Amath 582 Homework 3: Trouble at the Zoo

Due: February 4, 2016

Brian de Silva[1]

Abstract
In this homework writeup we use the Fast Fourier Transform to filter out high frequency components from noisy images in order to clean them up. We then use localized diffusion to smooth small corrupted domains in two images. This allows them to better blend in with their respective images to create more natural looking pictures.

[1] *Department of Applied Mathematics, University of Washington, Seattle*

Contents

1. Introduction

Derek Zoolander is in trouble! Ugly protesters are (literally) trying to tarnish his good image by sabotaging his touched-up photos with noise. Even worse, he did something completely rash and went into a photo shoot while nasty skin condition was flaring up on his face. In this assignment we attempt to restore the corrupted pictures to their former glory through the filtering of high frequency content using the Fast Fourier Transform (FFT). Once that crisis has been dealt with we use localized diffusion to remove the rash from Derek's photos and ultimately save the day.

This writeup is structured as follows: in Section 2 we give a brief review of the theory behind our techniques. Next we discuss our algorithms and their implementations in MATLAB in Section 3 before discussing our numerical results in Section 4. Finally we make some closing remarks in Section 5.

2. Theoretical Background

2.1 Denoising images with the Fast Fourier Transform

In much the same way we previously used the FFT to eliminate excess noise and uncover underlying signals from ultrasound measurements in Homework 1, we can also remove noise from images. To do so one first represents an image as a matrix, U, whose entries correspond to values at different pixels. For simplicity assume that the image is black and white. Then we pair with each pixel a scalar value between 0 and 255 inclusive, which gives the intensity of the light at that point. A value of 0 means that pixel is completely black, and a value of 255 means it is as white as possible. Values strictly between the two correspond to various shades of gray.

Given U one can take its 2-dimensional Discrete Fourier Transform to obtain its representation, $\hat{U}$, in the frequency domain. One may then determine the center frequency (the one of largest magnitude) and filter out frequencies which are not close to it, in some sense of the word "close". For natural images the center frequency is typically the one with wave components both 0, i.e. the lowest possible frequency in both the x and y directions. For this assignment we tested Gaussian and Shannon filters which remove frequencies which are not close to the center frequency, e.g. the high frequency content is removed if we are working with natural images. We also experiment with a thresholding filter which zeroes out Fourier components with magnitudes below some specified percentage of the magnitude of the center frequency.

2.2 Denoising images with diffusion

Another method of denoising an image is to cause a small amount of diffusion using the 2-dimensional heat equation:

$$u_t = D\nabla^2 u = D(u_{xx} + u_{yy}) \tag{1}$$

where $u(x,y)$ is the image in question, $D = D(x,y)$ is a diffusion coefficient, and some boundary conditions must be imposed. (1) models the spread of heat throughout a substance. $D(x,y)$ controls the rate of heat spread at each point (x,y). We can discretize (1) using the standard second order

finite difference approximations

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\Delta x,y)-2u(x,y)+u(x-\Delta x,y)}{\Delta x^2} \tag{2}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u(x,y+\Delta y)-2u(x,y)+u(x,y-\Delta y)}{\Delta y^2} \tag{3}$$

to obtain a system of coupled ODEs of the form

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{L}\boldsymbol{u} \tag{4}$$

for a sparse matrix $\boldsymbol{L}$ constructed using (2) and (3). In (4) we have discretized the spatial domain so that if $u(x_n, y_m) = u_{nm}$, then

$$\boldsymbol{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ \vdots \\ u_{1n} \\ u_{21} \\ u_{22} \\ \vdots \\ u_{n(n-1)} \\ u_{nn} \end{pmatrix}. \tag{5}$$

We can easily compute numerical approximations to the solution of (4) using any standard time-stepping scheme (we use a fourth order predictor-corrector Runge-Kutta method). Stacking the entries of the array of pixel values associated with an image into a vector allows us to feed the image into the ODE solver as the initial condition. Advancing time causes diffusion in the image.

One desirable property of the heat equation (1) is that the highest frequency components in the initial data are the most rapidly damped out by diffusion. This phenomenon has a particularly noticeable effect on image noise, which often consists of high frequencies. One can generally evolve time a short enough amount of time so as not to cause too much diffusion and degrade the quality of an image, but long enough to diffuse much of the noise. One can also choose to define the diffusion coefficient $D(x,y)$ so that diffusion is localized, e.g. by choosing it to be 0 in regions where diffusion should not occur and making it nonzero elsewhere. This is the approach we take to complete Task 2.

3. Algorithms Implementation and Development

In this section we describe the scripts developed in MATLAB to clean up the sabotaged images of Derek and to hide his rash using filtering and diffusion. Overviews of the two algorithms are given below and are accompanied by relevant details.

3.1 Task 1

Below we outline the MATLAB routine written to denoise the corrupted images of Derek. Our primary tool is filtering frequencies of lesser importance (often high frequencies) in the Fourier domain.

1. Read in the images and take their DFTs

We split the color image into its red, green, and blue (RGB) components and take each of their DFTs separately. From plots of the magnitudes of the frequency components we can determine the center frequencies and appropriate filters to employ.

2. Filter out the Fourier components of lesser importance

Typically the lower frequencies have larger weight and the very high frequencies correspond to noise and regions of sharp contrast. We filter out many of the high frequency content in the hopes of eliminating noise. For this assignment we experimented with a Gaussian filter, a Shannon (step function) filter, and a thresholding filter. The third of these simply removes all frequency components which have magnitudes less than some specified fraction of the magnitude of the center frequency. We apply filters to the DFTs of the red, green, and blue components of the color image separately.

3. Invert the filtered transforms and view the (hopefully) denoised pictures

We recombine the three components of the color photo back into a single color image.

