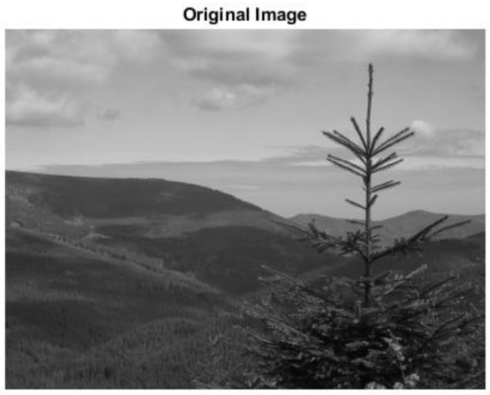
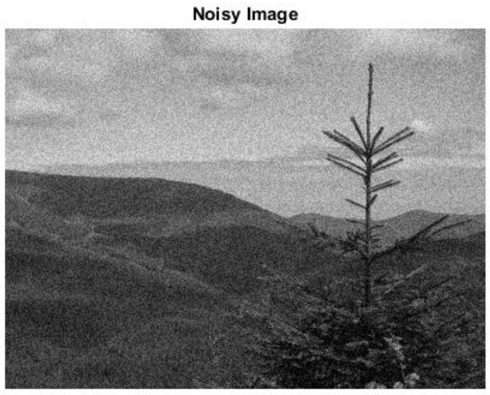
Image Denoising

1 Introduction

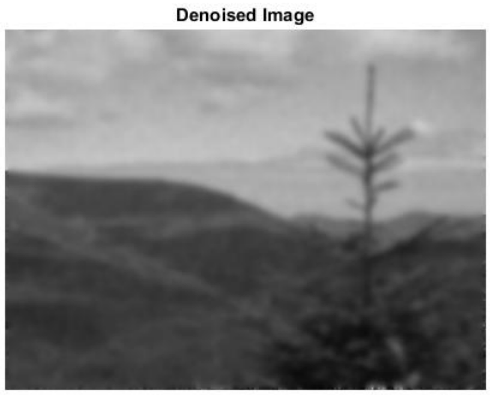
Image denoising is part of the process used when performing image restoration or alteration. The goal of image denoising is to remove noise from the image in such a way that the original image can be more clearly seen. This involves smoothing out the image; getting rid of what is called image noise. Noise is corruption of the image where certain pixels seem out of place, and makes an image appear grainy. To illustrate the purpose of image denoising, let me give an example scenario of when it might be necessary. Let's say that someone accidentally deleted their pictures from their hard drive, and wanted to restore them, but in the process some of the image files got corrupted. Originally one of the pictures on the hard drive looked like this:



After deletion and partial corruption, the file looked like this:



In order to restore the image back to the original, the first step is to get rid of corruption noise and smooth the image. After the denoising, the image would look something like this:



Then further steps would be performed in order to restore the image to get it as close as possible to the original. As you can see, this process does blur the image. This phenomenon is known as Gaussian blur and the amount of blurring is affected by the picture as well as certain parameters.

This smoothing step is important and can be achieved in many ways. One of these ways is the Perona-Malik method [1]. This method uses a nonlinear denoising algorithm which can be achieved by modeling or solving the nonlinear diffusion equation for heat. Consider a noisy image u with pixel values referenced by $u(i, j)$. Then denoising can be achieved by solving for the following nonlinear diffusion equation:

$$\frac{\delta u}{\delta t} = \nabla \cdot (g(\nabla u)\nabla u) \tag{1}$$

where

$$g(\nabla u) = \frac{v}{1 + \frac{||\nabla u||}{K}} \tag{2}$$

represents the diffusion coefficient. We can see that if $g(\nabla u)$ is constant, the equation turns into the linear diffusion equation for heat. The values v and K are constants that affect the diffusion. $v > 0$ affects the amount of diffusion and $K > 0$ controls how sharp the image boundaries are. These two constants affect the magnitude of Gaussian blur significantly. I have implemented my own method for solving this system numerically and I will be presenting it below.

