Homework 3: Image Filtering

Karl Marrett (kdmarrett@gmail.com)

Abstract

In this report, we use two common methods to clean images that are corrupted either globally or locally with noise. The first technique uses filtering by implementing the Fast Fourier Transform (FFT) to decompose the image into the spectral domain. Most image information is contained at low frequencies and higher frequencies tend to be noise, thus a low pass filter can significantly improve the quality of an image. A second technique to denoise images uses the diffusion equation to blur images, much in the same way heat diffuses in objects. The goals of this report are to:

1. Use filtering to denoise black and white and RGB images that have been corrupted with white noise
2. Use diffusion to locally denoise areas of images

1 Introduction and Overview

To practice our skills of filtering, we use four images of Derek Zoolander that have been corrupted, some by white noise, others by a local spot of high noise. The first image with high global noise is shown in Figure 1. Unfortunately, due to time constraints, this entire section can not be devoted to describing the plot of the movie Zoolander. In lieu, we will begin the summary of our proposed methods. To clean the images with these specific problems, we will be using the techniques of diffusion and linear filtering. First, we will analyze the two methods analytically, seeing their similarities mathematically. Next we will implement them in the programming language MATLAB. With our implementation, we will weigh these techniques by overfiltering, underfiltering and locally filtering our noisy images. Finally, we will discuss our techniques by evaluating the denoised images produced.

2 Theoretical Background

2.1 Linear Filtering

If an image is noisy it appears as having high contrast, meaning on average pixels have a large discrepancy in value with their neighbors. Since we can think about our colored pixels of images simply as integers, we can also think about a noisy image as a matrix with high

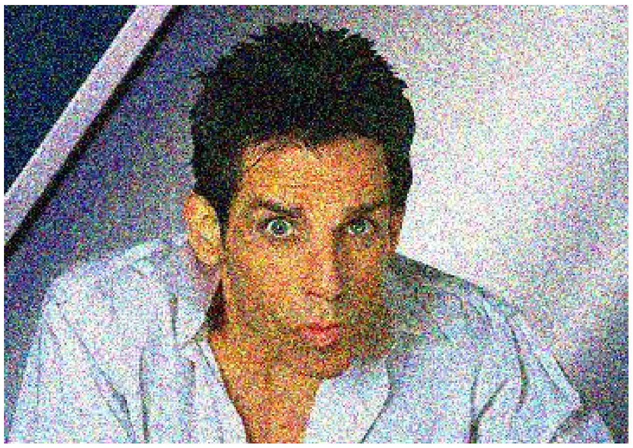

Figure 1: High global noise color image of Derek

variance and a blurry image as one with low variance between the elements. When we look for shapes in images we tend to look at shapes or structures that span significant parts of the images. These parts of the image that we care about, if transformed into the Fourier domain, would be low frequency components. Similarly, noise in an image, since it is likely fluctuctations from one pixel to the next, would appear as high frequencies in the Fourier domain. In order to take advantage of this fact we need to transform our data into this representation of a set of periodic functions. The Fourier transform does this by transforming a vector from, for example, the space domain, into a sum of sines and cosines as shown in Equation 1.[1]

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos nx + b_n \sin nx) \qquad x \in (-\pi, \pi] \tag{1}$$

Similarly, the Fourier transform kernel can also be in the imaginary domain in continuous space as shown in Equation 2. This is closer to the implementation that MATLAB uses.

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \tag{2}$$

Since this analysis is done with a computer, we will be implementing the fast Fourier transform (FFT), a discrete form of Fourier analysis. The FFT algorithm has a broad set of applications and implications. In the technique of linear filtering we take advantage of the fact that noise tends to appear in the higher frequencies by multiplying our FFT data u by the 2-dimensional Gaussian shown in Equation 3:

$$F(k_x, k_y) = e^{-D((k_x-a)^2+(k_x-b)^2)} \tag{3}$$

where D represents the width of the Gaussian (assumed to be the same for x and y dimensions in this case) and where a and b represent the shift across the x and y axes respectively.

