

Mini-Project 2

1. Color Image Demosaicing

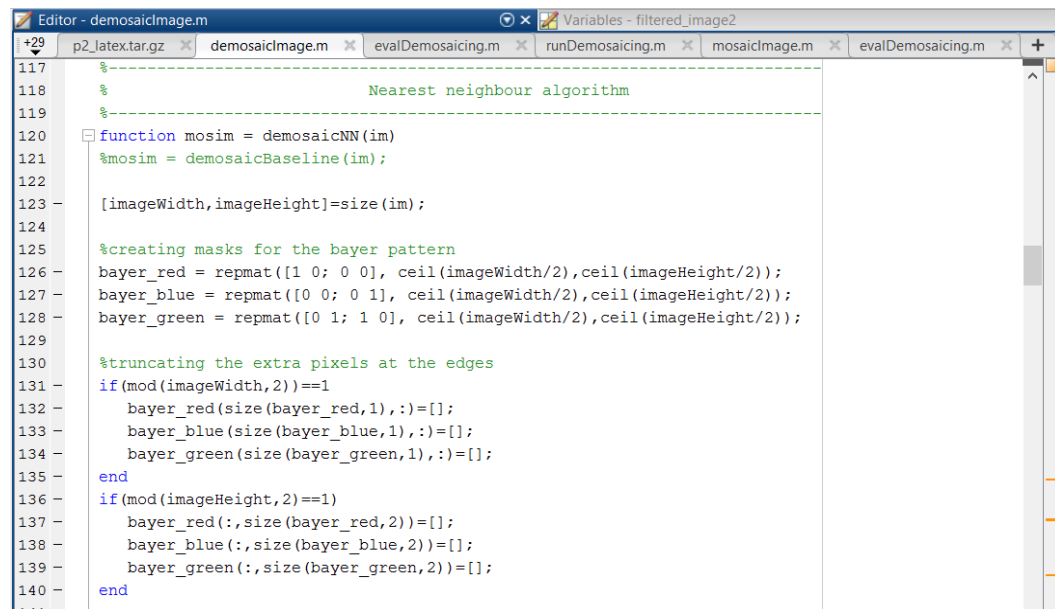
Implementation of DemosaicImage.m

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.



```
117 %-----
118 %                               Nearest neighbour algorithm
119 %-----
120 function mosim = demosaicNN(im)
121 %mosim = demosaicBaseline(im);
122
123 [imageWidth,imageHeight]=size(im);
124
125 %creating masks for the bayer pattern
126 bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
127 bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
128 bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));
129
130 %truncating the extra pixels at the edges
131 if(mod(imageWidth,2)==1
132     bayer_red(size(bayer_red,1),:)=[];
133     bayer_blue(size(bayer_blue,1),:)=[];
134     bayer_green(size(bayer_green,1),:)=[];
135 end
136 if(mod(imageHeight,2)==1)
137     bayer_red(:,size(bayer_red,2))=[];
138     bayer_blue(:,size(bayer_blue,2))=[];
139     bayer_green(:,size(bayer_green,2))=[];
140 end
141
```

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.

```

Editor - demosaicImage.m
p2_latex.tar.gz x demosaicImage.m x evalDemosaicing.m x runDemosaicing.m x mosaicImage.m x evalDemosaicing.m x +
141
142 %extracting the red, green and blue components of the image using the mask
143
144 red_image = im.*bayer_red;
145 blue_image = im.*bayer_blue;
146 green_image = im.*bayer_green;
147
148 %deducing the green pixels at missing points
149 green = green_image+imfilter(green_image,[0 1]);
150
151 %deducing the red pixels at missing points
152 redValue=im(1:2:imageWidth,1:2:imageHeight);
153 meanRed = mean(mean(redValue));
154 %red@blue
155 red_1 = imfilter(red_image, [0 0;0 1], meanRed);
156 %red@green
157 red_2 = imfilter(red_image, [0 1;1 0], meanRed);
158 %combine
159 red = red_image + red_1 +red_2;
160
161 %deducing the blue pixels at missing points
162 blueValue=im(1:2:imageWidth,1:2:imageHeight);
163 meanBlue = mean(mean(blueValue));
164 %blue@red
165 blue_1 = imfilter(blue_image, [0 0;0 1], meanBlue);
166 %blue@green
167 blue_2 = imfilter(blue_image, [0 1;1 0], meanBlue);
168 %combine
169 blue = blue_image + blue_1 +blue_2;
170
171 mosim(:,:,1) = red;
172 mosim(:,:,2) = green;
173 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.