3.2 Task 2

Here we describe the algorithm designed to remove the rash from Derek's otherwise pristine pictures. Our workhorse will be using the 2-D heat equation to induce diffusion of the pixels making up the rash.

1. Read in the images

Again we split the color image into its RGB components.

2. Discretize the 2-D heat equation by forming the 2-D discrete Laplacian matrix

As detailed in Section 2 discretizing the Laplacian differential operator we obtain a sparse matrix $\boldsymbol{L}$. So as not to degrade the outstanding quality of the images away from the rash we choose a non-constant diffusion coefficient $k = k(x,y)$ which is 0 outside of the rash's domain and 0.0005 inside it. This prevents any diffusion from occurring anywhere but Derek's rash.

3. Numerically solve the ODEs out to some time, using the images as the initial conditions

We use MATLAB's build in fourth order adaptive ODE solver `ode45` to advance the solution in time. See Appendix A for more details on `ode45`. The further in time we evolve the system the more diffusion takes place.

4. Computational Results

In this section we begin by giving an overview of the results of our attempts to denoise the two sabotaged pictures of Derek's face. We discuss the effects different filters have

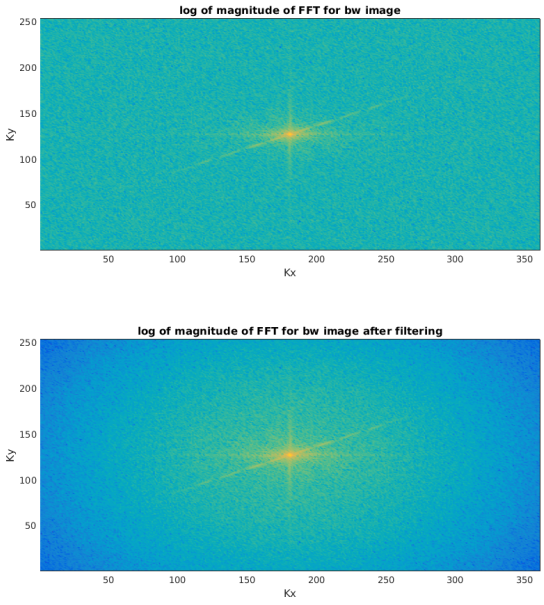


Figure 1. The frequency components of the noisy black and white image. Top: before filtering. Bottom: after Gaussian filter has been applied (width=0.0001)

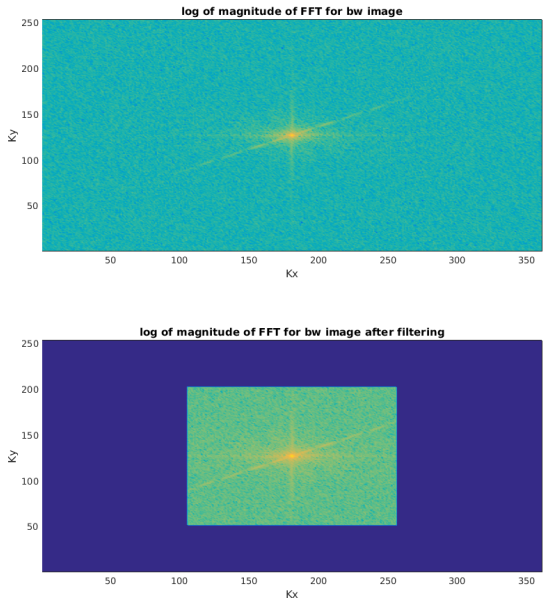


Figure 2. The frequency components of the noisy black and white image. Top: before filtering. Bottom: after Shannon filter has been applied (width=75)

on the cleaned up images. Next we showcase our success at removing the unfortunate rash from Derek's other two head shots.

4.1 Task 1 - denoising corrupted photos

First we visualize the FFTs of the two noisy images and the effects the three filters have upon them. Figure 1 plots the natural logarithm of the magnitudes of the frequency components of the noisy black and white image, both before and after a Gaussian filter has been used. Notice how most of the information seems to be contained near the center of the image. For this reason we center our Gaussian and Shannon filters about the "brightest" point. In the bottom of Figures 1 and 2 we see that the two filters eliminate frequency content away from the center Fourier component. With the Gaussian filter we witness a smooth, but rapid decay of the components as we move away from the center, whereas the Shannon filter has a drop off 75 pixels away from the center frequency. One could argue that more of the noise is allowed to persist when the Gaussian filter is used since the figures suggest it damps out fewer of them than the Shannon filter, but we will see that they achieve similar performance. The thresholding filter may eliminate the most frequency components as is demonstrated in Figure 3. However, unlike the Gaussian and Shannon filters, it does not target only high frequencies. Many low frequencies are eliminated and many high frequencies survive the filtering process. This will have repercussions when the inverse FFT is taken.

For completeness we also plot the FFTs of the RGB components of the color image before and after Gaussian filtering

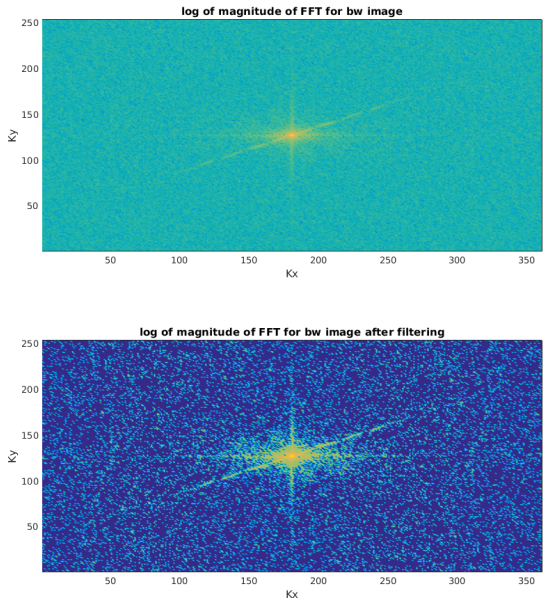


Figure 3. The frequency components of the noisy black and white image. Top: before filtering. Bottom: after thresholding filter has been applied (every component with magnitude less than 60% of max magnitude is zeroed out)