2 Algorithms

2.1 Discretization and Numerical Solution to the Equation

The subscripts (i, j) used below indicate coordinates that specify a pixel. Because our space for diffusion is not real space, but actually separate pixels, Δx and Δy can be set to 1. Setting them both equal to 1 allows for easier math and for every pixel to be equidistant, essentially centering them. This allows the diffusion to be even with respect to the pixels around a pixel, meaning that the pixel $(i, j+1)$ does not have a greater effect on (i, j) than $(i+1, j)$ does. I used the FTCS scheme presented below to obtain my numerical approximation to the system. The FTCS method is a finite difference method used for numerically solving the heat equation. It is a first order method in time, and remains stable for my purposes when the time step is small enough. The FTCS scheme is also an explicit method, allowing us to compute u^{n+1} using values of the previous time step, u^n, with no need to solve a linear system, which would be very large in this case. The bounds of our space are given by the amount of pixels the image contains in each direction. Because in this case, I am working with the original image, I set my boundary conditions, the outer edges, to equal the original picture at every step. Without the original it would be necessary to create an algorithm or just assign a frame as the boundary conditions. Below is how I derived my equation for u^{n+1} in terms of u.

Start with equation [1]. The function $g(\nabla u)$ produces a scalar (the diffusion coefficient for a given step) so it can be brought to the outside.

$$\frac{\delta u}{\delta t} = g(\nabla u)\nabla \cdot \nabla u$$

We then obtain the equation:

$$\frac{\delta u}{\delta t} = g(\nabla u)\nabla^2 u = g(\nabla u)\triangle u$$

where $\triangle$ is the Laplace operator. After performing the Laplace operator we obtain the diffusion equation of heat with an altering diffusion constant $g(\nabla u)$:

$$\frac{\delta u}{\delta t} = g(\nabla u)(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}) \tag{3}$$

After performing the gradient operator u we obtain:

$$\frac{\delta u}{\delta t} = g(\frac{\delta u}{\delta x}\hat{\mathbf{i}} + \frac{\delta u}{\delta y}\hat{\mathbf{j}})(\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2})$$

We can now discretize the system. I used an FTCS (Forward-Time Central-Space) scheme. The forward time step approximation for change in u over time:

$$\frac{\delta u}{\delta t} = \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t}$$

The central difference approximations for the spacial dimensions:

$$\frac{\delta u}{\delta x}\hat{\mathbf{i}} + \frac{\delta u}{\delta y}\hat{\mathbf{j}} = \frac{u^n_{i+1,j} - u^n_{i-1,j}}{2\Delta x}\hat{\mathbf{i}} + \frac{u^n_{i,j+1} - u^n_{i-1,j}}{2\Delta y}\hat{\mathbf{j}}$$

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = \frac{-u^n_{i+2,j} + 16u^n_{i+1,j} - 30u^n_{i,j} + 16u^n_{i-1,j} - u^n_{i-2,j}}{12(\Delta x)^2} + \frac{-u^n_{i,j+2} + 16u^n_{i,j+1} - 30u^n_{i,j} + 16u^n_{i,j-1} - u^n_{i,j-2}}{12(\Delta y)^2}$$

Overall equation:

$$\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} = g(\frac{u^n_{i+1,j} - u^n_{i-1,j}}{2\Delta x}\hat{\mathbf{i}} + \frac{u^n_{i,j+1} - u^n_{i-1,j}}{2\Delta y}\hat{\mathbf{j}})$$

$$(\frac{-u^n_{i+2,j} + 16u^n_{i+1,j} - 30u^n_{i,j} + 16u^n_{i-1,j} - u^n_{i-2,j}}{12(\Delta x)^2} + \frac{-u^n_{i,j+2} + 16u^n_{i,j+1} - 30u^n_{i,j} + 16u^n_{i,j-1} - u^n_{i,j-2}}{12(\Delta y)^2})$$

We can now deduce an equation for finding u^{n+1} as a function of u^n, where u^n is the processed image at time t^n and u^{n+1} is the processed image at time t^{n+1}:

$$u^{n+1}_{i,j} = u^n_{i,j} + \Delta t(g(\frac{u^n_{i+1,j} - u^n_{i-1,j}}{2\Delta x}\hat{\mathbf{i}} + \frac{u^n_{i,j+1} - u^n_{i-1,j}}{2\Delta y}\hat{\mathbf{j}})$$
$$(\frac{-u^n_{i+2,j} + 16u^n_{i+1,j} - 30u^n_{i,j} + 16u^n_{i-1,j} - u^n_{i-2,j}}{12(\Delta x)^2} + \frac{-u^n_{i,j+2} + 16u^n_{i,j+1} - 30u^n_{i,j} + 16u^n_{i,j-1} - u^n_{i,j-2}}{12(\Delta y)^2})) \tag{4}$$

2.2 My Pseudocode

My code uses the math presented above. Specifically, my code follows these steps:

1. Establish the domain through the variables 0, n, 0, m. Where the picture is an mxn matrix of grayscale double values. n represents the x direction and m represents the y direction.