```

Editor - demosaicImage.m
p2_latex.tar.gz x demosaicImage.m x evalDemosaicing.m x runDemosaicing.m x mosaicImage.m x evalDemosaicing.m x +
174
175 %----- Linear interpolation -----
176
177 function mosim = demosaicLinear(im)
178 % mosim = demosaicBaseline(im);
179 % mosim = repmat(im, [1 1 3]);
180
181 [imageWidth,imageHeight]=size(im);
182
183 %creating masks for the bayer pattern
184 bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
185 bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
186 bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));
187
188 %truncating the extra pixels at the edges
189 if(mod(imageWidth,2))==1
190     bayer_red(size(bayer_red,1),:)=[];
191     bayer_blue(size(bayer_blue,1),:)=[];
192     bayer_green(size(bayer_green,1),:)=[];
193 end
194 if(mod(imageHeight,2))==1
195     bayer_red(:,size(bayer_red,2))=[];
196     bayer_blue(:,size(bayer_blue,2))=[];
197     bayer_green(:,size(bayer_green,2))=[];
198 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.

```
199
200 %extracting the red, green and blue components of the image using the mask
201 - red_image = im.*bayer_red;
202 - blue_image = im.*bayer_blue;
203 - green_image = im.*bayer_green;
204
205 %deducing the green pixels at missing points
206 - green = green_image + imfilter(green_image, [0 1 0;1 0 1; 0 1 0]/4);
207
208 %deducing the red pixels at missing points
209 - red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
210 - red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
211 - red = red_image+red_1+red_2;
212
213 %deducing the blue pixels at missing points
214 - blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
215 - blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
216 - blue = blue_image+blue_1+blue_2;
217
218 - mosim(:,:,1) = red;
219 - mosim(:,:,2) = green;
220 - 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
p2_latex.tar.gz x demosaicImage.m x evalDemosaiing.m x runDemosaiing.m x mosaicImage.m x evalDemosaiing.m x
221 %-----
222
223 % Adaptive gradient
224 %-----
225 function mosim = demosaicAdagrad(im)
226     mosim = demosaicBaseline(im);
227
228     [imageWidth,imageHeight]=size(im);
229
230     %creating masks for the bayer pattern
231     bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
232     bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
233     bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));
234
235     %truncating the extra pixels at the edges
236     if(mod(imageWidth,2)==1
237         bayer_red(size(bayer_red,1),:)=[];
238         bayer_blue(size(bayer_blue,1),:)=[];
239         bayer_green(size(bayer_green,1),:)=[];
240     end
241     if(mod(imageHeight,2)==1
242         bayer_red(:,size(bayer_red,2))=[];
243         bayer_blue(:,size(bayer_blue,2))=[];
244         bayer_green(:,size(bayer_green,2))=[];
245     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.

```

247 %extracting the red, green and blue components of the image using the mask
248 red_image = im.*bayer_red;
249 blue_image = im.*bayer_blue;
250 green_image = im.*bayer_green;
251
252 %deducing the green pixels at the missing points
253 green = green_image + imfilter(green_image, [0 1 0; 1 0 1; 0 1 0]);
254 for x = 3:2:(imageWidth-2)
255     for y = 3:2:(imageHeight-2)
256         horizontal_gradient = abs((red_image(x,y-2)+red_image(x,y+2))/2 - red_image(x,y));
257         vertical_gradient = abs((red_image(x-2,y)+red_image(x+2,y))/2 - red_image(x,y));
258         if(horizontal_gradient<vertical_gradient)
259             mosim(x,y,2)=(green(x,y-1)+green(x,y+1))/2;
260         elseif (horizontal_gradient>vertical_gradient)
261             mosim(x,y,2)=(green(x-1,y)+green(x+1,y))/2;
262         else
263             mosim(x,y,2)=(green(x-1,y)+green(x+1,y)+green(x,y-1)+green(x,y+1))/4;
264         end
265     end
266 end

```

