Mini-Project 2

1. Color Image Demosaicing

Implementation of DemosaicImage.m

1. Nearest Neighbors

For nearest neighbor interpolation, the value of the nearest neighbor the value of the nearest neighbor is copied to interpolate the value of the pixel.
The code snippet shown below creates masks for the Bayer pattern for the red, green and blue pattern.
It ensures that the additional pixels at the edges of the Bayer pattern, are truncated, to match the size of the image.

```
%-----------------------------------------------------------------------
%                          Nearest neighbour algorithm
%-----------------------------------------------------------------------
function mosim = demosaicNN(im)
%mosim = demosaicBaseline(im);

[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end
```

The code snippet first extracts the red, green and blue components of the image using the mask of the Bayer filter.
It then deduces the green pixels at the missing points (at the location of the red and blue pixels) using the appropriate filter and is added to the green component of the image. After this, the red pixels are interpolated at green and blue pixel locations and similarly the blue pixels are interpolated at the blue and green pixel locations, using appropriate filters and added to the respective red and blue components of the image. The filter values are obtained using the nearest neighbors of the pixels, such that the nearest neighbor is copied to interpolate the value of the respective pixels.
The three layers are combined to form the image.

```matlab

%extracting the red, green and blue components of the image using the mask

red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at missing points
green = green_image+imfilter(green_image,[0 1]);

%deducing the red pixels at missing points
redValue=im(1:2:imageWidth,1:2:imageHeight);
meanRed = mean(mean(redValue));
%red@blue
red_1 = imfilter(red_image, [0 0;0 1], meanRed);
%red@green
red_2 = imfilter(red_image, [0 1;1 0], meanRed);
%combine
red = red_image + red_1 +red_2;

%deducing the blue pixels at missing points
blueValue=im(1:2:imageWidth,1:2:imageHeight);
meanBlue = mean(mean(blueValue));
%blue@red
blue_1 = imfilter(blue_image, [0 0;0 1], meanBlue);
%blue@green
blue_2 = imfilter(blue_image, [0 1;1 0], meanBlue);
%combine
blue = blue_image + blue_1 +blue_2;

mosim(:,:,1) = red;
mosim(:,:,2) = green;
mosim(:,:,3) = blue;
```

2. Linear Interpolation

For linear interpolation, the value of the nearest neighbor the average values of the neighbors is taken to interpolate the value of the pixel.

The code snippet shown below creates masks for the Bayer pattern for the red, green and blue pattern.

It ensures that the additional pixels at the edges of the Bayer pattern, are truncated, to match the size of the image.

```matlab
%-----------------------------------------------------------------------
%                          Linear interpolation
%-----------------------------------------------------------------------
function mosim = demosaicLinear(im)
% mosim = demosaicBaseline(im);
% mosim = repmat(im, [1 1 3]);
%
[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end
```

The code snippet first extracts the red, green and blue components of the image using the mask of the Bayer filter.

It then deduces the green pixels at the missing points (at the location of the red and blue pixels) using the appropriate filter and is added to the green component of the image. After this, the red pixels are interpolated at green and blue pixel locations and similarly the blue pixels are interpolated at the blue and green pixel locations, using appropriate filters and added to the respective red and blue components of the image. The filter values are obtained by taking the average of the pixels of the neighborhood of the corresponding layers, such that the average of the pixel values is used to interpolate the value of the respective pixels.

The three layers are combined to form the image.

```
%extracting the red, green and blue components of the image using the mask
red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at missing points
green = green_image + imfilter(green_image, [0 1 0;1 0 1; 0 1 0]/4);

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

mosim(:,:,1) = red;
mosim(:,:,2) = green;
mosim(:,:,3) = blue;
```

3. Adaptive Gradient Interpolation

For adaptive gradient interpolation, the difference of the values of the top and bottom pixels (vertical gradient) is compared to the difference of the values of the values of the left and right pixels (horizontal gradient). If the vertical gradient is greater than the horizontal gradient, then the vertical gradient, then the average of the left and right pixels is assigned to the center pixel. If the horizontal gradient is greater than the vertical gradient, then the average of the top and bottom pixels is assigned to the center pixel.

The code snippet shown below creates masks for the Bayer pattern for the red, green and blue pattern.

It ensures that the additional pixels at the edges of the Bayer pattern, are truncated, to match the size of the image.

Editor - demosaicImage.m | Variables - filtered_image2

+29 | p2_latex.tar.gz | demosaicImage.m | evalDemosaicing.m | runDemosaicing.m | mosaicImage.m | evalDemosaicing.m

```
%-----------------------------------------------------------------------

%                          Adaptive gradient
%-----------------------------------------------------------------------
function mosim = demosaicAdagrad(im)
 mosim = demosaicBaseline(im);
%
[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end
```

The code snippet first extracts the red, green and blue components of the image using the mask of the Bayer filter.

The green pixels at the position of the red and blue pixels are interpolated by comparing the horizontal gradient and the vertical gradient of the red and blue pixel layers, and replacing the green pixels with the appropriate averages.

After this, the red pixels are interpolated at green and blue pixel locations and similarly the blue pixels are interpolated at the blue and green pixel locations, using appropriate filters and added to the respective red and blue components of the image. The filter values are obtained by taking the average of the pixels of the neighborhood of the corresponding layers, such that the average of the pixel values is used to interpolate the value of the respective pixels.