2. Discretise the x and y dimensions and establish variables dx and dy which correspond to Δx and Δy. Set these variables both equal to 1.

3. Establish a time step and simulation length which are defined by the caller

4. Add noise to the given picture by adding random values to every pixel. This slightly lightens or darkens every pixel, but does not drastically deteriorate the image.

5. Set up initial conditions, the noisy picture.

6. Perform this loop until t_{final} is reached to denoise the image

   - Calculate for the boundary conditions (Set edges equal to original picture edges)

- Calculate ∇u for every pixel by setting up two mxn matrices that represent the gradient, one for the x-dimension and one for the y-dimension.
- Update $g(\nabla u)$ and store the temporary diffusion constant for every grid point in an mxn matrix g.
- Update the image using a double for loop to iterate through every pixel. Equation [4] is used to update each individual pixel.
- Update the time

7. Display the original image, the noisy image, and the smoothed image.

3 Results

In `MatLab`, my simulation code is invoked at the prompt by calling the function

`denoise( v , k, timeStep, simLength, pic )`

The function has 5 parameters. `v` and `k` correspond to the constants that are needed by Equation [2] and described in the introduction. They change how the image looks after the denoising. `timeStep` represents Δt and should not be greater than .15 in order to maintain stability. The smaller Δt, the more accurate the diffusion will be. `simLength` determines how many iterations the denoising will go on for, where the number of iterations is equal to `simLength` over `timeStep`. `pic` is a string that specifies what image should be processed. The image file specified by the string should be in the working directory and include the file extension.

`denoise( 1, .9, .01, 3, 'Super_Heros.jpg')`

will denoise the picture of superman provided by the professor by processing the image 300 times with a Δt value of .01. Also try `denoise( 1, .9, .01, 3, 'landscape.jpg')` and `denoise( 1, .9, .01, 3, 'images.jpg')`

Here are some images that have been processed using the specified parameters:

