

Hemanth Balaji Dandi**Problem 1.** *Demosaicing*

In this problem, we are going to implement different versions of demosaicing from RAW file (for a Nikon camera, this is a .nef file).

1. Install the Python package **rawpy**, instructions here: <https://pypi.python.org/pypi/rawpy>.
2. Read in the file “tetons.nef” using the rawpy package. Using the package and opencv, figure out how to save the original image as “tetons_original.png”. Next, implement subsampling and extract the red,green, and blue channels, and save a lower resolution image “tetons_subsample.png”.

```
1 ##### Section 1 -Demosaicing #####
2 ##### Section 1.2.1 -Read the file by rawpy and save it #####
3 path = '/home/owner/Documents/HW1_Blackboard/HW1_Blackboard/tetons.nef'
4 with rawpy.imread(path) as raw:
5     rgb = raw.postprocess()
6     imageio.imwrite('tetons_original.png', rgb.astype('uint8'))
```



original.png

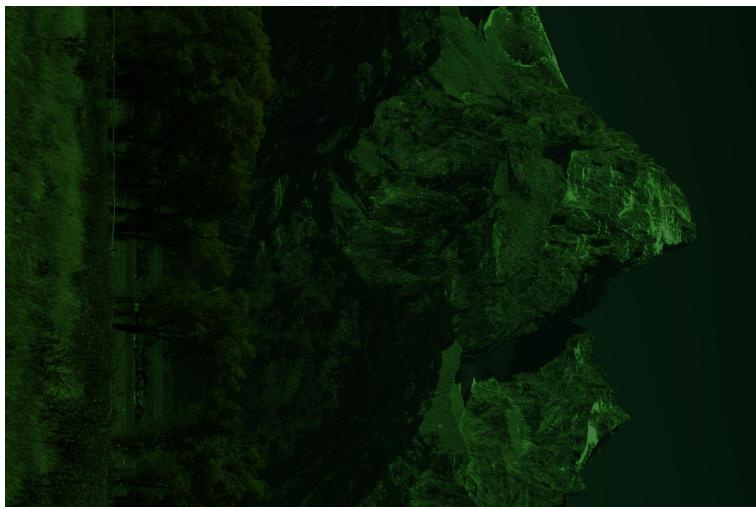
```
1 ##### Section 1 -Demosaicing #####
2 ##### Section 1.2.1 -Read the file by rawpy as a raw-image #####
3 path = '/home/owner/Documents/HW1_Blackboard/HW1_Blackboard/tetons.nef'
4 with rawpy.imread(path) as raw:
5     rgb = raw.postprocess()
6
7 with rawpy.imread(path) as raw:
8     raw_image = raw.raw_image.copy()
9     raw_image1 = raw.raw_image.copy()
10    raw_image2 = raw.raw_image.copy()
11    raw_image3 = raw.raw_image.copy()
```

```
1 ##### Section 1 -Demosaicing #####
2 ##### Section 1.2.1 -Extract R,G,B Channels & Subsampling #####
3 '''Algorithm to extract the Red Pixels as they occur at only even
   positions '''
4 for x in range(0, raw_image1.shape[0]):
5     for y in range(0, raw_image1.shape[1]):
6         if ((x%2==0) and (y%2==0)):
7             raw_image1[x,y]=raw_image1[x,y]
```

```

8     else :
9         raw_image1[x,y]=0
10
11     '''Algorithm to extract the Blue Pixels as they occur at odd positions
12     ,,
13     for x in range(0, raw_image2.shape[0]):
14         for y in range(0, raw_image2.shape[1]):
15             if ((x%2!=0) and (y%2!=0)):
16                 raw_image2[x,y]=raw_image2[x,y]
17             else :
18                 raw_image2[x,y]=0
19
20 red1= raw_image1.copy() # Red Pixels extractd from the BAYER Matrix
21 blue1= raw_image2.copy() # Blue Pixels extractd from the BAYER Matrix
22 green1= raw_image3-red1-blue1 # Green pixels extracted via subtracting
23     the Red and Blue Matices from the Original RAW Bayer Pattern
24
25 red = (red1/16).astype('float32') # Converting from 12-bit to 8-bit by
26     dividing by 2^4
27 blue = (blue1/16).astype('float32')# Converting from 12-bit to 8-bit by
28     dividing by 2^4
29 green = (green1/16).astype('float32')# Converting from 12-bit to 8-bit
30     by dividing by 2^4
31 down_r2=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.
32     INTER_AREA) # Downsampling the red pixels by half
33 down_b2=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.
34     INTER_AREA) # Downsampling the blue pixels by half
35 down_g2=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.
36     INTER_AREA)# Downsampling the green pixels by half
37
38 rgbArray2 = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix
39     where each of the channels will be stacked at different dimensions
40
41 rgbArray2[... , 0] = down_r2 # Red channel in dimension 1
42 rgbArray2[... , 1] = down_g2 # Green channel in dimension 2
43 rgbArray2[... , 2] = down_b2 # Blue channel in dimension 3
44
45 sub_img = Image.fromarray(rgbArray)
46 sub_img.save('tetons_subsample.jpeg') # Subsampled Image

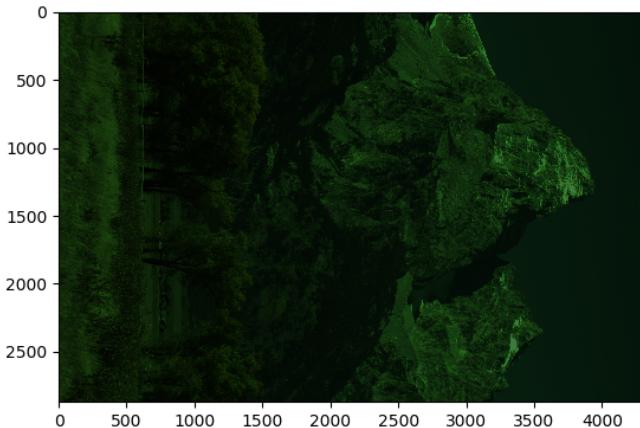
```