```

267 - for x = 4:2:(imageWidth-2)
268 -     for y = 4:2:(imageHeight-2)
269 -         horizontal_gradient = abs((blue_image(x,y-2)+blue_image(x,y+2))/2 - blue_image(x,y));
270 -         vertical_gradient = abs((blue_image(x-2,y)+blue_image(x+2,y))/2 - blue_image(x,y));
271 -         if(horizontal_gradient < vertical_gradient)
272 -             mosim(x,y,2) = (green(x,y-1)+green(x,y+1))/2;
273 -         elseif (horizontal_gradient > vertical_gradient)
274 -             mosim(x,y,2) = (green(x-1,y)+green(x+1,y))/2;
275 -         else
276 -             mosim(x,y,2) = (green(x-1,y)+green(x+1,y)+green(x,y-1)+green(x,y+1))/4;
277 -         end
278 -     end
279 - end
280
281 %deducing the blue pixels at missing points
282 - blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
283 - blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
284 - blue = blue_image+blue_1+blue_2;
285
286 %deducing the red pixels at missing points
287 - red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
288 - red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
289 - red = red_image+red_1+red_2;
290
291 - mosim(:, :, 1) = red;
292 - 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.

```
23 %% Denoising algorithm (Gaussian filtering)
24 for sigma = 0.1:0.1:2
25     gaussian_filter=fspecial('gaussian',[3 3],sigma);
26     filtered_image1=imfilter(noise1,gaussian_filter);
27     filtered_image2=imfilter(noise2,gaussian_filter);
28     error_1=sum(sum((im - filtered_image1).^2));
29     error_2=sum(sum((im - filtered_image2).^2));
30     fprintf('Sigma, Error 1, Error 2: %.2f %.2f %.2f\n', sigma, error_1,error_2);
31     figure;
32     imshow(filtered_image1);
33     figure;
34     imshow(filtered_image2);
35 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.

```

36  %% Denoising algorithm (Median filtering)
37  for m=1:5
38      for n=1:5
39          filtered_image3=medfilt2(noise1,[m n]);
40          filtered_image4=medfilt2(noise2,[m n]);
41          error_3=sum(sum((im - filtered_image3).^2));
42          error_4=sum(sum((im - filtered_image4).^2));
43          fprintf('Neighborhood Length, Neighborhood Width, Error 1, Error2: %.2f %.2f %.2f %.2f\n',
44                  figure;
45                  imshow(filtered_image3);
46                  figure;
47                  imshow(filtered_image4);
48              end
49      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

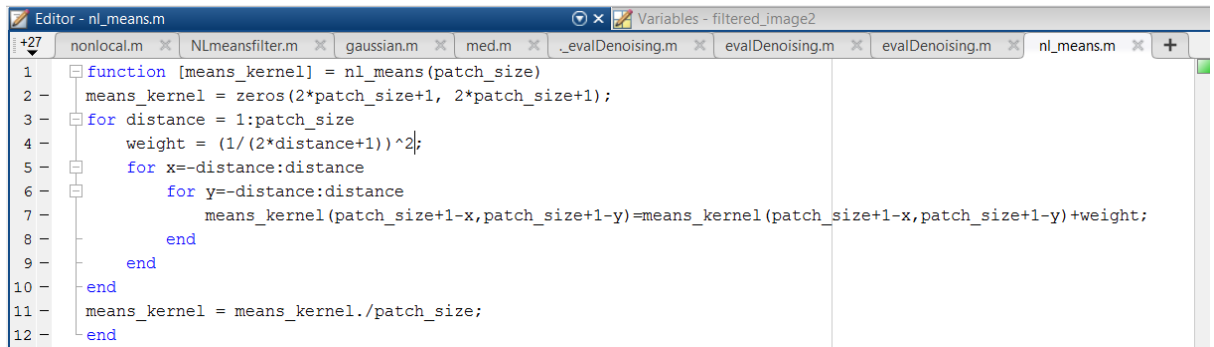
It is seen from the output of saturn2g and saturn2sp shown below that, for a window size of 5*5, the error for the first image is minimum and for a patch size of 3*3, the error for the second image is minimum.