The three layers are combined to form the image.

```
%extracting the red, green and blue components of the image using the mask
red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at the missing points
green = green_image + imfilter(green_image, [0 1 0; 1 0 1; 0 1 0]);
for x = 3:2:(imageWidth-2)
    for y = 3:2:(imageHeight-2)
        horizontal_gradient = abs((red_image(x,y-2)+red_image(x,y+2))/2 - red_image(x,y));
        vertical_gradient = abs((red_image(x-2,y)+red_image(x+2,y))/2 - red_image(x,y));
        if(horizontal_gradient<vertical_gradient)
            mosim(x,y,2)=(green(x,y-1)+green(x,y+1))/2;
        elseif (horizontal_gradient>vertical_gradient)
             mosim(x,y,2)=(green(x-1,y)+green(x+1,y))/2;
        else
            mosim(x,y,2)=(green(x-1,y)+green(x+1,y)+green(x,y-1)+green(x,y+1))/4;
        end
    end
end
```

```
for x = 4:2:(imageWidth-2)
    for y = 4:2:(imageHeight-2)
        horizontal_gradient = abs((blue_image(x,y-2)+blue_image(x,y+2))/2 - blue_image(x,y));
        vertical_gradient = abs((blue_image(x-2,y)+blue_image(x+2,y))/2 - blue_image(x,y));
        if(horizontal_gradient<vertical_gradient)
            mosim(x,y,2)=(green(x,y-1)+green(x,y+1))/2;
        elseif (horizontal_gradient>vertical_gradient)
             mosim(x,y,2)=(green(x-1,y)+green(x+1,y))/2;
        else
            mosim(x,y,2)=(green(x-1,y)+green(x+1,y)+green(x,y-1)+green(x,y+1))/4;
        end
    end
end

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

mosim(:,:,1) = red;
mosim(:,:,3) = blue;
```

Results

The figure shows the errors for the three different methods implemented, for the 10 given images. It is seen that these methods produce significantly better results the baseline model of demosaicing.

```
Command Window

>> evalDemosaicing
----------------------------------------------------------------------------------------------
#    image        baseline    nn          linear      adagrad
    ------------------------------------------------------------------------------------------
1    balloon.jpeg    0.179239    0.026872    0.016389    0.016101
2    cat.jpg      0.099966    0.027676    0.014445    0.014887
3    ip.jpg       0.231587    0.029006    0.018174    0.024435
4    puppy.jpg    0.094093    0.020216    0.008351    0.008254
5    squirrel.jpg    0.121964    0.042743    0.024781    0.024756
6    pencils.jpg     0.181449    0.030610    0.017935    0.018425
7    house.png    0.117667    0.034542    0.018978    0.018178
8    light.png    0.097868    0.032120    0.018538    0.018115
9    sails.png    0.074946    0.027171    0.015554    0.014718
10   tree.jpeg    0.167812    0.032858    0.018794    0.018224
----------------------------------------------------------------------------------------------
     average      0.136659    0.030381    0.017194    0.017610
----------------------------------------------------------------------------------------------
fx >>
```

2. Image Denoising

a) Gaussian Filtering

A Gaussian filter with a size of 3x3 was used to filter the image. The error was calculated over a range of sigmas in the range 0.1 to 2 with a step size of 0.1.

```
%% Denoising algorithm (Gaussian filtering)
for sigma = 0.1:0.1:2
    gaussian_filter=fspecial('gaussian',[3 3],sigma);
    filtered_image1=imfilter(noise1,gaussian_filter);
    filtered_image2=imfilter(noise2,gaussian_filter);
    error_1=sum(sum((im - filtered_image1).^2));
    error_2=sum(sum((im - filtered_image2).^2));
    fprintf('Sigma, Error 1, Error 2: %.2f %.2f% .2f\n', sigma, error_1,error_2);
    figure;
    imshow(filtered_image1);
    figure;
    imshow(filtered_image2);
end
```

The result for saturn1g and saturn1sp is shown below, where it is seen that sigma = 1.5 gives the minimum error for image 1 while sigma = 2 gives the minimum error for image 2.

```
Command Window
Input, Errors: 692.66 1982.10
Sigma, Error 1, Error 2: 0.10 692.66 1982.10
Sigma, Error 1, Error 2: 0.20 692.64 1982.04
Sigma, Error 1, Error 2: 0.30 673.67 1923.49
Sigma, Error 1, Error 2: 0.40 517.21 1439.69
Sigma, Error 1, Error 2: 0.50 327.15 847.74
Sigma, Error 1, Error 2: 0.60 230.03 539.03
Sigma, Error 1, Error 2: 0.70 190.39 407.91
Sigma, Error 1, Error 2: 0.80 173.84 349.80
Sigma, Error 1, Error 2: 0.90 166.45 321.63
Sigma, Error 1, Error 2: 1.00 162.95 306.81
Sigma, Error 1, Error 2: 1.10 161.25 298.47
Sigma, Error 1, Error 2: 1.20 160.41 293.53
Sigma, Error 1, Error 2: 1.30 160.02 290.47
Sigma, Error 1, Error 2: 1.40 159.87 288.52
Sigma, Error 1, Error 2: 1.50 159.84 287.23
Sigma, Error 1, Error 2: 1.60 159.88 286.37
Sigma, Error 1, Error 2: 1.70 159.95 285.78
Sigma, Error 1, Error 2: 1.80 160.04 285.37
Sigma, Error 1, Error 2: 1.90 160.13 285.09
Sigma, Error 1, Error 2: 2.00 160.22 284.89
```