Nearest Neighbour Demosaicing

3. Implement nearest neighbor demosaicing (to get back to the original resolution). Save this image as “tetons_nn.png”.

```
1 ##### Section 1 –Demosaicing #####
2 ##### Section 1.3.1 Nearest Neighbours Interpolation #####
3 down_r2=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) # Downsampling the red pixels by half
4 down_b2=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) # Downsampling the blue pixels by half
5 down_g2=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) # Downsampling the green pixels by half
6 rgbArray2 = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix where each of the channels will be stacked at different dimensions
7 rgbArray2[:, :, 0] = down_r2 # Red channel in dimension 1
8 rgbArray2[:, :, 1] = down_g2 # Green channel in dimension 2
9 rgbArray2[:, :, 2] = down_b2 # Blue channel in dimension 3
10 sub_img = Image.fromarray(rgbArray2)
11 sub_img.save('tetons_subsample.jpeg') # Subsampled Image
12 up_NN=cv2.resize(rgbArray2,None,fx=2, fy=2, interpolation= cv2.INTER_AREA) # Upsampling the newly formed matrix by 2 times to get the original image
13 img2 = Image.fromarray(up_NN)
14 img2.save('Nearest_Neighbour.jpeg')
15 plt.imshow(up_NN.astype('uint8'))
16 plt.show()
```