2.2 Diffusion Filtering

One might intuit that another way to smooth high noise data would be to apply the heat equation. After all, we are just taking a data matrix that has high contrast between points and averaging points in a local spatial location. This would describe diffusion processes such as the dissipation of heat governed by the equation:

$$u_t = D\nabla^2 u \tag{4}$$

where $\nabla^2 = \partial_x^2 + \partial_y^2$, and D is simply our diffusion coefficient.[1] For our purposes u will be our pixel space which is a function of x and y. If we assume periodic boundary conditions then the analytical solution to the equation is simply:

$$\hat{u}_t = -D(k\partial_x^2 + \partial_y^2)\hat{u} \qquad \rightarrow \qquad \hat{u}_t = \hat{u}_0 e^{-D(k\partial_x^2+\partial_y^2)t} \tag{5}$$

where $\hat{u}$ is the FFT of u and $\hat{u}_0$ is the initial conditions i.e. our original image matrix. By examining this function we can notice that it is the same general solution of our linear filtering method, namely it is a Gaussian filter applied in the Fourier domain to our data. Despite these similarities, formalizing the filtering process in terms of the heat equation has several advantages over the linear filtering method. In the example above, D was simply a coefficient, but in more advanced filtering schemes, D could be dependent on x and y, if for example we wanted to diffuse a specific part of an image. By applying the diffusion equation to the local areas of rash on Derek's latter pictures, we can essentially build a D coefficient that is x, y dependent.

3 Algorithm Implementation and Development

By default image data comes in *uint*8 format. Also, color images are loaded as a separate 3rd dimension (Red value, Green value, Blue value). In other words, image data of pixels is a two dimensional matrix for gray pictures, and three dimensional for color pictures. This will clarify most for loops within the code.

3.1 Linear Filtering

In order to have a modular way to filter images I created the function *imageflt* this function takes a raw *uint*8 image and processes each dimension separately depending on whether it is a color or grayscale image. This is performed by taking the FFT of the image data shifting it then applying a Gaussian filter. This data is then inverse shifted, inverse Fourier transformed, reconverted back into *uint*8 format and placed back into the dimension of the original data matrix if it is in color. If the image is in gray (i.e. 2 dimensional) this is performed only once. To choose the final filtering strength for the images, a variety of strengths were chosen. From these, the one that balanced some degree of noise filtering without significantly blurring the image was chosen.

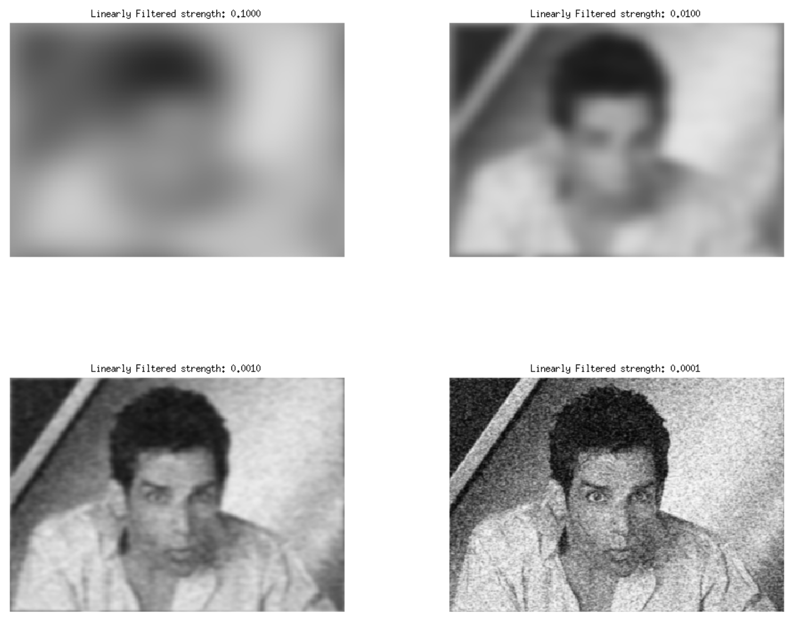
Linearly Filtered strength: 0.1000
Linearly Filtered strength: 0.0100
Linearly Filtered strength: 0.0010
Linearly Filtered strength: 0.0001

Figure 2: The globally linearly filtered images of Derek in black and white. We can see that the strength of the Gaussian filter is important as a strong filter cuts out too much of the high frequency information thus blurring the image while too weak of a filter leaves the image with the high contrast noise.

3.2 Diffusion Filtering

A function similar in design to $imageflt$ was created to diffuse the images. This function similarly took the raw image files in $uint8$ format. As we have discussed however, the diffusion equation relies on two parameters. One a time constant, denoted tau in the code, and the other the coefficient D. In order to see the effects of over and under diffusing images, I chose various tau factors to evolve the image through.