The result for saturn2g and saturn2sp is shown below, where it is seen that sigma=2 gives the minimum error for both images.

```
Command Window
>> evalDenoising
Input, Errors: 3118.10 3829.28
Sigma, Error 1, Error 2: 0.10 3118.10 3829.28
Sigma, Error 1, Error 2: 0.20 3118.02 3829.17
Sigma, Error 1, Error 2: 0.30 3035.40 3718.08
Sigma, Error 1, Error 2: 0.40 2352.06 2799.31
Sigma, Error 1, Error 2: 0.50 1513.72 1671.84
Sigma, Error 1, Error 2: 0.60 1074.49 1080.74
Sigma, Error 1, Error 2: 0.70 886.96 828.04
Sigma, Error 1, Error 2: 0.80 803.42 715.24
Sigma, Error 1, Error 2: 0.90 762.72 660.13
Sigma, Error 1, Error 2: 1.00 741.19 630.87
Sigma, Error 1, Error 2: 1.10 729.01 614.25
Sigma, Error 1, Error 2: 1.20 721.74 604.29
Sigma, Error 1, Error 2: 1.30 717.22 598.05
Sigma, Error 1, Error 2: 1.40 714.30 594.00
Sigma, Error 1, Error 2: 1.50 712.36 591.29
Sigma, Error 1, Error 2: 1.60 711.04 589.44
Sigma, Error 1, Error 2: 1.70 710.13 588.15
Sigma, Error 1, Error 2: 1.80 709.49 587.22
Sigma, Error 1, Error 2: 1.90 709.04 586.56
Sigma, Error 1, Error 2: 2.00 708.71 586.07
```

b) Median Filtering

A median filter of various sizes in the range 1x1 to 5x5 was applied to the image, and the error for every size was calculated.

```
%% Denoising algorithm (Median filtering)
for m=1:5
    for n=1:5
        filtered_image3=medfilt2(noise1,[m n]);
        filtered_image4=medfilt2(noise2,[m n]);
        error_3=sum(sum((im - filtered_image3).^2));
        error_4=sum(sum((im - filtered_image4).^2));
        fprintf('Neighborhood Length,Neighborhood Width, Error 1, Error2: %.2f %.2f %.2f% .2f\n'
        figure;
        imshow(filtered_image3);
        figure;
        imshow(filtered_image4);
    end
end
```

It is seen from the output of saturn1g and saturn1sp shown below that, for a window size of 5*5, the error for the first image is minimum and for a patch size of 3*3, the error for the second image is minimum.

Command Window

```
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 1.00 692.66 1982.10
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 2.00 401.15 1036.15
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 3.00 330.38 171.06
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 4.00 249.92 113.77
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 5.00 228.91 25.80
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 1.00 409.90 1038.08
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 2.00 268.62 138.94
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 3.00 194.99 46.37
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 4.00 177.92 56.58
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 5.00 149.45 42.12
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 1.00 337.92 170.53
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 2.00 191.61 42.02
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 3.00 140.10 8.78
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 4.00 124.81 33.72
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 5.00 103.21 13.04
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 1.00 265.82 123.78
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 2.00 184.58 59.10
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 3.00 135.24 40.28
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 4.00 131.03 58.64
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 5.00 110.24 42.44
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 1.00 248.94 37.60
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 2.00 156.55 42.64
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 3.00 114.65 16.93
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 4.00 111.39 41.20
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 5.00 91.99 19.14
```

It is seen from the output of saturn2g and saturn2sp shown below that, for a window size of 5*5, the error for the first image is minimum and for a patch size of 3*3, the error for the second image is minimum.

Command Window

```
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 1.00 3118.10 3829.28
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 2.00 1774.52 2006.83
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 3.00 1501.92 588.17
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 4.00 1067.96 349.06
Neighborhood Length,Neighborhood Width, Error 1, Error2: 1.00 5.00 1009.07 114.94
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 1.00 1771.96 2022.16
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 2.00 1081.48 358.80
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 3.00 787.15 97.89
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 4.00 649.61 70.63
Neighborhood Length,Neighborhood Width, Error 1, Error2: 2.00 5.00 544.01 53.63
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 1.00 1508.59 575.92
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 2.00 786.14 86.84
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 3.00 607.87 18.57
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 4.00 465.09 40.96
Neighborhood Length,Neighborhood Width, Error 1, Error2: 3.00 5.00 407.90 20.69
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 1.00 1078.95 345.44
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 2.00 653.38 73.16
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 3.00 472.08 47.65
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 4.00 397.44 67.55
Neighborhood Length,Neighborhood Width, Error 1, Error2: 4.00 5.00 336.20 50.32
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 1.00 1022.67 130.62
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 2.00 554.46 51.39
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 3.00 418.22 24.63
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 4.00 338.60 48.78
Neighborhood Length,Neighborhood Width, Error 1, Error2: 5.00 5.00 292.39 25.26
```

c) Non-local Means Filtering

The non-local means filter takes the mean of all the neighboring pixels of an image in a window, weighted by how similar the pixels are to the target pixel.