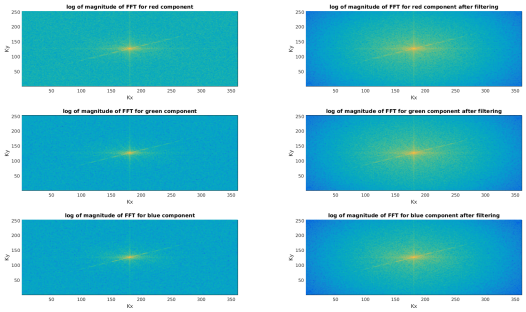


Figure 4. The frequency components of the red (top), green (middle), and blue (bottom) components of the noisy color image. Left column: before filtering. Right column: after Gaussian filter has been applied (width=0.0001)

in Figure 4. These plots look qualitatively very similar to their black and white image counterparts.

Figure 5 shows the original and denoised black and white photos of Derek for a Gaussian filter and Figures 6 - 8 show the original and denoised color pictures using Gaussian, Shannon, and thresholding filters, respectively. The cleaned images use the same width and thresholding parameters as were used to produce the plots of the filters. The black and white image was fairly noisy to begin with so while we are able to make marginal improvements to the sabotaged image, none of the filters were able to completely remove the noise. We saw more success with the color image, possibly because it essentially consists of three times as much information. The Gaussian and Shannon filters appear give similar (satisfactory) results, however, upon closer inspection the Gaussian filter actually slightly outperforms the Shannon filter, at least in the Author's opinion. One can decrease the width of the Shannon filter in an attempt to cut out more of the noise, but this leads to a significant amount of blurring. The thresholding filter clearly has does the worst job of removing noise. Too much information is lost and the picture looks blurry. Weakening the thresholding criterion slightly allows much more noise to slip through the filter, making for a much noisier touched up image.

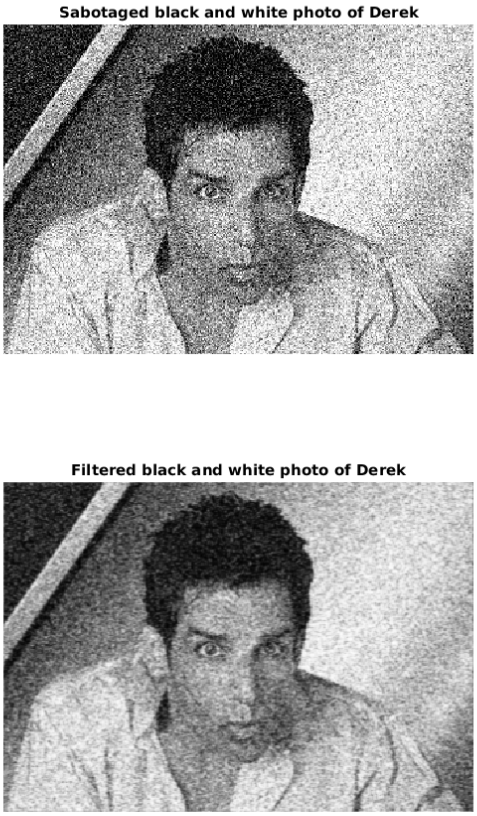


Figure 5. The original and denoised black and white pictures of Derek using a Gaussian filter

4.2 Task 2 - cleaning up a small corrupted region using diffusion

The method presented in Section 3.2 was very effective at removing the rash from Derek's face, both in his color and black and white photos. Figures 9 - 12 show how the black and white image changes as the heat equation is applied to it. Time 0 corresponds to the original picture, rash included. Already by $t = 0.015$ we see a marked increase in the smoothness of the appearance of the rash region in Figure 10. By $t = 0.045$ you can hardly tell there was ever a rash to begin with. Observe also that the parts of the image outside of the rash region do not change at all over time. This is due to the fact that we chose a piecewise constant diffusion coefficient which is 0 away from the rash, meaning that the heat equation does

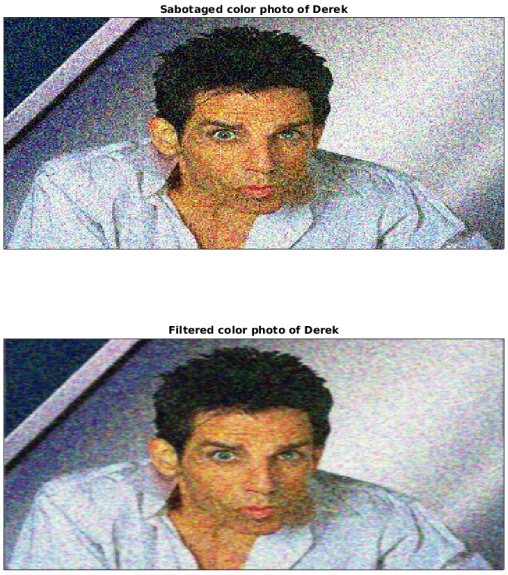


Figure 6. The original and denoised color pictures of Derek using a Gaussian filter

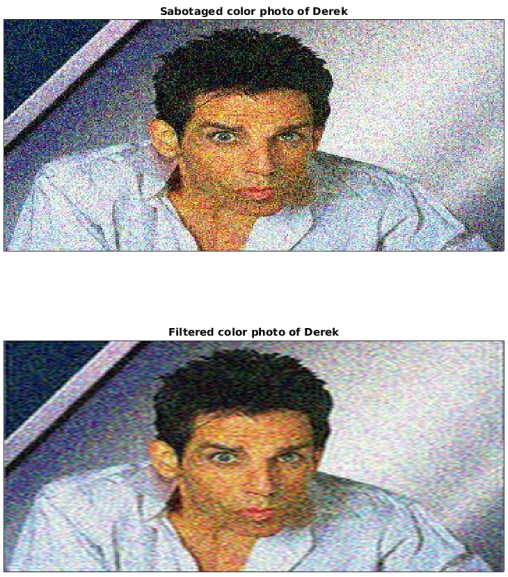


Figure 7. The original and denoised color pictures of Derek using a Shannon filter