The image diffusion function $imagedfs$ relies on several deeper functions. In order to implement our operator ∇^2 we will create a Laplacian operator that takes an estimate of the double partial derivatives of the x and y coordinates. This operator, L is created in the function $buildL$. It is a $x*y$ by $x*y$ sparse matrix with -2 along the major diagonal and 1 on the surrounding left and right diagonal. The L operator is one method for taking the double partial derivate. With L we can simply treat the heat system as an ordinary differential equation and call the function $ode113$ to solve for various solutions of the evolutions of the heat decomposition over varying time constants. This is accomplished by passing the function $dfsODE$ to the $ode113$ function. The $dfsODE$ returns product of D, L, and u, the initial matrix of the image.

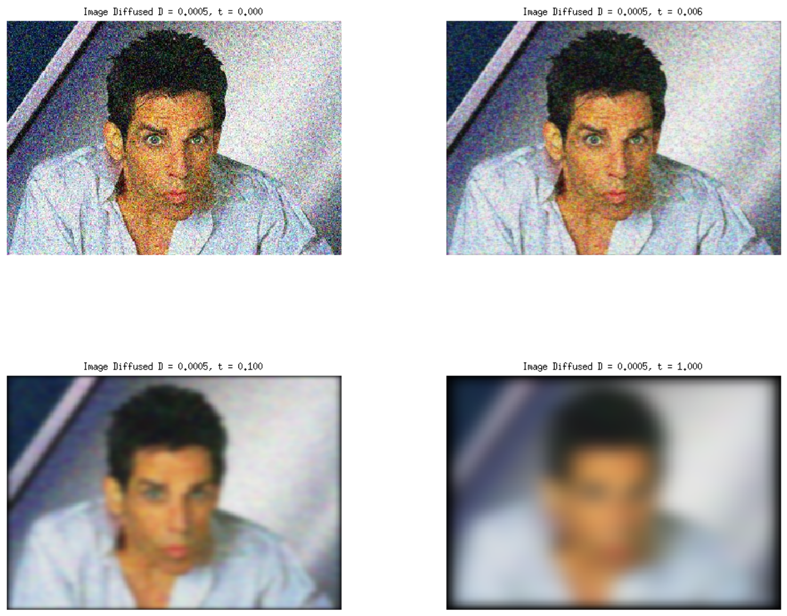
Image Diffused D = 0.0005, t = 0.000
Image Diffused D = 0.0005, t = 0.006
Image Diffused D = 0.0005, t = 0.100
Image Diffused D = 0.0005, t = 1.000

Figure 3: High global diffusion filtering of the color image of Derek. As we can see varying the time constant of the heat equation can have both positive and negative effects on the quality of the resulting image. For example, by under-filtering with a tau of 0 our image remains noisy but longer time constants will blur our image.

3.3 Local Diffusion

In the case of the latter two images of Derek, the photo was only corrupted in one minor area. In order to specifically filter this area, the approach of copying the affected area, running diffusion on this patch then restitching the diffused portion back into the original image was used. This is the purpose of the function *smoothArea* that takes a pixel focal point, and width of an area to be diffused. In order to choose an ideal diffusion location, close inspection with the program *gimp* was used. However, there were some modifications needed to improve the quality of the local area. As learned from experimenting with strong diffusion in Figure 3, long time constants of the type needed to successfully blur the areas of the image make the edges of the image dark and dissipate energy over the region in general. In order to combat these effects, the diffused area was trimmed before adding it back into the final image and a constant was added to every element to mildly brighten the area back to its likely original value.

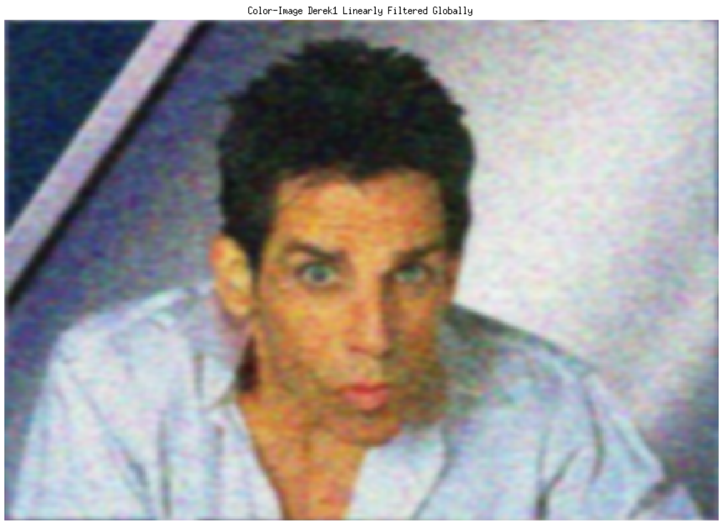

Figure 4: The globally linearly filtered images of Derek in color. The strength of filter, in this case .0005 was chosen to get the best trade-off to recover the original image.