Original Image

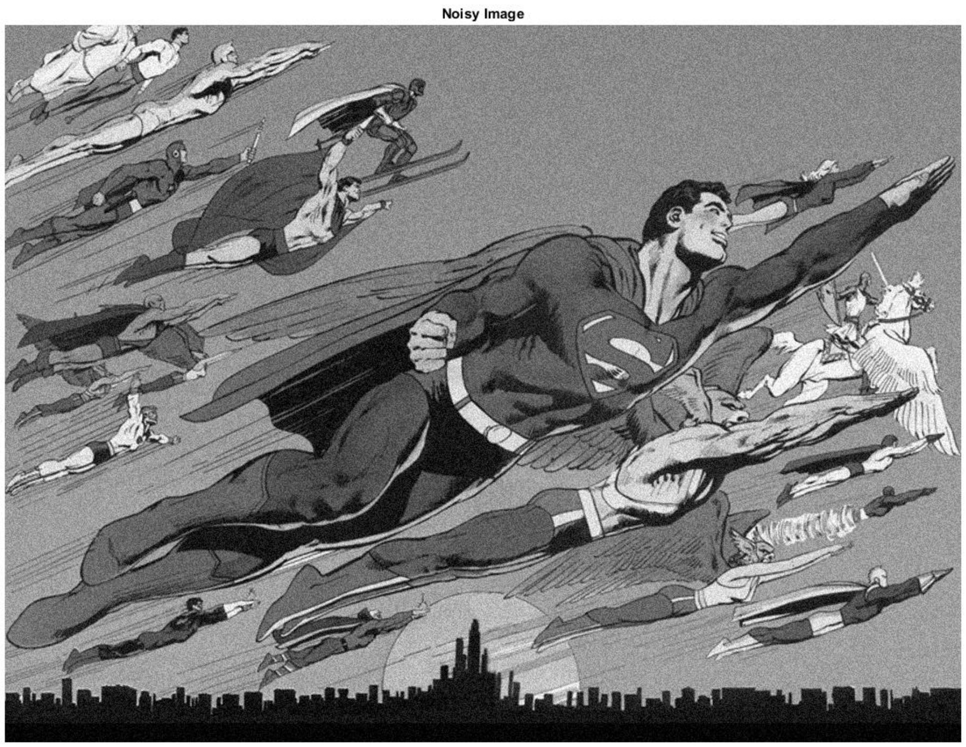


Figure 1: v=.25, k=15, timeStep=.01, simLength=3

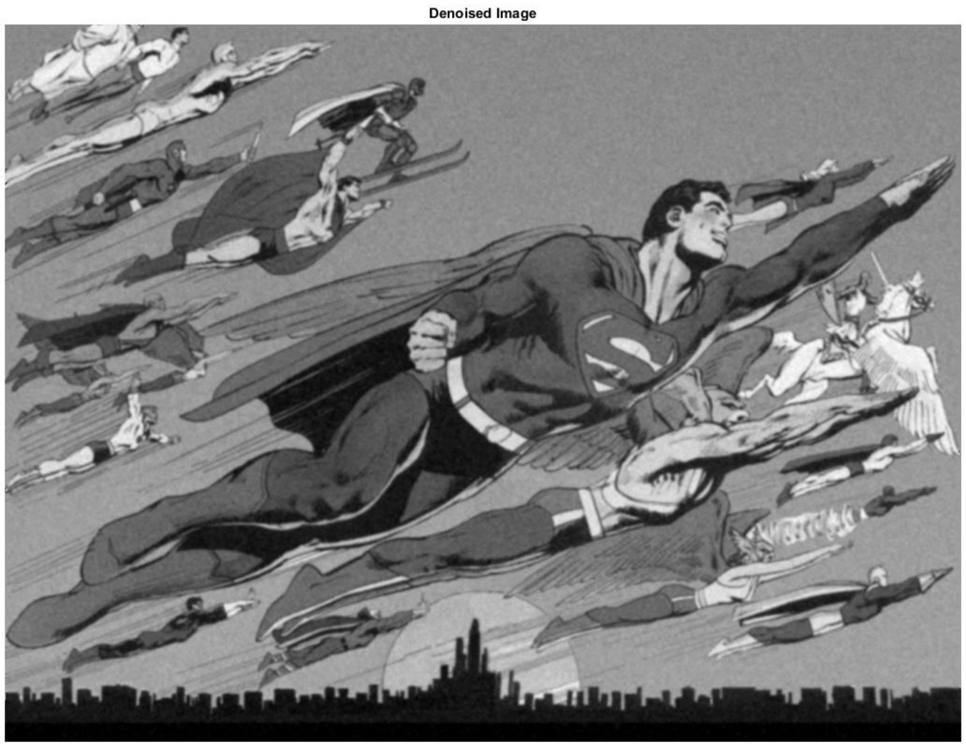


Figure 2: v=.25, k=15, timeStep=.01, simLength=3