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 $(2 \cdot \text{distance} + 1)^2$. Therefore, the center pixel is weighed the highest, and as we move away from the center, the pixel weights are reduced by this factor.

The image shows a MATLAB Editor window with the file 'nl_means.m' open. The function 'nl_means' is defined, which takes 'patch_size' as input and returns 'means_kernel'. The code initializes 'means_kernel' as a zero matrix of size (2*patch_size+1) x (2*patch_size+1). It then uses nested loops to calculate weights for each pixel in the patch. The weight is calculated as 1/(2*distance+1)^2, where 'distance' is the distance from the center pixel. The weights are then used to calculate the 'means_kernel' by summing the pixel values weighted by these factors. Finally, the 'means_kernel' is normalized by dividing it by the patch_size.

```
1 function [means_kernel] = nl_means(patch_size)
2     means_kernel = zeros(2*patch_size+1, 2*patch_size+1);
3     for distance = 1:patch_size
4         weight = (1/(2*distance+1))^2;
5         for x=-distance:distance
6             for y=-distance:distance
7                 means_kernel(patch_size+1-x,patch_size+1-y)=means_kernel(patch_size+1-x,patch_size+1-y)+weight;
8             end
9         end
10    end
11    means_kernel = means_kernel./patch_size;
12 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.

```
Editor - nl_mean_out.m
Variables - filtered_image2
NLmeansfilter.m x gaussian.m x med.m x _evalDenoising.m x evalDenoising.m x evalDenoising.m x nl_means.m x nl_mean_out.m x
1 function [denoised_image] = nl_mean_out(image_in, window_size, patch_size, gamma)
2
3 [imageWidth, imageHeight]=size(image_in);
4 denoised_image = zeros(imageWidth,imageHeight);
5 image_in = padarray(image_in,[patch_size,patch_size],'circular');
6
7 nl_kernel = nl_means(patch_size);
8 nl_kernel = nl_kernel/sum(sum(nl_kernel));
9
10 squared_gamma = gamma*gamma;
11
12 for x = 1:imageWidth
13     for y = 1:imageHeight
14
15         x_new = x+patch_size;
16         y_new = y+patch_size;
17
18         wind_1 = image_in(x_new-patch_size:x_new+patch_size,y_new-patch_size:y_new+patch_size);
19
20         wind_max = 0;
21         ave = 0;
22         s_cum = 0;
23
24         min_r = max(x_new-window_size,patch_size+1);
25         max_r = min(x_new+window_size, imageWidth+patch_size);
26         min_s = max(y_new-window_size, patch_size+1);
27         max_s = min(y_new+window_size, imageHeight+patch_size);
28
29         for i = min_r:max_r
30             for j = min_s:max_s
31
32                 if(i==x_new && j==y_new)
33                     continue;
34                 end
35
36                 wind_2 = image_in(i-patch_size:i+patch_size, j-patch_size:j+patch_size);
37
38                 diff = sum(sum(nl_kernel.*(wind_1-wind_2).*(wind_1-wind_2)));
39
40                 weigh_param = exp(-diff/squared_gamma);
41
42                 if weigh_param > wind_max
43                     wind_max = weigh_param;
44                 end
45
46                 s_cum = s_cum+weigh_param;
47                 ave = ave +weigh_param*image_in(i,j);
48             end
49         end
50
51         ave = ave + wind_max*image_in(x_new,y_new);
52         s_cum = s_cum + wind_max;
53
54         if s_cum > 0
55             denoised_image(x,y) =ave/s_cum;
56         else
57             denoised_image(x,y) = image_in(x,y);
58         end
59     end
60 end
61 figure;
62 imshow(denoised_image);
63 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.

```

50 %% Denoising alogirthm (Non-local means)
51
52 - filtered_image5 = nl_mean_out(noise1, 8, 2, 0.1);
53 - filtered_image6 = nl_mean_out(noise2, 4, 2, 0.25);
54 - error_5=sum(sum((im - filtered_image5).^2));
55 - error_6=sum(sum((im - filtered_image6).^2));
56 - fprintf('Error 1, Error2: %.2f %.2f\n', error_5,error_6);
57
58