The function shown below, implements a non-linear filter kernel, where the weight of the pixel in a patch reduces by a factor of $(2*distance+1)^2$. Therefore, the center pixel is weighed the highest, and as we move away from the center, the pixel weights are reduced by this factor.

```
function [means_kernel] = nl_means(patch_size)
means_kernel = zeros(2*patch_size+1, 2*patch_size+1);
for distance = 1:patch_size
    weight = (1/(2*distance+1))^2;
    for x=-distance:distance
        for y=-distance:distance
            means_kernel(patch_size+1-x,patch_size+1-y)=means_kernel(patch_size+1-x,patch_size+1-y)+weight;
        end
    end
end
means_kernel = means_kernel./patch_size;
end
```

The function implemented below, denoises the image using the non-local means algorithm described in the work of Buades. et al [1]. In this, first, a non-linear kernel of a size equal to the patch size is created. Then two patches around the two target pixels are created in a window, taking appropriate limits for the boundaries of the window.
After that, the sum of squared differences between the two patches is calculated, weighted by the non-linear kernel.
A Gaussian weighing function [2], defined by

$$weight = e^{\frac{-|P(x)-P(y)|^2}{\gamma^2}}$$

was calculated, and was multiplied to the image pixel at each point in the patch and was stored as an cumulative sum. The denoised image was this cumulative sum divided by the cumulative sum of the weighing factors.

```matlab
function [denoised_image] = nl_mean_out(image_in, window_size, patch_size, gamma)

[imageWidth, imageHeight]=size(image_in);
denoised_image = zeros(imageWidth,imageHeight);
image_in = padarray(image_in,[patch_size,patch_size],'circular');

nl_kernel = nl_means(patch_size);
nl_kernel = nl_kernel/sum(sum(nl_kernel));

squared_gamma = gamma*gamma;

for x = 1:imageWidth
    for y = 1:imageHeight

        x_new = x+patch_size;
        y_new = y+patch_size;

        wind_1 = image_in(x_new-patch_size:x_new+patch_size,y_new-patch_size:y_new+patch_size);

        wind_max = 0;
        ave = 0;
        s_cum = 0;

        min_r = max(x_new-window_size,patch_size+1);
        max_r = min(x_new+window_size, imageWidth+patch_size);
        min_s = max(y_new-window_size, patch_size+1);
        max_s = min(y_new+window_size, imageHeight+patch_size);

        for i = min_r:max_r
            for j = min_s:max_s

                if(i==x_new && j==y_new)
                    continue;
                end

                wind_2 = image_in(i-patch_size:i+patch_size, j-patch_size:j+patch_size);

                diff = sum(sum(nl_kernel.*(wind_1-wind_2).*(wind_1-wind_2)));

                weigh_param = exp(-diff/squared_gamma);

                if weigh_param > wind_max
                    wind_max = weigh_param;
                end

                s_cum = s_cum+weigh_param;
                ave = ave +weigh_param*image_in(i,j);
            end
        end

        ave = ave + wind_max*image_in(x_new,y_new);
        s_cum = s_cum + wind_max;

        if s_cum > 0
            denoised_image(x,y) =ave/s_cum;
        else
            denoised_image(x,y) = image_in(x,y);
        end
    end
end
figure;
imshow(denoised_image);
end
```

In the main function, the function which performs non-local means, defined above is called, with different values of window size, patch size and gamma. It is seen that the following values of window, size, patch size and gamma produce the least error for the two noisy images.

```
%% Denoising alogirthm (Non-local means)

filtered_image5 = nl_mean_out(noise1, 8, 2, 0.1);
filtered_image6 = nl_mean_out(noise2, 4, 2, 0.25);
error_5=sum(sum((im - filtered_image5).^2));
error_6=sum(sum((im - filtered_image6).^2));
fprintf('Error 1, Error2: %.2f %.2f\n', error_5,error_6);
```

Error for saturn1g and saturn1sp

```
Command Window
>> evalDenoising
Input, Errors: 692.66 1982.10
Error 1, Error2: 97.17 123.96
```

Error for saturn2g and saturn2sp

```
Command Window
Error 1, Error2: 946.26 262.98
>>
```

Qualitative Comparison of the Denoising Algorithm

The table shown below compares the minimum noises obtained by the various algorithms for the given images.

Algorithm and Parameters	**Error for Saturn1g**		**Error for Saturn1sp**	
Gaussian filter	Sigma	Error	Sigma	Error
	1.5	159.84	2	284.89
Median filter	Window Size	Error	Window Size	Error
	5*5	8.78	3*3	91.19
Non local means filter	Window Size Patch Size Gamma	Error	Widow Size Patch Size Gamma	Error
	8 2 0.1	97.17	4 2 0.25	123.96

Algorithm and Parameters	**Error for Saturn2g**		**Error for Saturn2sp**	
Gaussian filter	Sigma	Error	Sigma	Error
	2	708.71	2	586.07
Median filter	Window Size	Error	Window Size	Error
	5*5	292.39	3*3	18.57
Non local means filter	Window Size Patch Size Gamma	Error	Widow Size Patch Size Gamma	Error
	8 2 0.25	514.91	4 2 0.25	262.98

It is seen that, median filtering produces the best results for all the images, because, median filtering handles the outliers in a more efficient way.

3. Extension 1

a) Linear Transformation of Color Spaces

The implementation of this is similar to the one done in Color Demosaicing (Question 1) apart from the transformation of color spaces.

Editor - C:\Users\User\Documents\UMass Amherst\Semester 1\COMPSCI 670 - Computer Vision\Mini Projects\2\p2_code\code\demosaicImage.m

prepareData.m | alignChannels.m | HW3.m | convolution.m | run.m | first.m | p2_latex.tar.gz | demosaicImage.m

```
%--------------------------------------------------------------------

%                              Transformed Color Spaces - Linear
%--------------------------------------------------------------------
function mosim = demosaicTransformation(im)
 %mosim = demosaicBaseline(im);

[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end
```

The red and blue channels are divided by the green channels, after replacing the green pixels with value zero with the next minimum value in the green matrix.