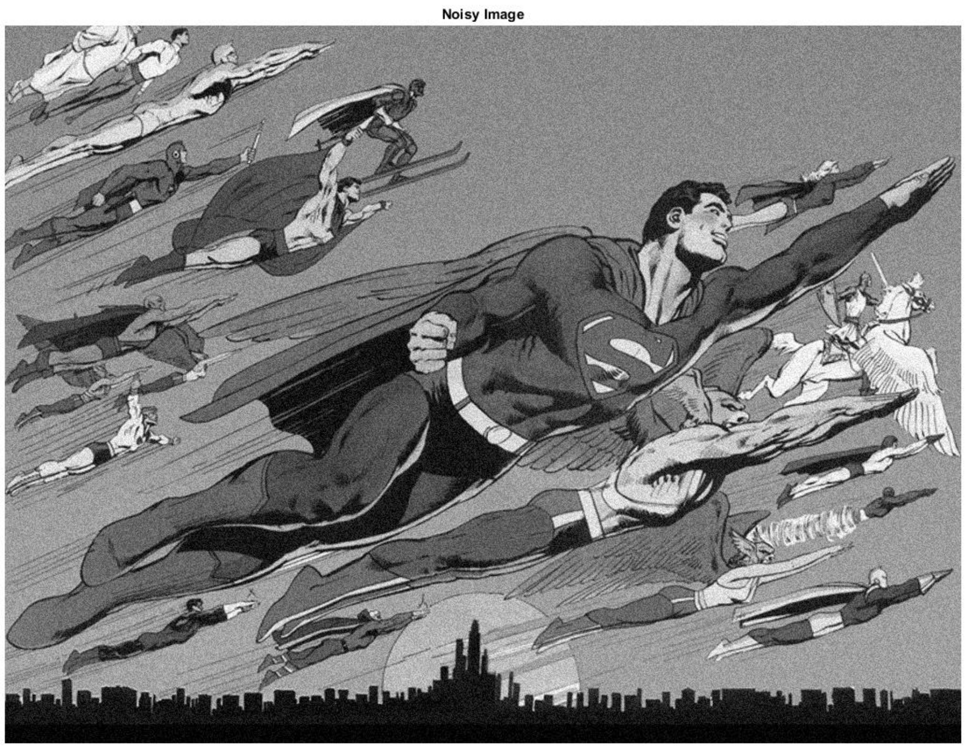


Figure 3: v=1, k=3, timeStep=.01, simLength=3

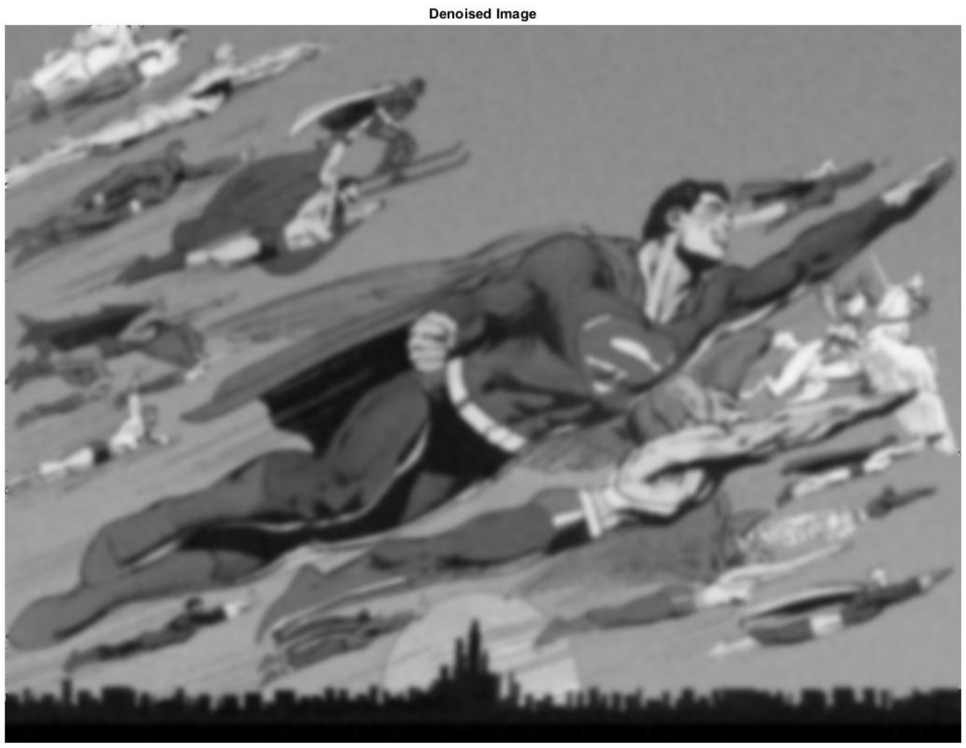
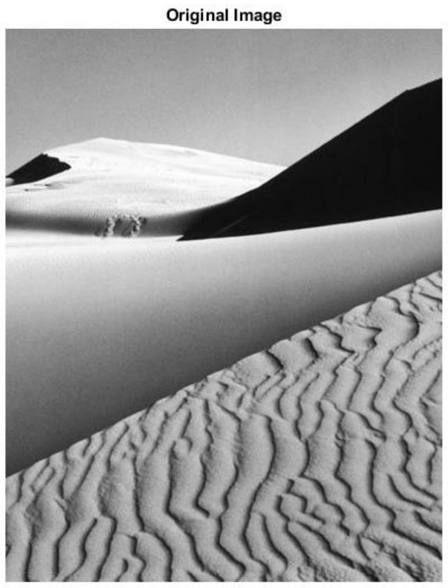