4 Computational Results

From our results we can see that both methods are effective at denoising the images. Applying the linear filtering method and choosing the strength of filter optimally for the first color photo yields the results shown in 4. This result shows a mix between the tradeoffs of the high contrast but noisy original image with an overfiltered highly blurred image. In this figure, the filter strength used is .0005.

Applying the linear filter with a variety of strengths on the black and white image yields similar results as shown in 2. In this figure, the original image appears in the top left, but as we move right and downward stronger and stronger filtering strengths are used. A stronger filtering strength corresponds to the parameter D, the width of our Gaussian filter from Equation 3. While we lose image quality due to high noise with no filter, applying too strong of filter, blurs the entire image as shown by the bottom right picture in Figure 2. These pictures were used as a proxy for choosing the filter strength of the color photo.

To experiment and compare, diffusion was also used on the first two globally noisy images as shown in Figure 3. We can see that over-filtering with the method of diffusion has similar effects as linear filtering with some certain peculiarities. These differences are especially evident in the over-filtered picture in the bottom right. This image appears darker than the rest especially around the edges. This is an immediate warning that some adjustments need

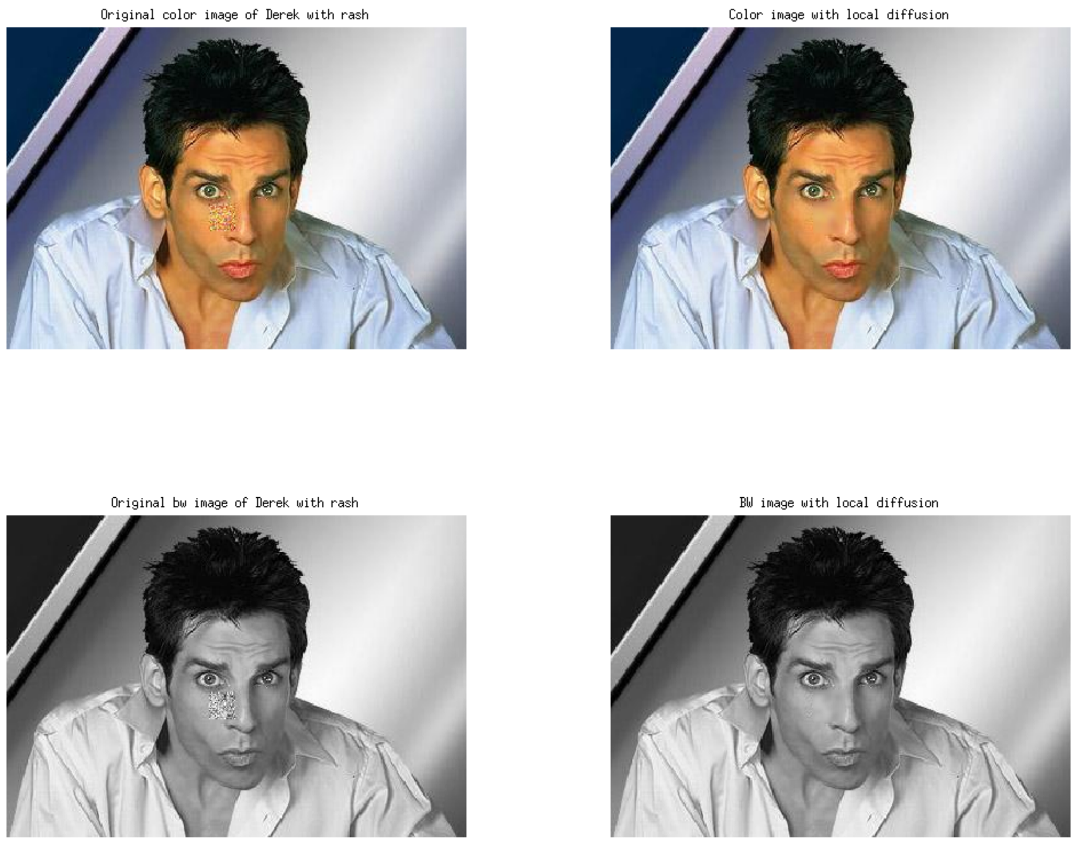
Original color image of Derek with rash
Color image with local diffusion
Original bw image of Derek with rash
BW image with local diffusion

Figure 5: Color and black and white images on the left of Derek shown with a rash near his nose. The rash appears to be a local area of noise. An effective way to treat this image would be to filter around only that specific area. Local diffusion of the area produces retouched images shown on the right.

to be made to make a clean local diffusion in the final image.