Editor - C:\Users\User\Documents\UMass Amherst\Semester 1\COMPSCI 670 - Computer Vision\Mini Projects\2\p2_code\code\demosaicImage.m

prepareData.m | alignChannels.m | HW3.m | convolution.m | run.m | first.m | p2_latex.tar.gz | demosaicImage.m

```
%extracting the red, green and blue components of the image using the mask

red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at missing points
green = green_image + imfilter(green_image, [0 1 0;1 0 1; 0 1 0]/4);

%checking for pixel values to be zero and correcting it
min_green = min(min(green(green~=0)));
green(green==0) = min_green;

%transforming the images
red_image = red_image./green;
red_image = red_image.*bayer_red;

blue_image = blue_image./green;
blue_image = blue_image.*bayer_blue;
```

After demosaicing, the color spaces are transformed back to the original color space by applying the inverse transformation.

```
%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

%applying inverse transformation
mosim(:,:,1) = red.*green;
mosim(:,:,2) = green;
mosim(:,:,3) = blue.*green;
```

b) Logarithmic Transformation of Color Spaces

The implementation of this is similar to the one done in Color Demosaicing (Question 1) apart from the transformation of color spaces.

```
%--------------------------------------------------------------------------

%                         Transformed Color Spaces - Logarithmic
%--------------------------------------------------------------------------
function mosim = demosaicLogTransformation(im)
 %mosim = demosaicBaseline(im);

[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end

```

The red and blue channels are divided by the green channels and the logarithm of the ratio is taken. In this case, none of the channel values can be zero, so the pixels with a value of zero are replaced with the next minimum value.

After demosaicing, the color spaces are transformed back to the original color space by applying the inverse transformation.

Editor - C:\Users\User\Documents\UMass Amherst\Semester 1\COMPSCI 670 - Computer Vision\Mini Projects\2\p2_code\code\demosaicImage.m

prepareData.m | alignChannels.m | HW3.m | convolution.m | run.m | first.m | p2_latex.tar.gz | demosaicImage.m

```matlab
%extracting the red, green and blue components of the image using the mask

green_image = im.*bayer_green;
blue_image = im.*bayer_blue;
red_image = im.*bayer_red;

%deducing the green pixels at missing points
green = green_image + imfilter(green_image, [0 1 0;1 0 1; 0 1 0]/4);

%checking for pixel values to be zero and correcting it
min_green = min(min(green(green~=0)));
green(green==0) = min_green;

min_blue = min(min(blue_image(blue_image~=0)));
blue_image(blue_image==0) = min_blue;

min_red = min(min(red_image(red_image~=0)));
red_image(red_image==0) = min_red;

%transforming the images
red_image = log(red_image./green);
red_image = red_image.*bayer_red;

blue_image = log(blue_image./green);
blue_image = blue_image.*bayer_blue;

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

%applying the inverse transformation
mosim(:,:,1) = exp(red).*green;
mosim(:,:,2) = green;
mosim(:,:,3) = exp(blue).*green;
```

Results

Command Window

```
>> evalDemosaicing
```

#	image	baseline	nn	linear	adagrad	linear_trans	log_trans
1	balloon.jpeg	0.179239	0.026872	0.016389	0.016101	0.026058	0.021362
2	cat.jpg	0.099966	0.027676	0.014445	0.014887	0.012443	0.011917
3	ip.jpg	0.231587	0.029006	0.018174	0.024435	0.018852	0.018668
4	puppy.jpg	0.094093	0.020216	0.008351	0.008254	0.008699	0.007259
5	squirrel.jpg	0.121964	0.042743	0.024781	0.024756	0.019980	0.019200
6	pencils.jpg	0.181449	0.030610	0.017935	0.018425	0.019102	0.019217
7	house.png	0.117667	0.034542	0.018978	0.018178	0.016336	0.015066
8	light.png	0.097868	0.032120	0.018538	0.018115	0.016069	0.015438
9	sails.png	0.074946	0.027171	0.015554	0.014718	0.014198	0.012822
10	tree.jpeg	0.167812	0.032858	0.018794	0.018224	0.018014	0.016034
	average	0.136659	0.030381	0.017194	0.017610	0.016975	0.015698

From the results shown in the figure above, we see that linear and logarithmic transformation of color spaces do produce significantly better results than other methods. So, it can be a good idea to implement color space transformation to images, if the computational complexity of computing the transformation (dividing and taking logarithms and their inverses) is not of a major concern to the system it is being developed for.

References

[1]. Buades, Antoni (20–25 June 2005). "A non-local algorithm for image denoising". *Computer Vision and Pattern Recognition, 2005*. **2**: 60–65. doi:10.1109/CVPR.2005.38