Figure 4: v=1, k=3, timeStep=.01, simLength=3



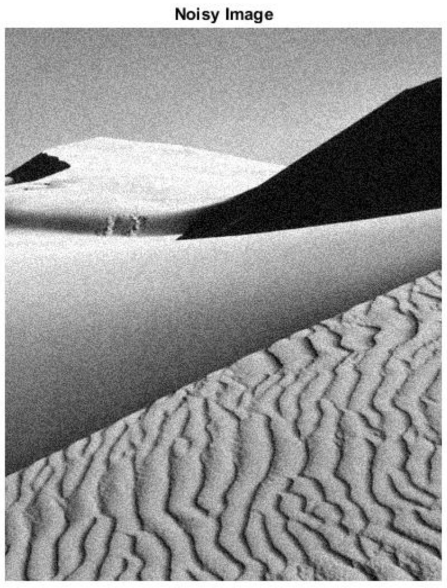


Figure 5: v=1, k=.3, timeStep=.01, simLength=5

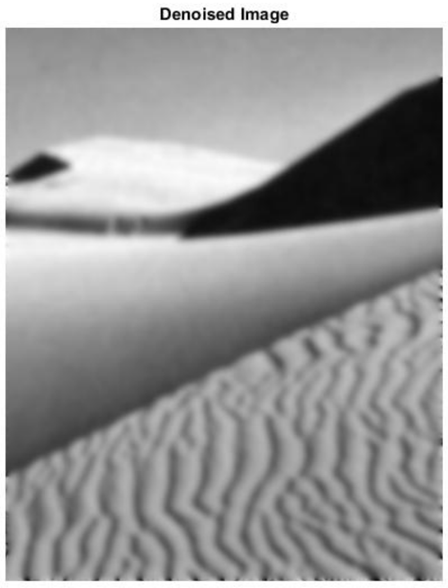


Figure 6: v=1, k=.3, timeStep=.01, simLength=5

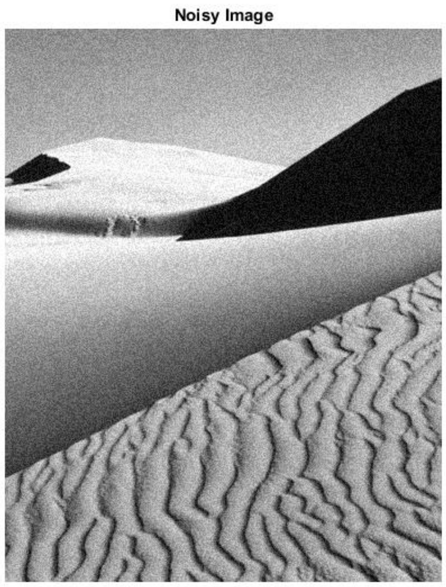


Figure 7: v=.15, k=.07, timeStep=.001, simLength=10

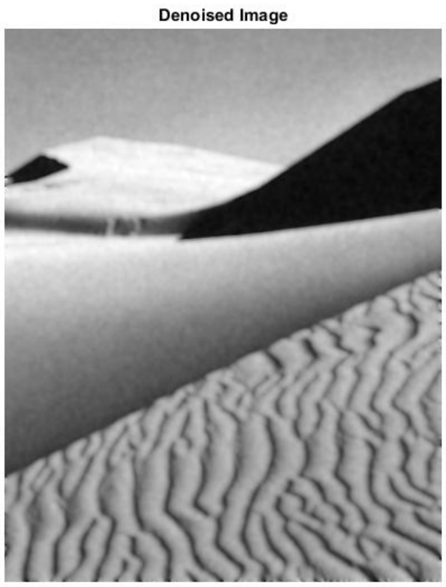


Figure 8: v=.15, k=.07, timeStep=.001, simLength=10

4 Conclusion

My implementation works. There are only two areas which need to be improved upon. One is the boundary conditions of the system. I simply used the boundaries of the original image, however in a real world application where the uncorrupted image is unobtainable, an algorithm is needed to calculate for the edges. One could either use a fixed border, or something more complicated. Secondly, I left the constants `v` and `k` needed for $g(\nabla u)$ open to be specified by a caller because I do not know how to optimize these values for a given image. According to Perona and Malik [1] there is a way to optimize `k` by using the "noise estimater" described by Canny. This updates it at every iteration of denoising. Perona and Malik also choose to hold `v` at a value of 1, but I left it up to the user.