```

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	
	Sigma	Error	Sigma	Error
Gaussian filter	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	Error	Widow Size	Error
	Patch Size		Patch Size	
	Gamma		Gamma	
	8 2 0.1	97.17	4 2 0.25	123.96

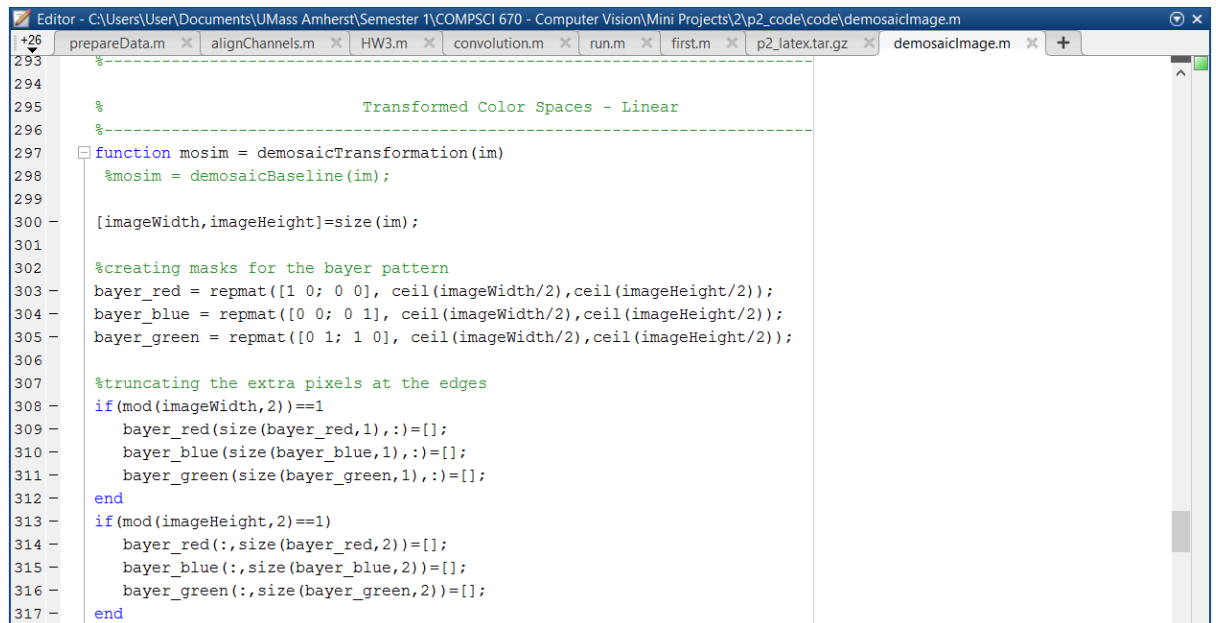
Algorithm and Parameters	Error for Saturn2g		Error for Saturn2sp	
	Sigma	Error	Sigma	Error
Gaussian filter	2	708.71	2	586.07
Median filter	Window Size	Error	Window Size	Error
	5*5	292.39	3*3	18.57
Non local means filter	Window Size	Error	Widow Size	Error
	Patch Size		Patch Size	
	Gamma		Gamma	
	8 2 0.25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