Avoiding some of the effects of diffusion such as local darkening and edge darkening yielded the final local diffusion technique shown in Figure 5. The left images show the originals and the right images the retouched versions. The noise in the local areas was much higher than the global noise in the first two images considering that it took a tau of 1 and D of .0005 to produce the final denoised images shown.

5 Summary and Conclusions

The techniques used in this report reaffirm the practicality of linear algebra to a wide array of problems. By casting image data of an arbitrary type to a regular double matrix, we were able to apply the abstract mathematical techniques to have practical effects on data. We were able to apply past concepts such as low pass filtering to take advantage of the fact that noise in image data tends to be at higher frequencies. Similarly, we were able to take the analogy of the diffusion of heat applied to filtering of image data, finding that the analytical solution and the implementation produces an equivalent technique as linear filtering.[1]

A
imread(A, B) read image data of name A type B into a uint8 matrix
uint8() cast data into uint8 format
double() cast data into double format
eye(A) identity matrix of size A
kronA,B If A is an m-by-n matrix and B is a p-by-q matrix, then kron(A,B) is an m*p-by-n*q matrix formed by taking all possible products between the elements of A and the matrix B.
ode113(A,B,C,D) odinary differential equation solver of ODE suite takes the function handle A and evolves the function according to the time span specified
spdiags(b,d,m,n) creates an m-by-n sparse matrix by taking the columns of B and placing them along the diagonals specified by d.