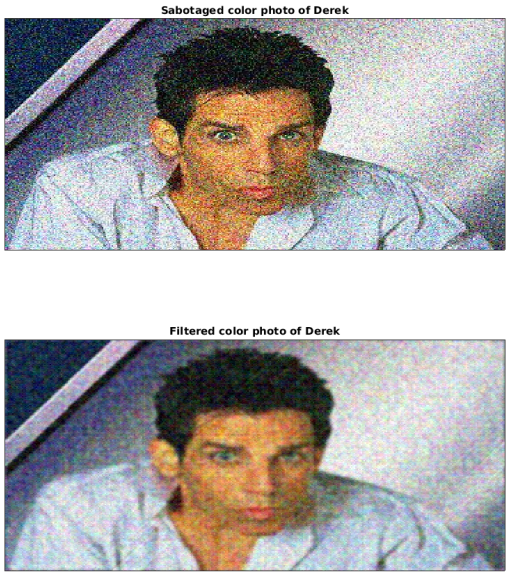


Figure 8. The original and denoised color pictures of Derek using a thresholding filter

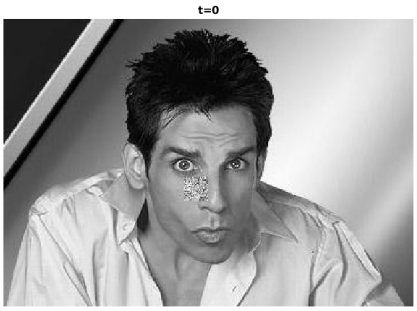


Figure 9. Diffusion applied only to the rash region of black and white photo of Derek at $t = 0$

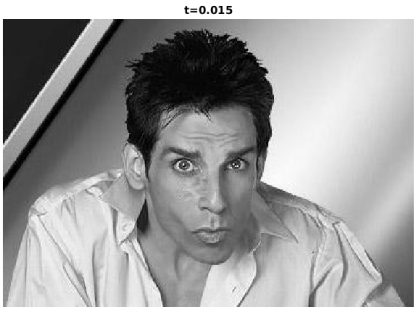


Figure 10. Diffusion applied only to the rash region of black and white photo of Derek at $t = 0.015$

Figure 11. Diffusion applied only to the rash region of black and white photo of Derek at $t = 0.03$

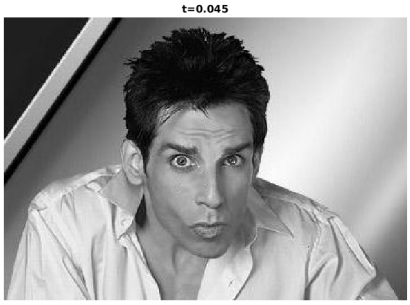


Figure 12. Diffusion applied only to the rash region of black and white photo of Derek at $t = 0.045$

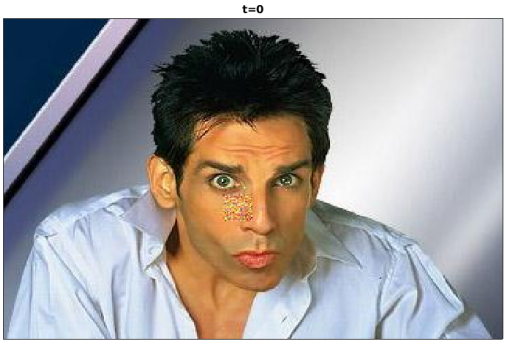


Figure 13. Diffusion applied only to the rash region of color photo of Derek at $t = 0$

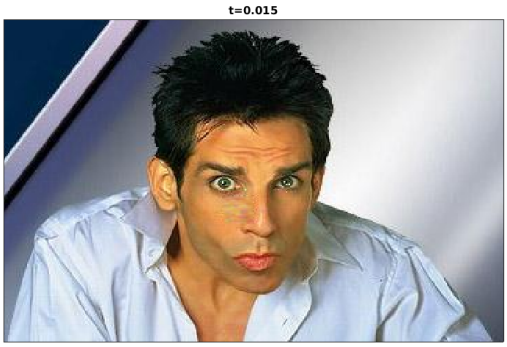


Figure 14. Diffusion applied only to the rash region of color photo of Derek at $t = 0.015$

not affect this area since it then reduces to $u_t = 0$.

Figures 13 - 16 give snapshots of the solutions of the heat equation using the color picture in Figure 13 as initial data. Again the method performs very well and produces satisfactory results by $t = 0.03$ (see Figure 15). If one looks closely at the rash region, in Figure 16, say, it is easier to spot artifacts of our diffusive smoothing. In particular the pixels in the square where diffusion was applied are a slightly different color than their neighbors outside the rash region. It may be possible to avoid this effect by slowly increasing the size of the domain where diffusion takes place so that the colors better blend together.

5. Summary and Conclusions

We succeeded in helping Derek save face! In this writeup we developed a method based on filtering in the Fourier domain to clean up noisy pictures. We found a Gaussian filter to give the best performance, though we do not rule out the possibility that a Shannon filter with a different shape or window size could outperform a simple Gaussian filter. The black and white image proved to be more difficult to denoise than the color one. We plotted representative ouputs of our algorithm with each of the three filters considered.