[2] https://en.wikipedia.org/wiki/Non-local_means

Appendix 1: Code for Color Image Demosaicing with Extension 1

```
function output = demosaicImage(im, method)
% DEMOSAICIMAGE computes the color image from mosaiced
input
%   OUTPUT = DEMOSAICIMAGE(IM, METHOD) computes a
demosaiced OUTPUT from
%   the input IM. The choice of the interpolation METHOD
can be
%   'baseline', 'nn', 'linear', 'adagrad'.
%
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%

switch lower(method)
    case 'baseline'
        output = demosaicBaseline(im);
    case 'nn'
        output = demosaicNN(im);         % Implement this
    case 'linear'
        output = demosaicLinear(im);     % Implement this
    case 'adagrad'
        output = demosaicAdagrad(im);    % Implement this
    case 'transformation'
        output = demosaicTransformation(im); % Extra
Implementation
    case 'log_transformation'
        output = demosaicLogTransformation(im); % Extra
Implementation
end

%----------------------------------------------------------
------------------
%                          Baseline demosacing algorithm.
%                          The algorithm replaces missing
values with the
%                          mean of each color channel.
%----------------------------------------------------------
------------------
function mosim = demosaicBaseline(im)
```

```
mosim = repmat(im, [1 1 3]); % Create an image by
stacking the input
[imageHeight, imageWidth] = size(im);

% Red channel (odd rows and columns);
redValues = im(1:2:imageHeight, 1:2:imageWidth);
meanValue = mean(mean(redValues));
mosim(:,:,1) = meanValue;
mosim(1:2:imageHeight, 1:2:imageWidth,1) =
im(1:2:imageHeight, 1:2:imageWidth);

% Blue channel (even rows and colums);
blueValues = im(2:2:imageHeight, 2:2:imageWidth);
meanValue = mean(mean(blueValues));
mosim(:,:,3) = meanValue;
mosim(2:2:imageHeight, 2:2:imageWidth,3) =
im(2:2:imageHeight, 2:2:imageWidth);

% Green channel (remaining places)
% We will first create a mask for the green pixels (+1
green, -1 not green)
mask = ones(imageHeight, imageWidth);
mask(1:2:imageHeight, 1:2:imageWidth) = -1;
mask(2:2:imageHeight, 2:2:imageWidth) = -1;
greenValues = mosim(mask > 0);
meanValue = mean(greenValues);
% For the green pixels we copy the value
greenChannel = im;
greenChannel(mask < 0) = meanValue;
mosim(:,:,2) = greenChannel;

%----------------------------------------------------------
------------------
%                            Nearest neighbour algorithm
%----------------------------------------------------------
------------------
function mosim = demosaicNN(im)
%mosim = demosaicBaseline(im);

[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1],
ceil(imageWidth/2),ceil(imageHeight/2));
```

```
bayer_green = repmat([0 1; 1 0],
ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end

%extracting the red, green and blue components of the
image using the mask

red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at missing points
green = green_image+imfilter(green_image,[0 1]);

%deducing the red pixels at missing points
redValue=im(1:2:imageWidth,1:2:imageHeight);
meanRed = mean(mean(redValue));
%red@blue
red_1 = imfilter(red_image, [0 0;0 1], meanRed);
%red@green
red_2 = imfilter(red_image, [0 1;1 0], meanRed);
%combine
red = red_image + red_1 +red_2;

%deducing the blue pixels at missing points
blueValue=im(1:2:imageWidth,1:2:imageHeight);
meanBlue = mean(mean(blueValue));
%blue@red
blue_1 = imfilter(blue_image, [0 0;0 1], meanBlue);
%blue@green
blue_2 = imfilter(blue_image, [0 1;1 0], meanBlue);
%combine
blue = blue_image + blue_1 +blue_2;

mosim(:,:,1) = red;
mosim(:,:,2) = green;
```

```
mosim(:,:,3) = blue;
%-----------------------------------------------------------
------------------
%                            Linear interpolation
%-----------------------------------------------------------
------------------
function mosim = demosaicLinear(im)
% mosim = demosaicBaseline(im);
% mosim = repmat(im, [1 1 3]);
%
[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0],
ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end

%extracting the red, green and blue components of the
image using the mask
red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at missing points
green = green_image + imfilter(green_image, [0 1 0;1 0 1;
0 1 0]/4);

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;
```

```
%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

mosim(:,:,1) = red;
mosim(:,:,2) = green;
mosim(:,:,3) = blue;
%----------------------------------------------------------
------------------

%                            Adaptive gradient
%----------------------------------------------------------
------------------
function mosim = demosaicAdagrad(im)
 mosim = demosaicBaseline(im);
%
[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0],
ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end

%extracting the red, green and blue components of the
image using the mask
red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at the missing points
```

```
green = green_image + imfilter(green_image, [0 1 0; 1 0
1; 0 1 0]);
for x = 3:2:(imageWidth-2)
    for y = 3:2:(imageHeight-2)
        horizontal_gradient = abs((red_image(x,y-
2)+red_image(x,y+2))/2 - red_image(x,y));
        vertical_gradient = abs((red_image(x-
2,y)+red_image(x+2,y))/2 - red_image(x,y));
        if(horizontal_gradient<vertical_gradient)
            mosim(x,y,2)=(green(x,y-1)+green(x,y+1))/2;
        elseif (horizontal_gradient>vertical_gradient)
             mosim(x,y,2)=(green(x-1,y)+green(x+1,y))/2;
        else
            mosim(x,y,2)=(green(x-
1,y)+green(x+1,y)+green(x,y-1)+green(x,y+1))/4;
        end
    end
end
for x = 4:2:(imageWidth-2)
    for y = 4:2:(imageHeight-2)
        horizontal_gradient = abs((blue_image(x,y-