B

```
% Image Filtering.m  Main Script for calling functions and plotting
% images
close all; clear all

strength = .0005;
strengths = [.1 .01 .001 .0001];

% load dereks
num = 4;
for j=1:num
  name{j} = strcat('derek',int2str(j));
  im.raw{j} = imread(name{j}, 'jpg');
end

% Filter first (two) image(s) with gaussian
% low pass filter globally
for j=1:1
  im.Flt{j} = imageflt(im.raw{j}, strength);
end

% color
figure(1)
% set(gcf, 'visible', 'off');
imshow(im.Flt{1}, []);
title('Color-Image Derek1 Linearly Filtered Globally');
saveas(1,'derek1F','png')

break
strengths = [.1 .01 .001 .0001];
% b/w
figure(2)
% set(gcf, 'visible', 'off');
for j = 1:length(strengths)
  subplot(2,2,j);
  im.Flt2{j} = imageflt(im.raw{2}, strengths(j));
  imshow(im.Flt2{j});
  title(sprintf('Linearly Filtered strength: %0.4f', strengths(j)));
end
saveas(2,'derek2F','png');

% Diffuse first im globally
```

```
tau=[0 0.006 0.1 1]; % time of diffusion
D=0.0005; % diffusion coefficient

for j=1:1
  im.Dfs{j} = imagedfs(im.raw{j}, tau, D);
end

% color
figure(3)
% set(gcf, 'visible', 'off');
for j=1:length(tau)
  subplot(2,2,j);
  imshow(im.Dfs{1}(:,:,:,j));
  title(sprintf('Image Diffused D = %0.4f, t = %0.3f', D, tau(j)));
end
saveas(3,'derek1D','png');

% b/w
% figure(4)
% set(gcf, 'visible', 'off');
% for j = 1:length(tau)
%   subplot(2,2,j);
%   imshow(im.Dfs{2}(:,:,:, j));
%   drawnow; pause(3);
%   % saveas(2,'derek2D','png');
% end

%%%%%%%% Part II local diffusion %%%%%%%%%%

tau=[0 0.002 0.004 1]; % time of diffusion
D=0.0025; % diffusion coefficient

% center pixel no. of rash location
% estimated with gimp
centery = 172;
centerx = 150;
% square width of affected region
pixSize = 34;

% Diffuse local area of rash for last two images
for j=1:2
  im.dfsL{j} = smoothArea(im.raw{j + 2}, ...
     centerx, centery, pixSize, tau, D);
end

figure(5)
subplot(2,2,1)
imshow(im.raw{3})
title('Original color image of Derek with rash')
subplot(2,2,3)
imshow(im.raw{4})
title('Original bw image of Derek with rash')

%color retouched
subplot(2,2,2)
imshow(im.dfsL{1}(:,:,:,length(tau)));
title('Color image with local diffusion')

% b/w
subplot(2,2,4)
imshow(im.dfsL{2}(:,:,:,length(tau)));
title('BW image with local diffusion')
saveas(5,'derekLD','png');

function [filImage] = imageflt(image, strength)
  % Takes a uint8 2 or 3 dimensional matrix and
  % an int strength (which parameterizes the width
  % of the filter in the spectral domain) and applies
  % gaussian filtering in the 2nd dimension
  % returning the filtered uint8 matrix back

  [x,y,z]=size(image);
  kx = 1:x; ky = 1:y;
  [Kx, Ky] = meshgrid(kx,ky);
  centerx = ceil(x/2);
  centery = ceil(y/2);
  % transposed Gaussian filter
  fil = (exp(-strength*(Kx - centerx).^2 - strength*(Ky - centery).^2))';
  for k=1:z
    imageFFT = fftshift(fft2(double(image(:,:,k))));
    temp = imageFFT.*fil;
    filImage(:, :, k) = uint8(ifft2(ifftshift(temp)));
  end
```

```
end

function [dfsImage] = imagedfs(image, tau, D)
  % Takes a uint8 2 or 3 dimensional matrix and
  % an int strength (which parameterizes the width
  % of the filter in the spectral domain) and applies
  % gaussian filtering in the 2nd dimension
  % returning the filtered uint8 matrix back

  [x,y,z]=size(image);
  L = buildL(x,y);
  for k=1:z
    % convert to double place in vector form
    u = reshape(double(image(:,:,k)), x*y, 1);
    % evolve through diffusion/heat equation
    [t, usol] = ode113('dfsODE', tau, u, [], L, D);
    % place back into dfs image
    for l = 1:length(tau);
      % create an extra dimension l for each tau value
      dfsImage(:, :, k, l) = uint8(reshape(usol(l,:),x,y));
    end
  end
end

function [L] = buildL(nx, ny)
  % Build the L operator for diffusion
  % equation of 2-d matrix
  % return L

  x=linspace(0,1,nx);
  y=linspace(0,1,ny);
  dx=x(2)-x(1);
  dy=y(2)-y(1);
  ones_x=ones(nx,1);
  ones_y=ones(ny,1);
  Dx=(spdiags([ones_x -2*ones_x ones_x],[-1 0 1],nx,nx))/dx^2;
  Dy=(spdiags([ones_y -2*ones_y ones_y],[-1 0 1],ny,ny))/dy^2;
  Iy=eye(ny);  % identity matrix of size ny
  Ix=eye(nx);
  L=kron(Iy,Dx)+kron(Dy,Ix);
end
```

```
function dfs = dfsODE(t, u, ~, L, D)
  % handle for ODE function of diffusion
  dfs = D*L*u;
end

function [smoothImage] = smoothArea(image, ...
  centerX, centerY, pixSize, tau, D)
  % Takes a uint8 matrix image, center pixel
  % location in x y coordinates and a pixel size
  % of a local area of an image to clean
  % Takes doubles tau and diffusion constant D
  % for diffusion filtering
  % cleans the specific area of the image and
  % returns the new matrix in unint8 format

  pixelTrimFactor = 5;
  brightEnhance = 6;
  [x,y,z] = size(image);
  hWid = floor(pixSize / 2);
  lowX = centerX - hWid; highX = centerX + hWid;
  lowY = centerY - hWid; highY = centerY + hWid;

  % Cut patch out of image
  temp = image(lowX:highX, lowY:highY, :);
  % create higher dimensional copies for time slices
  for k = 1:length(tau)
    smoothImage(:,:,:, k) = image(:,:,:);
  end

  % Diffuse locally on that patch
  smoothTemp = imagedfs(temp, tau, D);
  [m,n,o,p] = size(smoothTemp);
  % trim out edges  % add energy of brightEnhance to compensate for diffusion
  trimTemp = smoothTemp(((1 + pixelTrimFactor):(m - pixelTrimFactor)), ...
     ((1 + pixelTrimFactor):(n - pixelTrimFactor)), :, :) + brightEnhance;
  %place back into higher dimensional copies
  smoothImage((lowX + pixelTrimFactor):(highX - pixelTrimFactor), ...
     (lowY + pixelTrimFactor):(highY - pixelTrimFactor), :, :) = trimTemp;

end
```

References

[1] J N Kutz. *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*. OUP Oxford, 2013.