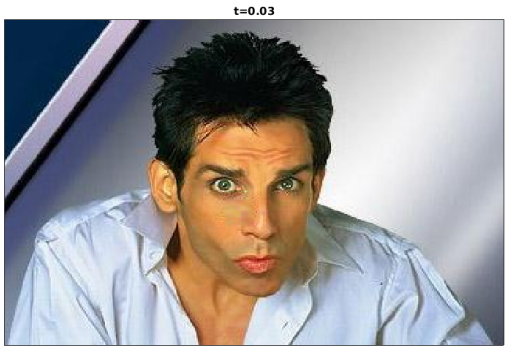


Figure 15. Diffusion applied only to the rash region of color photo of Derek at $t = 0.03$

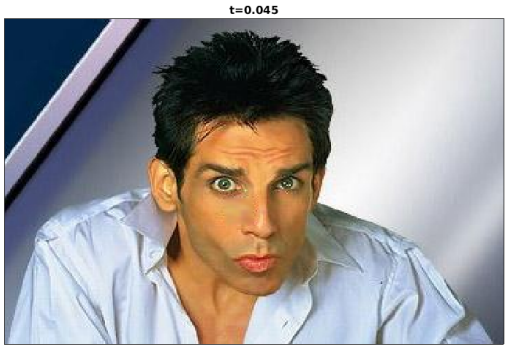


Figure 16. Diffusion applied only to the rash region of color photo of Derek at $t = 0.045$

We also wrote and described an algorithm for removing evidence of a rash from Derek's face in both color and black and white photos using diffusion. For the given pictures a diffusion coefficient of 0.0005 inside the rash region was sufficient to smooth the rash by $t = 0.045$. Choosing a diffusion coefficient which was 0 outside this region allowed us to preserve the high quality of the rest of the image while applying diffusion just where we wanted it. Finally we show off some of the touched up photos.

Appendix A: MATLAB Functions

Here we outline the nonstandard MATLAB functions used to complete this assignment.

`cat(dim,A,B,C)`: Concatenate the arrays `A`, `B`, and `C` along the dimension specified by `dim`.

`fft2(X)`: Given a $2-$dimensional array of data, `X`, `fft2(X)` returns the $2-$dimensional DFT of `X`. The output has the same size as the input. To plot the frequency components given by `fft2` one should first apply the `fftshift` command as `fft2` returns a shifted version of the frequency components.

`fftshift(v)`: This command shifts the vector/matrix output of `fft`, `fft2`, and `fftn` so that the 0 frequency lies at the center of the vector/matrix. For example, in one dimension `v = fft(data)` is a vector and `fftshift(v)` swaps the first and second halves of `v`. This command is useful for plotting the FFT of data.

`imread(filename,format)`: Reads in an image file with format `format` from the location specified by the string `filename` and stores the result in a two or three dimensional array. If it is a grayscale image then it saves a 2-dimensional array with an entry for each pixel in the image. If it is a color image it saves a 3-dimensional array composed of three 2-dimensional arrays giving the red, green, and blue components of each pixel.

`kron(A,B)`: Returns the Kronecker-tensor product of the matrices `A` and `B`. If we denote the entries of `A` by $a_{i,j}$ then

$$\texttt{kron(A,B)} = \begin{pmatrix} a_{1,1}\texttt{B} & a_{1,2}\texttt{B} & \cdots & a_{1,n}\texttt{B} \\ a_{2,1}\texttt{B} & a_{2,2}\texttt{B} & \cdots & \vdots \\ \vdots & & \ddots & \\ a_{m,1}\texttt{B} & \cdots & & a_{m,n}\texttt{B} \end{pmatrix}.$$

`ode45(odefun,tspan,y0)`: Uses a fourth order adaptive Runge-Kutta time-stepping scheme to find a numerical solution to the following ordinary differential equation at all the times specified in the vector `tspan`

$$\begin{cases} \frac{dy}{dt} = \texttt{odefun}(t,y), \\ y(t_0) = \texttt{y0}. \end{cases}$$

In the above y could be a vector and `odefun` may be vector-valued. The outputs `t` and `Y` in the expression `[t, Y] = ode45(odefun,tspan,y0)` are a vector of the times when the numerical solution is known and a matrix containing the numerical solution at each time it was requested. It should be noted that `t` = `tspan` if `length(tspan)` $>$ 2.

`spdiags(V,D,n,m)`: Produces an `n`$\times$`m` sparse matrix with entries given by the i^{th} column of `V` entered `D(i)` off the diagonal, for $i = 1, 2, \ldots,$ `length(D)`. That is the i^{th} column of `V` is placed along the $D(i)^{\text{th}}$ superdiagonal (or subdiagonal if `D(i)` < 0) of the output.

`speye(n)`: Generates the sparse representation of the $n \times n$ identity matrix.

Appendix B: MATLAB Code

See the following pages published in MATLAB for the implementation of the algorithm presented in Section 3.