2)+blue_image(x,y+2))/2 - blue_image(x,y));
        vertical_gradient = abs((blue_image(x-
2,y)+blue_image(x+2,y))/2 - blue_image(x,y));
        if(horizontal_gradient<vertical_gradient)
            mosim(x,y,2)=(green(x,y-1)+green(x,y+1))/2;
        elseif (horizontal_gradient>vertical_gradient)
             mosim(x,y,2)=(green(x-1,y)+green(x+1,y))/2;
        else
            mosim(x,y,2)=(green(x-
1,y)+green(x+1,y)+green(x,y-1)+green(x,y+1))/4;
        end
    end
end

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

mosim(:,:,1) = red;
mosim(:,:,3) = blue;
```

```
%------------------------------------------------------------
-------------------

%             Transformed Color Spaces - Linear
%------------------------------------------------------------
-------------------
function mosim = demosaicTransformation(im)
 %mosim = demosaicBaseline(im);

[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0],
ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
end

%extracting the red, green and blue components of the
image using the mask

red_image = im.*bayer_red;
blue_image = im.*bayer_blue;
green_image = im.*bayer_green;

%deducing the green pixels at missing points
green = green_image + imfilter(green_image, [0 1 0;1 0 1;
0 1 0]/4);

%checking for pixel values to be zero and correcting it
min_green = min(min(green(green~=0)));
green(green==0) = min_green;

%transforming the images
red_image = red_image./green;
```

```
red_image = red_image.*bayer_red;

blue_image = blue_image./green;
blue_image = blue_image.*bayer_blue;

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

%applying inverse transformation
mosim(:,:,1) = red.*green;
mosim(:,:,2) = green;
mosim(:,:,3) = blue.*green;
%---------------------------------------------------------
------------------

%           Transformed Color Spaces - Logarithmic
%---------------------------------------------------------
------------------
function mosim = demosaicLogTransformation(im)
 %mosim = demosaicBaseline(im);

[imageWidth,imageHeight]=size(im);

%creating masks for the bayer pattern
bayer_red = repmat([1 0; 0 0],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_blue = repmat([0 0; 0 1],
ceil(imageWidth/2),ceil(imageHeight/2));
bayer_green = repmat([0 1; 1 0],
ceil(imageWidth/2),ceil(imageHeight/2));

%truncating the extra pixels at the edges
if(mod(imageWidth,2))==1
   bayer_red(size(bayer_red,1),:)=[];
   bayer_blue(size(bayer_blue,1),:)=[];
   bayer_green(size(bayer_green,1),:)=[];
end
if(mod(imageHeight,2)==1)
   bayer_red(:,size(bayer_red,2))=[];
   bayer_blue(:,size(bayer_blue,2))=[];
   bayer_green(:,size(bayer_green,2))=[];
```

```
end

%extracting the red, green and blue components of the
image using the mask

green_image = im.*bayer_green;
blue_image = im.*bayer_blue;
red_image = im.*bayer_red;

%deducing the green pixels at missing points
green = green_image + imfilter(green_image, [0 1 0;1 0 1;
0 1 0]/4);

%checking for pixel values to be zero and correcting it
min_green = min(min(green(green~=0)));
green(green==0) = min_green;

min_blue = min(min(blue_image(blue_image~=0)));
blue_image(blue_image==0) = min_blue;

min_red = min(min(red_image(red_image~=0)));
red_image(red_image==0) = min_red;

%transforming the images
red_image = log(red_image./green);
red_image = red_image.*bayer_red;

blue_image = log(blue_image./green);
blue_image = blue_image.*bayer_blue;

%deducing the red pixels at missing points
red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
red = red_image+red_1+red_2;

%deducing the blue pixels at missing points
blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
blue = blue_image+blue_1+blue_2;

%applying the inverse transformation
mosim(:,:,1) = exp(red).*green;
mosim(:,:,2) = green;
mosim(:,:,3) = exp(blue).*green;
```

```
% Entry code for evaluating demosaicing algorithms
% The code loops over all images and methods, computes
the error and
% displays them in a table.
%
%
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%
% Path to your data directory
dataDir = fullfile('C:','Users','User','Documents','Umass
Amherst', 'Semester 1', 'COMPSCI 670 - Computer
Vision','Mini Projects','2','p2_data','data','demosaic');

% Path to your output directory
outDir = fullfile('C:','Users','User','Documents','Umass
Amherst', 'Semester 1', 'COMPSCI 670 - Computer
Vision','Mini
Projects','2','p2_code','output','demosaic');
if ~exist(outDir, 'file')
    mkdir(outDir);
end

% List of images
imageNames = {'balloon.jpeg',    'cat.jpg',
'ip.jpg','puppy.jpg','squirrel.jpg', ...
              'pencils.jpg',     'house.png', 'light.png',
'sails.png', 'tree.jpeg'};
numImages = length(imageNames);

% List of methods you have to implement
methods = {'baseline', 'nn', 'linear', 'adagrad',
'transformation','log_transformation'};
numMethods = length(methods);

% Global variables
display = false;
error = zeros(numImages, numMethods);

% Loop over methods and print results
fprintf([repmat('-',[1 100]),'\n']);
fprintf('# \t image \t\t baseline \t nn \t\t linear \t
adagrad \t linear_trans \t log_trans\n');
fprintf([repmat('-',[1 100]),'\n']);
```

```
for i = 1:numImages,
    fprintf('%i \t %s ', i, imageNames{i});
    for j = 1:numMethods,
        thisImage = fullfile(dataDir, imageNames{i});
        thisMethod = methods{j};
        [error(i,j), colorIm] = runDemosaicing(thisImage,
thisMethod, display);
        fprintf('\t %f ', error(i,j));

        % Write the output
        outfileName = fullfile(outDir,
[imageNames{i}(1:end-5) '-' thisMethod '-dmsc.jpg']);
        imwrite(colorIm, outfileName);

    end
    fprintf('\n');
end

% Compute average errors
fprintf([repmat('-',[1 100]),'\n']);
fprintf(' \t %s ', 'average');
for j = 1:numMethods,
        fprintf('\t %f ', mean(error(:,j)));
end
fprintf('\n');
fprintf([repmat('-',[1 100]),'\n']);
```

