conduction coefficients:
            if i == 7:
                plt.figure()
                plt.title('c_S Diffusion Coefficients\nChannel
{}'.format(channel),fontsize=30)
                plt.tick_params(labelsize=0)
                plt.imshow(c_S,interpolation='none')
                cb=plt.colorbar()
                cb.ax.tick_params(labelsize=20)
                plt.show()
```

Following equation 7 in the paper: Update the image using the diffusion equation:

```
temp2 = temp + LAMBDA*(c_S*grad_S + c_N*grad_N + c_E*grad_E + c_W*grad_W)
```

Reset boundaries since the paper uses adiabatic boundary conditions and above steps intentionally set boundaries to NaNs.

```
 temp2[:,0] = temp[:,0] #Left edge
            temp2[:,-1] = temp[:,-1] #Right edge
            temp2[-1,:] = temp[-1,:] #Bottom edge
            temp2[0,:] = temp[0,:] #Top edge
```

Update this channel of the image at this time step

```
image_t[:,:,channel] = temp2


        image_t = np.expand_dims(image_t,axis=0)
        image_stack = np.append(image_stack,image_t,axis=0)
```

The image_stack is stack of all iterations.

Iteration 0 is original image, iteration -1 is final image.

Intermediate images are also returned for visualization and diagnostics

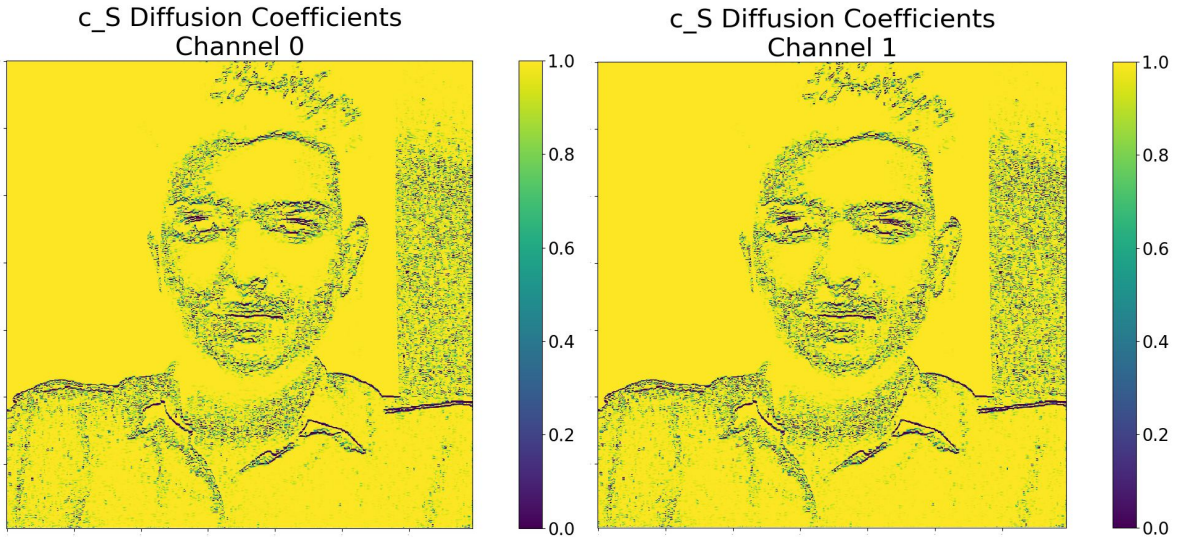
```
return image_stack
```

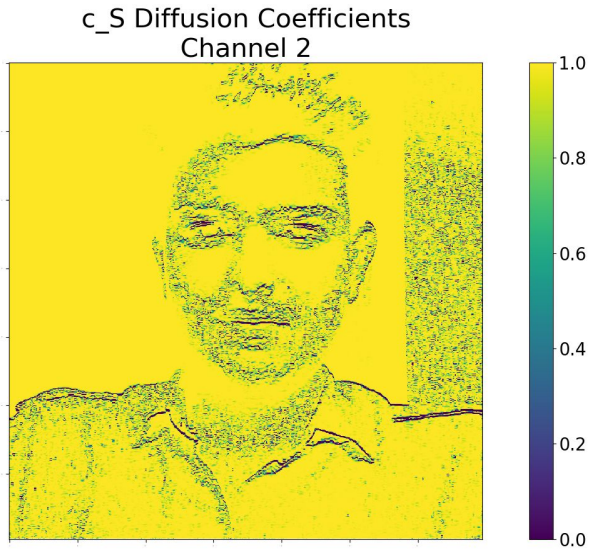
Now, in the main function:

```
if __name__ == '__main__':
    import matplotlib.pyplot as plt
    import matplotlib.image as mpimg
    #Load test image
    image =
mpimg.imread('C:/Users/Krishanu/Desktop/MAT2002Perona-Malik/Capture.png')#
[300:800,200:1000,:3]#[300:800,200:300,:3]#
    print (image.shape)


    #Set algorithm parameters
    K = .1
    LAMBDA = .15
    nIterations = 25#100
    gfunction = 'Exponential' #'Cauchy'


    #Plot grayscale example
    PMimage_stack =
PeronaMalik_Smoother(image,K,LAMBDA,gfunction,nIterations,convert_to_grays
cale=True)
```

```
    fig=plt.figure()
    fig.add_subplot(121)
    plt.title('Original',fontsize=30)

plt.imshow(np.squeeze(PMimage_stack[0]),interpolation='None',cmap='copper'
)
    plt.tick_params(labelsize=0)
    fig.add_subplot(122)
    plt.title('After {} Iterations'.format(nIterations),fontsize=30)

plt.imshow(np.squeeze(PMimage_stack[-1]),interpolation='None',cmap='copper
')
    plt.tick_params(labelsize=0)
    plt.show()
    #Plot colour example
    PMimage_stack_color =
PeronaMalik_Smoother(image,K,LAMBDA,gfunction,nIterations,convert_to_grays
cale=False)
    fig=plt.figure()
    fig.add_subplot(121)
    plt.title('Original',fontsize=30)
    plt.imshow(PMimage_stack_color[0],interpolation='None')
    plt.tick_params(labelsize=0)
    fig.add_subplot(122)
    plt.title('After {} Iterations'.format(nIterations),fontsize=30)
    plt.imshow(PMimage_stack_color[-1],interpolation='None')
    plt.tick_params(labelsize=0)
    plt.show()
```

Result


c_S Diffusion Coefficients
Channel 0
1.0
0.8
0.6
0.4
0.2
0.0
c_S Diffusion Coefficients
Channel 1
1.0
0.8
0.6
0.4
0.2
0.0


c_S Diffusion Coefficients
Channel 2
1.0
0.8
0.6
0.4
0.2
0.0

Original

After 50 Iterations

Original

After 20 Iterations

References

http://image.diku.dk/imagecanon/material/PeronaMalik1990.pdf

https://github.com/InsightSoftwareConsortium/ITKAnisotropicDiffusionLBR

https://github.com/tombena/image-denoising

https://www.youtube.com/watch?v=pEM--aR53vY

https://www.youtube.com/watch?v=zJNdajktguA

https://www.youtube.com/watch?v=IzeJlBomZ_s

https://www.youtube.com/watch?v=CdWReOzhc-g

https://github.com/loli/medpy