Table of Contents

Amath 582 Homework 3

Brian de Silva 1242824

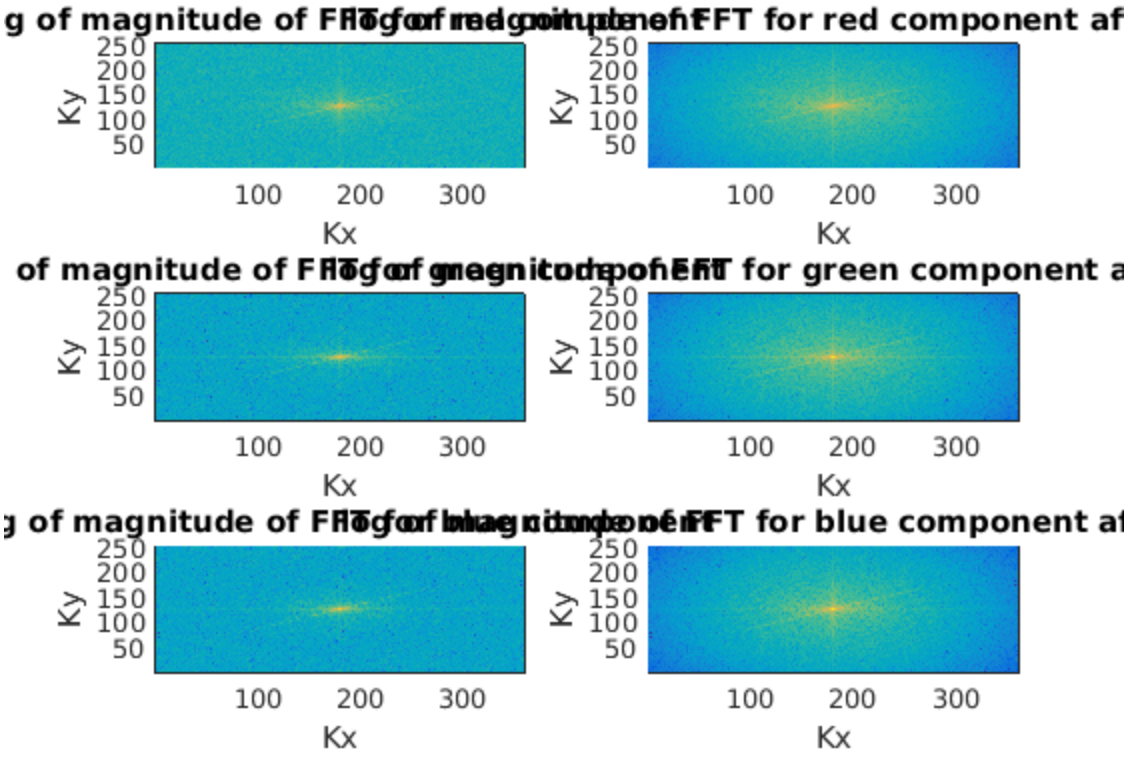
```
clear all; close all;
```

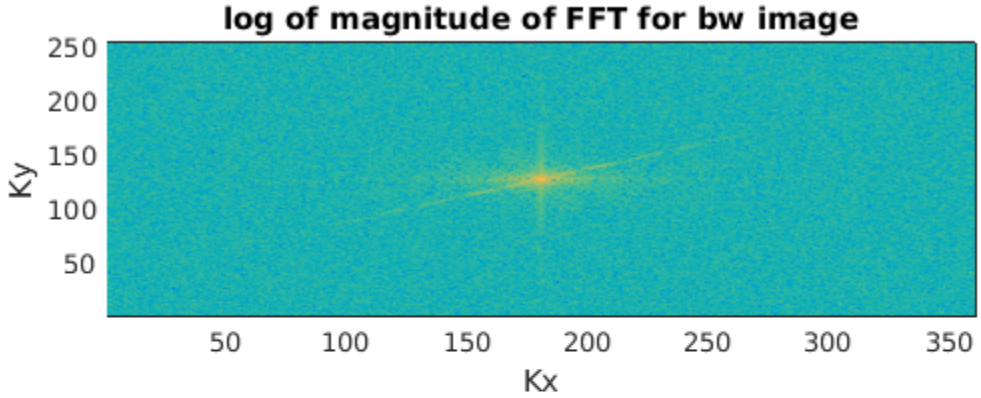
Task 1 - clean up the corrupted images of Derek