Appendix 2: Code for Image Denoising

```
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%
% Load images
im = im2double(imread('C:/Users/User/Documents/UMass
Amherst/Semester 1/COMPSCI 670 - Computer Vision/Mini
Projects/2/p2_data/data/denoising/saturn.png'));
noise1 = im2double(imread('C:/Users/User/Documents/UMass
Amherst/Semester 1/COMPSCI 670 - Computer Vision/Mini
Projects/2/p2_data/data/denoising/saturn-noise2g.png'));
noise2 = im2double(imread('C:/Users/User/Documents/UMass
Amherst/Semester 1/COMPSCI 670 - Computer Vision/Mini
Projects/2/p2_data/data/denoising/saturn-noise2sp.png'));

% Compute errors
error1 = sum(sum((im - noise1).^2));
error2 = sum(sum((im - noise2).^2));
fprintf('Input, Errors: %.2f %.2f\n', error1, error2)

% Display the images
figure(1);
subplot(1,3,1); imshow(im); title('Input');
subplot(1,3,2); imshow(noise1); title(sprintf('SE %.2f',
error1));
subplot(1,3,3); imshow(noise2); title(sprintf('SE %.2f',
error2));

%% Denoising algorithm (Gaussian filtering)
for sigma = 0.1:0.1:2
    gaussian_filter=fspecial('gaussian',[3 3],sigma);
    filtered_image1=imfilter(noise1,gaussian_filter);
    filtered_image2=imfilter(noise2,gaussian_filter);
    error_1=sum(sum((im - filtered_image1).^2));
    error_2=sum(sum((im - filtered_image2).^2));
    fprintf('Sigma, Error 1, Error 2: %.2f %.2f% .2f\n',
sigma, error_1,error_2);
%     figure;
%     imshow(filtered_image1);
%     figure;
%     imshow(filtered_image2);
end
%% Denoising algorithm (Median filtering)
```

```
for m=1:5
    for n=1:5
        filtered_image3=medfilt2(noise1,[m n]);
        filtered_image4=medfilt2(noise2,[m n]);
        error_3=sum(sum((im - filtered_image3).^2));
        error_4=sum(sum((im - filtered_image4).^2));
        fprintf('Neighborhood Length,Neighborhood Width,
Error 1, Error2: %.2f %.2f %.2f% .2f\n', m, n,
error_3,error_4);
%          figure;
%          imshow(filtered_image3);
%          figure;
%          imshow(filtered_image4);
    end
end
%% Denoising alogirthm (Non-local means)

filtered_image5 = nl_mean_out(noise1, 10, 2, 0.5);
filtered_image6 = nl_mean_out(noise2, 4, 2, 0.25);
error_5=sum(sum((im - filtered_image5).^2));
error_6=sum(sum((im - filtered_image6).^2));
fprintf('Error 1, Error2: %.2f %.2f\n', error_5,error_6);


%% Function to build the non-linear kernel

function [means_kernel] = nl_means(patch_size)
means_kernel = zeros(2*patch_size+1, 2*patch_size+1);
for distance = 1:patch_size
    weight = (1/(2*distance+1))^2;
    for x=-distance:distance
        for y=-distance:distance
            means_kernel(patch_size+1-x,patch_size+1-
y)=means_kernel(patch_size+1-x,patch_size+1-y)+weight;
        end
    end
end
means_kernel = means_kernel./patch_size;
end

%% Function to perform non local means denoising

function [denoised_image] = nl_mean_out(image_in,
window_size, patch_size, gamma)

[imageWidth, imageHeight]=size(image_in);
denoised_image = zeros(imageWidth,imageHeight);
```

```
image_in =
padarray(image_in,[patch_size,patch_size],'circular');

nl_kernel = nl_means(patch_size);
nl_kernel = nl_kernel/sum(sum(nl_kernel));

squared_gamma = gamma*gamma;

for x = 1:imageWidth
    for y = 1:imageHeight

        x_new = x+patch_size;
        y_new = y+patch_size;

        wind_1 = image_in(x_new-
patch_size:x_new+patch_size,y_new-
patch_size:y_new+patch_size);

        wind_max = 0;
        ave = 0;
        s_cum = 0;

        min_r = max(x_new-window_size,patch_size+1);
        max_r = min(x_new+window_size,
imageWidth+patch_size);
        min_s = max(y_new-window_size, patch_size+1);
        max_s = min(y_new+window_size,
imageHeight+patch_size);

        for i = min_r:max_r
            for j = min_s:max_s

                if(i==x_new && j==y_new)
                    continue;
                end

                wind_2 = image_in(i-
patch_size:i+patch_size, j-patch_size:j+patch_size);

                diff = sum(sum(nl_kernel.*(wind_1-
wind_2).*(wind_1-wind_2)));

                weigh_param = exp(-diff/squared_gamma);

                if weigh_param > wind_max
                    wind_max = weigh_param;
                end
```

```
                s_cum = s_cum+weigh_param;
                ave = ave +weigh_param*image_in(i,j);
            end
        end

        ave = ave + wind_max*image_in(x_new,y_new);
        s_cum = s_cum + wind_max;

        if s_cum > 0
            denoised_image(x,y) =ave/s_cum;
        else
            denoised_image(x,y) = image_in(x,y);
        end
    end
end
figure;
imshow(denoised_image);
end
```