Bilinear Interpolation Demosaicing

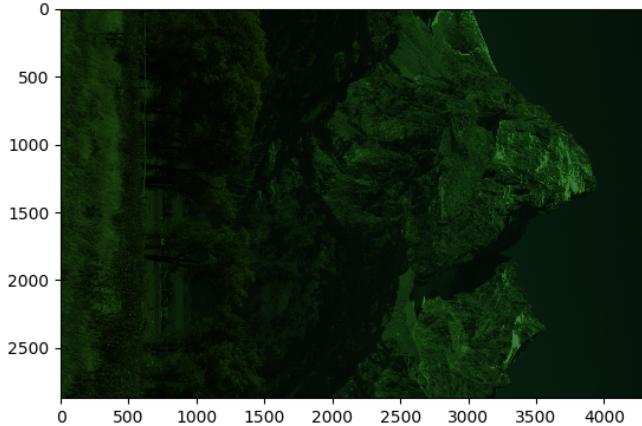
4. Implement bilinear interpolation demosaicing. Save this image as “tetons_bl.png”.

```
1 ##### Section 1 –Demosaicing #####
2 ##### Section 1.3.1 Bilinear Interpolation #####
3 down_r=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR) # Downsampling the red pixels by half
4 down_b=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR) # Downsampling the blue pixels by half
```

```

5 down_g=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR)# Downsampling the green pixels by half
6
7 rgbArray = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix
     where each of the channels will be stacked at different dimensions
8 rgbArray [..., 0] = down_r # Red channel in dimension 1
9 rgbArray [..., 1] = down_g # Green channel in dimension 2
10 rgbArray [..., 2] = down_b # Blue channel in dimension 3
11
12 up_BL=cv2.resize(rgbArray,None,fx=2, fy=2, interpolation= cv2.INTER_LINEAR) # Upsampling the newly formed matrix by 2 times to get
     the original image
13 img = Image.fromarray(up_BL)
14 img.save('Bilinear.jpeg')
15 plt.imshow(up_BL.astype('uint8'))
16 plt.show()

```



Edge Based Interpolation

5. Read the paper by Gunturk et al., *Demosaicing: Color Filter Interpolation*, IEEE Signal Processing Magazine, 2005, http://www.ece.lsu.edu/ipl/papers/IEEE_SPM2005.pdf. Implement the algorithm of edge-based demosaicing for the green channel in Figure 4. For the R/B channels, feel free to just use bilinear interpolation as before. Save this image as “tetons_dm.png”.

```

1 ##### Section 1 –Demosaicing #####
2 ##### Section 1.3.1 Edge Based Demosaicing #####
3 # Predicting Green Pixels
4
5 for x in range(2, red.shape[0]-2,2): #Predicting Green Pixels at Red
    pixel positions
6     for y in range(2, red.shape[1]-2,2):#Starting from (2,2) so as to
        avoid corners
7         dH= (red[x,y-2]+red[x,y+2])/2-red[x,y]
8         dV= (red[x-2,y]+red[x+2,y])/2-red[x,y]
9         if (dH<dV):
10             green[x,y]= (green[x,y-1]+green[x,y+1])/2
11         elif (dH>dV) :

```

```

12     green[x,y] = (green[x-1,y]+green[x+1,y])/2
13 else :
14     green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
15
16
17 for x in range(3, blue.shape[0]-3,2): #Predicting Green Pixels at Blue
18     pixel positions
19     for y in range(3, blue.shape[1]-3,2): #Starting from (3,3) so as to
20         avoid corners
21         dH= (blue[x,y-2]+blue[x,y+2])/2-blue[x,y]
22         dV= (blue[x-2,y]+blue[x+2,y])/2-blue[x,y]
23         if (dH<dV):
24             green[x,y]= (green[x,y-1]+green[x,y+1])/2
25         elif (dH>dV) :
26             green[x,y]= (green[x-1,y]+green[x+1,y])/2
27         else :
28             green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
29 ##### Algorithm to predict Green Pixels at corners and at pixel positions
30     not taken into account above #####
31 for x in range(1, 2):
32     for y in range(1, blue.shape[1]-1,2):
33         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
34
35 for x in range(2, red.shape[0]-2, 2):
36     for y in range(red.shape[1]-2,red.shape[1]-1):
37         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
38
39 for x in range(red.shape[0]-2, red.shape[0]-1):
40     for y in range(2,red.shape[1]-1,2):
41         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
42
43 for x in range(1, blue.shape[0]-1,2):
44     for y in range(blue.shape[1]-1,blue.shape[1]):
45         green[x,y]=(green[x,y-1]+green[x-1,y]+green[x+1,y])/4
46
47 for x in range(blue.shape[0]-1, blue.shape[0]):
48     for y in range(1,blue.shape[1]-1,2):
49         green[x,y]=(green[x,y-1]+green[x-1,y])/4
50
51 for x in range(1,blue.shape[0]-1,2):
52     for y in range(1,2):
53         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
54
55 for x in range(0, 1):
56     for y in range(2, red.shape[1]-1,2):
57         green[x,y]=(green[x,y-1]+green[x,y+1])/2
58
59 for x in range(0, blue.shape[0],2):
60     for y in range(0,1):
61         green[x,y]=(green[x,y+1])/2
62
63 # Predicting Red Pixels by Bilinear Interpolation
64
65 for x in range(0, red.shape[0],2):
66     for y in range(1, red.shape[1]-1,2):