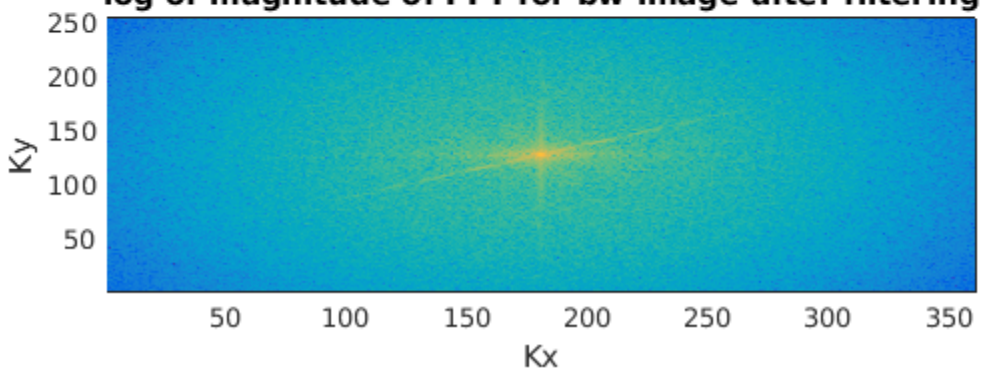
```
% Global parameters (set specific filter parameters below)
generate_plots = true;                % Determines whether or not
 script makes plots
filter_choice = 'gaussian';           % Choice of filter: {'gaussian',
 'shannon', 'threshold'}


Acol = imread('derek1','jpg');      % Load color image
Abw = imread('derek2','jpg');      % Load black & white image

% Convert uint8 arrays to double arrays and split the color image
% into its RGB components
Abw = double(Abw);
AR = double(Acol(:,:,1));
AG = double(Acol(:,:,2));
AB = double(Acol(:,:,3));

% Take 2D FFT of each image/image component
Abwt = fft2(Abw);
ARt = fft2(AR);
AGt = fft2(AG);
ABt = fft2(AB);

% Construct frequency vectors
kxcenter = 181;                          % Specify center frequencies
kycenter = 127;
kx = 1:size(Abw,2);                     % Shifted frequencies
ky = 1:size(Abw,1);
[Kx, Ky] = meshgrid(kx,ky);

switch filter_choice
    case 'gaussian'
        % Contstruct Gaussian filter
        xwidth = 0.0001;
        ywidth = 0.0001;
```

```
        filter = exp(-xwidth*(kxcenter - Kx).^2 - ywidth*(kycenter -
 Ky).^2);
    case 'shannon'
        % Construct Shannon (step function) filter
        width = 75;
        filter = zeros(size(Abwt));
        filter((kycenter-width):(kycenter+width),(kxcenter-width):
(kxcenter+width)) = ones(2*width+1,2*width+1);
    case 'threshold'
        % Filter out frequency components with too low magnitude
        thresh = 0.6;
        ARtf = ARt .* (log(abs(ARt)+1) > thresh *
 max(log(abs(ARt(:))+1)));
        AGtf = AGt .* (log(abs(AGt)+1) > thresh *
 max(log(abs(AGt(:))+1)));
        ABtf = ABt .* (log(abs(ABt)+1) > thresh *
 max(log(abs(ABt(:))+1)));
        Abwtf = Abwt .* (log(abs(Abwt)+1) > thresh *
 max(log(abs(Abwt(:))+1)));
end

% Apply filter (if not using thresholding)
if exist('filter','var')
    ARtf = ifftshift(filter.*fftshift(ARt));
    AGtf = ifftshift(filter.*fftshift(AGt));
    ABtf = ifftshift(filter.*fftshift(ABt));
    Abwtf = ifftshift(filter.*fftshift(Abwt));
end

% Plot Fourier components before and after filtering
if generate_plots

    % Color image
    figure()
    subplot(3,2,1)
    pcolor(log(fftshift(abs(ARt)+1))), shading interp
    title('log of magnitude of FFT for red component'), xlabel('Kx'),
 ylabel('Ky')
    subplot(3,2,3)
    pcolor(log(fftshift(abs(AGt))+1)), shading interp
    title('log of magnitude of FFT for green component'),
 xlabel('Kx'), ylabel('Ky')
    subplot(3,2,5)
    pcolor(log(fftshift(abs(ABt))+1)), shading interp
    title('log of magnitude of FFT for blue component'), xlabel('Kx'),
 ylabel('Ky')
    subplot(3,2,2)
    pcolor(log(fftshift(abs(ARtf))+1)), shading interp
    title('log of magnitude of FFT for red component after
 filtering'), xlabel('Kx'), ylabel('Ky')
    subplot(3,2,4)
    pcolor(log(fftshift(abs(AGtf))+1)), shading interp
    title('log of magnitude of FFT for green component after
 filtering'), xlabel('Kx'), ylabel('Ky')
```

```
    subplot(3,2,6)
    pcolor(log(fftshift(abs(ABtf))+1)), shading interp
    title('log of magnitude of FFT for blue component after
 filtering'), xlabel('Kx'), ylabel('Ky')

    % Black and white image
    figure()
    subplot(2,1,1)
    pcolor(log(fftshift(abs(Abwt))+1)), shading interp
    title('log of magnitude of FFT for bw image'), xlabel('Kx'),
 ylabel('Ky')
    subplot(2,1,2)
    pcolor(log(fftshift(abs(Abwtf))+1)), shading interp
    title('log of magnitude of FFT for bw image after filtering'),
 xlabel('Kx'), ylabel('Ky')
end


% Invert transform

Acolf = uint8(zeros(size(Acol)));
Acolf(:,:,1) = uint8(ifft2(ARtf));
Acolf(:,:,2) = uint8(ifft2(AGtf));
Acolf(:,:,3) = uint8(ifft2(ABtf));
Abwf = uint8(ifft2(Abwtf));

% Plot original and filtered images
if generate_plots
    % Color images
    figure()
    subplot(2,1,1)
    image(Acol); set(gca,'Xtick',[],'Ytick',[])
    title('Sabotaged color photo of Derek')

    subplot(2,1,2)
    image(Acolf); set(gca,'Xtick',[],'Ytick',[])
    title('Filtered color photo of Derek')

    % Black and white images
    figure()
    subplot(2,1,1);
    imshow(uint8(Abw))
    title('Sabotaged black and white photo of Derek')
%     title(findobj('Parent',hFig,'Type','axes'),'Sabotaged black and
 white photo of Derek')
    subplot(2,1,2);
    imshow(Abwf)
    title('Filtered black and white photo of Derek')
%     title(findobj('Parent',hFig,'Type','axes'),'Filtered black and
 white photo of Derek');
end
```


g of magnitude of FFT for red component af
of magnitude of FFT for green component a
g of magnitude of FFT for blue component af
Ky
Kx
250
200
150
100
50
100
200
300


log of magnitude of FFT for bw image
Ky
Kx
250
200
150
100
50
50
100
150
200
250
300
350


log of magnitude of FFT for bw image after filtering
Ky
Kx
250
200
150
100
50
50
100
150
200
250
300
350