There are different ways to solve the differential equation used in this system. The method presented by Perona and Malik [1] is a good one. They use a k-nearest Neighbor implementation. However I chose to use the FTCS scheme in order to compare my results to the k-nearest Neighbor method. I could not tell the difference between my denoised image and a k-nearest Neighbor denoised image for most cases I tested. I do not present these comparisons here because as I said I could not see any significant difference.

When running my code, I recommend using a time step of .01 and a simulation length of 3. This allows for accurate diffusion, and takes less than 20 seconds to compute. Adjusting the values of `v` and `k` will affect the denoised image, as can be seen in the Results section.

A Implementation in MatLab

The `MatLab` implementation with comments:

```
denoise.m

function denoise( v , k, timeStep, simLength, pic )

    I=imread(pic);% Load image file and store it as variable I.

    I=rgb2gray(I); % Convert to gray scale
    I=im2double(I); % Convert the variable into double.

    m=size(I,1);
    n=size(I,2);

    %Center pixel distances
    dy = 1;
    dx = 1;

    % Add some noise to the image
    I_noisy=I;
    for i=1:m
        for j=1:n
            I_noisy(i,j)=I(i,j)+((rand(1))-.5)/5;
        end
    end

    % Process the noisy image ...

    t=0;%Time variable to keep track of iterations
    orig = I;%Keep a copy of the original image for boundary conditions
    I = I_noisy;%Make the Image the noisy image

    %PPERFORM DENOISING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    while t< simLength

        %%%%%%BOUNDARY CONDITIOS = ORIGINAL IMAGE%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        for y=1:m
            I(y,1) = orig(y,1);
            I(y,2) = orig(y,2);
        end
        for y=1:m
            I(y,n) = orig(y,n);
            I(y,n-1) = orig(y,n-1);
        end
        for x=1:n
            I(1,x) = orig(1,x);
            I(2,x) = orig(2,x);
            %I(3,x) = orig(3,x);
            %I(4,x) = orig(4,x);
                %I(5,x) = orig(5,x);
            %I(6,x) = orig(6,x);
        end
        for x=1:n
            I(m,x) = orig(m,x);
            I(m-1,x) = orig(m-1,x);
            %I(m-2,x) = orig(m-2,x);
            %I(m-3,x) = orig(m-3,x);
            %I(m-4,x) = orig(m-4,x);
            %I(m-5,x) = orig(m-5,x);
        end
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %First derivative approximations of the iamge brightness with respect to
        %   x and y
        dI_dx = (circshift(I,-1,2) - circshift(I,1,2))/2*dx;
        dI_dy = (circshift(I,1,1) - circshift(I,-1,1))/2*dy;

        %Matrix of conduction coefficients updated at every iteration as a
        %   function of the brightness gradient
        g = v./(1+(dI_dx.^2+dI_dy.^2)/k);
        g = g*timeStep;


        temp=I;%Holds I of previous time step
        temp2=I;%Holds I of previous time step
        temp = g.*temp;%Now contains values of the previous time step times
        %their respective conduction coeeficients to make calculations simpler

        %Updates I at every grid point
        for x=3:n-2
            for y=3:m-2
                I(y,x) = temp2(y,x) + (-temp(y,x-2) + 16*temp(y,x-1)...
                        - 30*temp(y,x) + 16*temp(y,x+1) - temp(y,x+2)...
```

```
                        - temp(y-2,x) + 16*temp(y-1,x) - 30*temp(y,x)...
                        + 16*temp(y+1,x) - temp(y+2,x))/12;
            end
        end

        t=t+timeStep;
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Display original image
    figure;
    imshow(orig);
    title('Original Image');

    %Display noisy image
    figure;
    imshow(I_noisy);
    title('Noisy Image');

    %Display Denoised Image
    figure;
    imshow(I);
    title('Denoised Image');

end
```

References

[1] Pietro Perona and Jitendra Malik. *Scale-Space and Edge Detection Using Anisotropic Diffusion.* Proceedings of IEEE Computer Society Workshop on Computer Vision, pages 16–22, 1987.