```

```

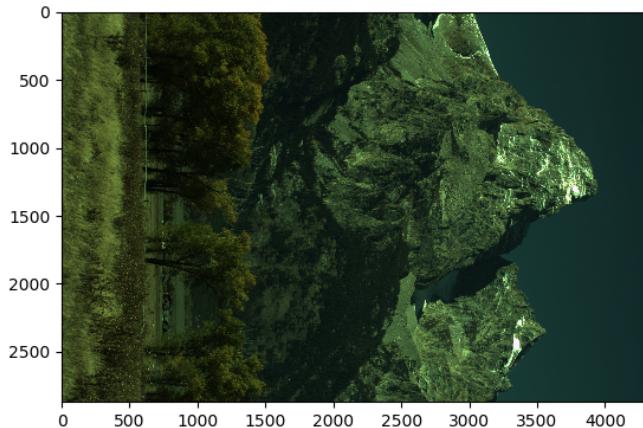
64     red[x,y] = (red[x,y-1]+red[x,y+1])/2
65
66
67 for x in range(1, red.shape[0]-1,2):
68     for y in range(0, red.shape[1],2):
69         red[x,y]=(red[x-1,y]+red[x+1,y])/2
70
71
72 for x in range(1, red.shape[0]-1,2):
73     for y in range(1, red.shape[1]-1,2):
74         red[x,y]=(red[x-1,y]+red[x-1,y+1]+red[x+1,y-1]+red[x+1,y+1])/4
75
76 for x in range(red.shape[0]-1, red.shape[0]):
77     for y in range(0, red.shape[1],1):
78         red[x,y]=(red[x-1,y])/2
79
80 for x in range(0, red.shape[0]-1):
81     for y in range(4310, red.shape[1],1):
82         red[x,y]=525
83
84
85 # Predicting Blue Pixels by Bilinear Interpolation
86
87 for x in range(1, blue.shape[0]-1,2):
88     for y in range(2, blue.shape[1]-2,2):
89         blue[x,y]=(blue[x,y-1]+blue[x,y+1])/2
90
91
92 for x in range(2, blue.shape[0]-2,2):
93     for y in range(1, blue.shape[1]-1,2):
94         blue[x,y]=(blue[x-1,y]+blue[x+1,y])/2
95
96 for x in range(2, blue.shape[0]-2,2):
97     for y in range(2, blue.shape[1]-2,2):
98         blue[x,y]=(blue[x,y-1]+blue[x,y+1]+blue[x-1,y]+blue[x+1,y])/4
99
100 for x in range(1, blue.shape[0],1):
101     for y in range(0, blue.shape[1]-2,blue.shape[1]):
102         blue[x,y]=(blue[x,y+1])/2
103
104 for x in range(1, blue.shape[0],1):
105     for y in range(blue.shape[1]-1, blue.shape[1]):
106         blue[x,y]=(blue[x,y-1])/2
107
108 for x in range(0, blue.shape[0],blue.shape[0]):
109     for y in range(0, blue.shape[1]):
110         blue[x,y]=(blue[x+1,y])/2
111
112 for x in range(blue.shape[0]-2, blue.shape[0],blue.shape[0]):
113     for y in range(0, blue.shape[1]):
114         blue[x,y]=(blue[x+1,y]+blue[x-1,y])/2
115
116 for x in range(blue.shape[0]-1, blue.shape[0],blue.shape[0]):
117     for y in range(2, blue.shape[1]-2):
118         blue[x,y]=(blue[x-1,y])/2
119