Sabotaged color photo of Derek

Filtered color photo of Derek

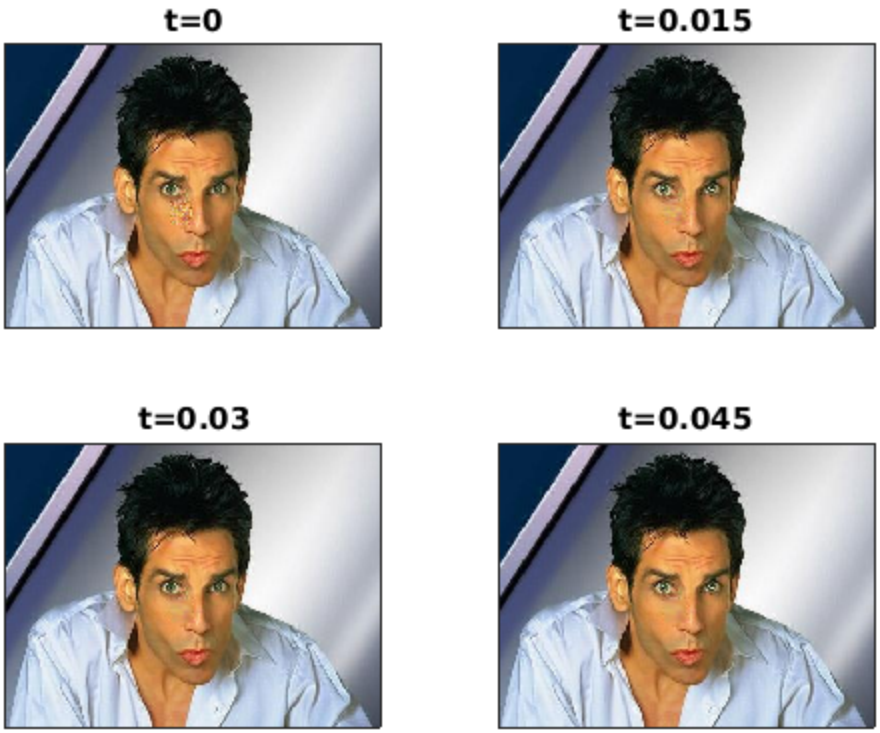
Sabotaged black and white photo of Derek

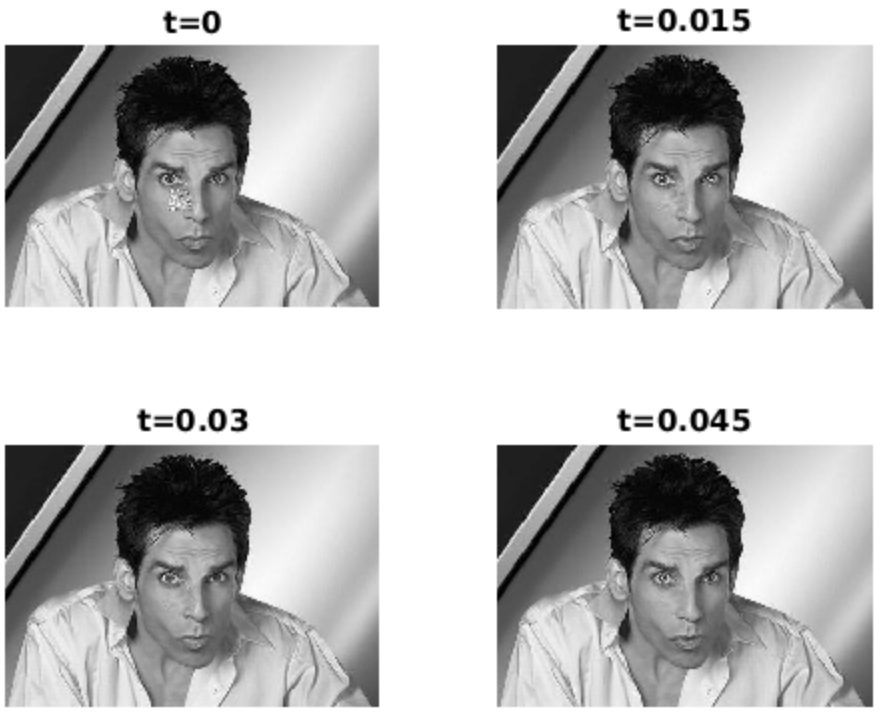
Filtered black and white photo of Derek

Task 2: Remove Derek's rash

```
Ac = imread('derek3','jpg');          % Load color image
AR = Ac(:,:,1);                              % Get red component of
 color image
AG = Ac(:,:,2);                              % Get green component of
 color image
AB = Ac(:,:,3);                              % Get blue component of
 color image
Abw = imread('derek4','jpg');       % Load black and white image
[nx, ny] = size(Abw);



% Indices of rash: (140:160, 160:180)

% Construct discretization of the Laplacian
x = linspace(0,1,nx);
y = linspace(0,1,ny);
dx = x(2) - x(1);
dy = y(2) - y(1);
onex = ones(nx,1);
oney = ones(ny,1);
Dx = spdiags([onex, -2*onex, onex],[-1 0 1],nx,nx) / (dx^2);
Ix = speye(nx);
Dy = spdiags([oney, -2*oney, oney],[-1 0 1],ny,ny) / (dy^2);
Iy = speye(ny);
L = kron(Iy,Dx) + kron(Dy, Ix);        % Discretization of the
 Laplacian
D = 0.0005;                                        % Diffusion
 coefficient inside rash domain

% Create matrix to enforce that no diffusion happens outside of the
 rash
B = zeros(size(Abw));
B(140:160,160:180) = 1;

% Define function encapsulating rhs of the ode to be solved
odefun = @(t,u) (D*L*u) .* B(:);

% Solve the ODE at different times
tpts = [0, 0.015, 0.03, 0.045];
   % Points in time when we want to view solution
[~, usolnR] = ode45(odefun, tpts, double(AR(:)));                %
 Solve the ODE for the red component
[~, usolnG] = ode45(odefun, tpts, double(AG(:)));               %
 Solve the ODE for the green component
[~, usolnB] = ode45(odefun, tpts, double(AB(:)));               %
 Solve the ODE for the blue component
[~, usolnbw] = ode45(odefun, tpts, double(Abw(:)));         % Solve
 the ODE for the black and white image

% Plot results
```

```
if generate_plots
    % Color
    figure()
    for i=1:4
        subplot(2,2,i)

 image(uint8(reshape(cat(3,usolnR(i,:),usolnG(i,:),usolnB(i,:)),nx,ny,3)));
        set(gca,'Xtick',[],'Ytick',[]);
        title(['t=' num2str(tpts(i))]);
    end

    % Black and white
    figure()
    for i=1:4
        subplot(2,2,i)
        imshow(uint8(reshape(usolnbw(i,:),nx,ny)));
        title(['t=' num2str(tpts(i))]);
    end

end
```





Published with MATLAB® R2015b