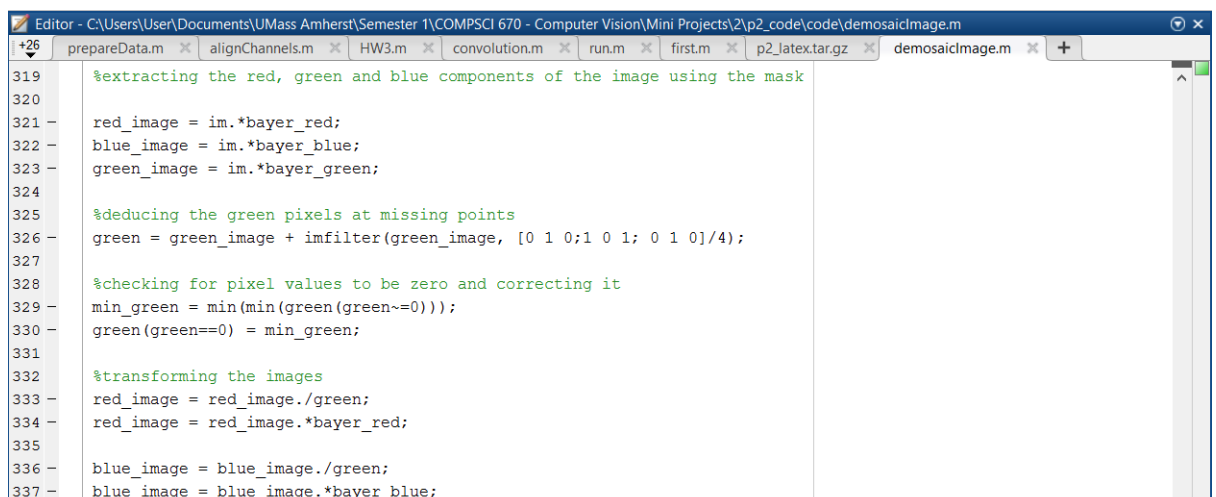
a) Linear Transformation of Color Spaces

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
+26 prepareData.m alignChannels.m HW3.m convolution.m run.m first.m p2_latex.tar.gz demosaicImage.m
293 %-----
294
295 %                               Transformed Color Spaces - Linear
296 %-----
297 function mosim = demosaicTransformation(im)
298     %mosim = demosaicBaseline(im);
299
300     [imageWidth,imageHeight]=size(im);
301
302     %creating masks for the bayer pattern
303     bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2),ceil(imageHeight/2));
304     bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2),ceil(imageHeight/2));
305     bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2),ceil(imageHeight/2));
306
307     %truncating the extra pixels at the edges
308     if(mod(imageWidth,2))==1
309         bayer_red(size(bayer_red,1),:)=[];
310         bayer_blue(size(bayer_blue,1),:)=[];
311         bayer_green(size(bayer_green,1),:)=[];
312     end
313     if(mod(imageHeight,2))==1
314         bayer_red(:,size(bayer_red,2))=[];
315         bayer_blue(:,size(bayer_blue,2))=[];
316         bayer_green(:,size(bayer_green,2))=[];
317     end
318 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
+26 prepareData.m alignChannels.m HW3.m convolution.m run.m first.m p2_latex.tar.gz demosaicImage.m
319 %extracting the red, green and blue components of the image using the mask
320
321 red_image = im.*bayer_red;
322 blue_image = im.*bayer_blue;
323 green_image = im.*bayer_green;
324
325 %deducing the green pixels at missing points
326 green = green_image + imfilter(green_image, [0 1 0;1 0 1; 0 1 0]/4);
327
328 %checking for pixel values to be zero and correcting it
329 min_green = min(min(green(green~=0)));
330 green(green==0) = min_green;
331
332 %transforming the images
333 red_image = red_image./green;
334 red_image = red_image.*bayer_red;
335
336 blue_image = blue_image./green;
337 blue_image = blue_image.*bayer_blue;
```

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
+26 prepareData.m alignChannels.m HW3.m convolution.m run.m first.m p2_latex.tar.gz demosaicImage.m
339 %deducing the red pixels at missing points
340 red_1 = imfilter(red_image, [1 0 1;0 0 1;0 1 0]/4);
341 red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
342 red = red_image+red_1+red_2;
343
344 %deducing the blue pixels at missing points
345 blue_1 = imfilter(blue_image, [1 0 1;0 0 1;0 1 0]/4);
346 blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
347 blue = blue_image+blue_1+blue_2;
348
349 %applying inverse transformation
350 mosim(:, :, 1) = red.*green;
351 mosim(:, :, 2) = green;
352 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.