```

```

120
121 edge_interpolation = np.zeros((2868,4320,3), 'uint8')#Stacking the Red,
122     Green and Blue Matrices into a Matrix
123 edge_interpolation[... , 0] = red
124 edge_interpolation[... , 1] = green
125 edge_interpolation[... , 2] = blue
126 img3 = Image.fromarray(edge_interpolation)
127 img3.save('Edge_Interpolation.jpeg')
128 plt.imshow(edge_interpolation.astype('uint8'))
129 plt.show()

```



6. Create a side by side comparison of the three methods (NN, bilinear, and Gunturk et al.). Also include a zoom in of 2-3 regions (30x30 pixel windows or so) to illustrate the advantages of the methods closeup.

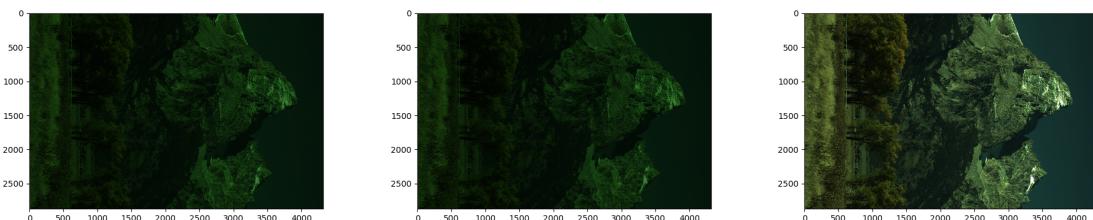


Fig1. Nearest Neighbours Demosaicing,Bilinear Demosaicing and Edge-Based Demosaicing