```

Editor - C:\Users\User\Documents\UMass Amherst\Semester 1\COMPSCI 670 - Computer Vision\Mini Projects\2\p2_code\code\demosaicImage.m
+26 prepareData.m alignChannels.m HW3.m convolution.m run.m first.m p2_latex.tar.gz demosaicImage.m
353
354
355 %----- Transformed Color Spaces - Logarithmic -----
356 %-----
357 function mosim = demosaicLogTransformation(im)
358     %mosim = demosaicBaseline(im);
359
360     [imageWidth, imageHeight] = size(im);
361
362     %creating masks for the bayer pattern
363     bayer_red = repmat([1 0; 0 0], ceil(imageWidth/2), ceil(imageHeight/2));
364     bayer_blue = repmat([0 0; 0 1], ceil(imageWidth/2), ceil(imageHeight/2));
365     bayer_green = repmat([0 1; 1 0], ceil(imageWidth/2), ceil(imageHeight/2));
366
367     %truncating the extra pixels at the edges
368     if(mod(imageWidth, 2) == 1)
369         bayer_red(size(bayer_red, 1), :) = [];
370         bayer_blue(size(bayer_blue, 1), :) = [];
371         bayer_green(size(bayer_green, 1), :) = [];
372     end
373     if(mod(imageHeight, 2) == 1)
374         bayer_red(:, size(bayer_red, 2)) = [];
375         bayer_blue(:, size(bayer_blue, 2)) = [];
376         bayer_green(:, size(bayer_green, 2)) = [];
377     end
378

```

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\demosaiImage.m
+25 prepareData.m x alignChannels.m x HW3.m x convolution.m x run.m x first.m x p2_latex.tar.gz x demosaiImage.m x +
379 %extracting the red, green and blue components of the image using the mask
380
381 green_image = im.*bayer_green;
382 blue_image = im.*bayer_blue;
383 red_image = im.*bayer_red;|
384
385 %deducing the green pixels at missing points
386 green = green_image + imfilter(green_image, [0 1 0;1 0 1; 0 1 0]/4);
387
388 %checking for pixel values to be zero and correcting it
389 min_green = min(min(green(green~=0)));
390 green(green==0) = min_green;
391
392 min_blue = min(min(blue_image(blue_image~=0)));
393 blue_image(blue_image==0) = min_blue;
394
395 min_red = min(min(red_image(red_image~=0)));
396 red_image(red_image==0) = min_red;
397
398 %transforming the images
399 red_image = log(red_image./green);
400 red_image = red_image.*bayer_red;
401
402 blue_image = log(blue_image./green);
403 blue_image = blue_image.*bayer_blue;
404
405 %deducing the red pixels at missing points
406 red_1 = imfilter(red_image, [1 0 1;0 0 0;1 0 1]/4);
407 red_2 = imfilter(red_image, [0 1 0;1 0 1;0 1 0]/2);
408 red = red_image+red_1+red_2;
409
410 %deducing the blue pixels at missing points
411 blue_1 = imfilter(blue_image, [1 0 1;0 0 0;1 0 1]/4);
412 blue_2 = imfilter(blue_image, [0 1 0;1 0 1;0 1 0]/2);
413 blue = blue_image+blue_1+blue_2;
414
415 %applying the inverse transformation
416 mosim(:, :, 1) = exp(red).*green;
417 mosim(:, :, 2) = green;
418 mosim(:, :, 3) = exp(blue).*green;

```

Results

>> evalDemosaiing							
#	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](https://doi.org/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 demosaicing 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 columns);
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', 'CMPSCI 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', 'CMPSCI 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] = runDemosaiicing(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
            end
        end
    end
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

```