Variation between the 3 methods across 2 regions

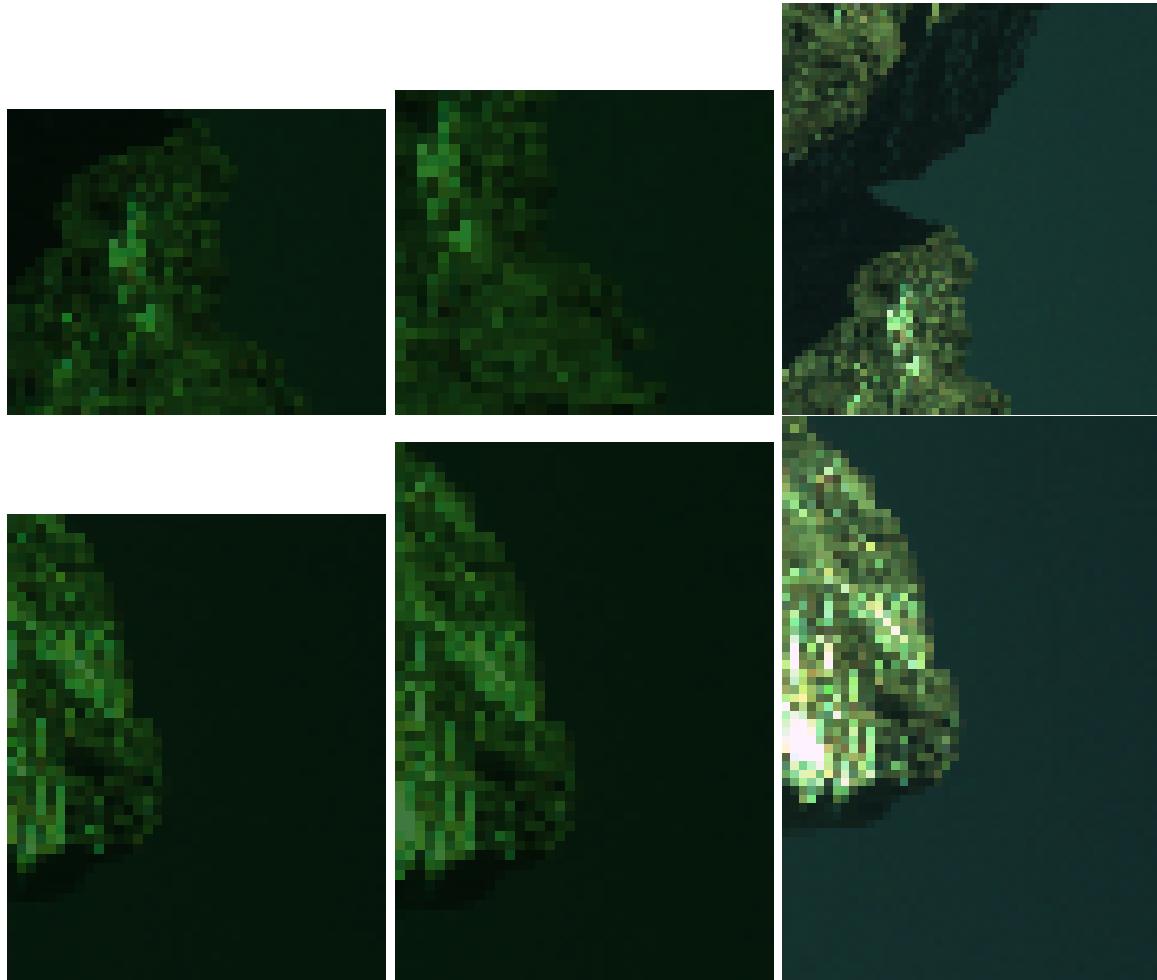


Fig2. Nearest Neighbours Demosaicing,Bilinear Demosaicing and Edge-Based Demosaicing

As we can see, the demosaicing through Nearest neighbours is slightly more pixelated and unclear than the bilinear interpolation demosaicing, whereas the edge-based demosaicing provides comparatively better edges and clearer image.

Problem 2. HDR Imaging

The point of this problem is to implement the basic algorithm for high dynamic range imaging. In the folder is an exposure stack of .nef files. Each filename is an exposure taken at the time $t_k = \frac{1}{2048} 2^{k-1}$ where k varies from 1 to 16.

1. First we need to process the RAW files to do color balancing, and all ISP operations **except** gamma compression and auto-brightness. The relevant RawPy command is

```
1  rgb = raw.postprocess(gamma=(1,1), no_auto_bright=True, output_bps=16)
```

NOTE: For this problem, work on $1/2$ or $1/4$ the resolution, whatever allows your pictures to still process in a reasonable amount of time. Output/save the processed images as “processed_exposurek.tiff” where $k = 1, \dots, 16$. For the rest of the problem, you will just work on these processed images.

```
1 ##### Section 2 -HDR Imaging #####
```

```

2 ##### Section 1.2.1 –Read the RAW files by rawpy and save the exposures
3 #####
4 exposures=['exposure1.nef','exposure2.nef','exposure3.nef','exposure4.
5 nef','exposure5.nef','exposure6.nef','exposure7.nef','exposure8.nef',
6 , 'exposure9.nef','exposure10.nef','exposure11.nef','exposure12.nef',
7 , 'exposure13.nef','exposure14.nef','exposure15.nef','exposure16.nef']
8 #Above is a list containing the exposures
9
10 for count,exp in enumerate(exposures,1): #Iterating through all of the
11 exposures
12 with rawpy.imread(exp) as raw:
13     rgb = raw.postprocess(gamma=(1,1), no_auto_bright=True,
14     output_bps=16) # Color-balancing and converting the exposures into
15     display-form
16     rgb_down=cv2.resize(rgb,None,fx=0.1, fy=0.1, interpolation=cv2.
17     INTER_CUBIC) # Downsampling them for faster computation
18     imageio.imwrite('processed_exposure'+str(count)+'.tiff',rgb_down) #
19     Saving the processed exposures respectively

```

2. Calculate the HDR image using the following formula:

$$I_{HDR}[i, j, c] = \frac{\sum_k w(I_{LDR}^k[i, j, c]) I_{LDR}^k[i, j, c] / t^k}{\sum_k w(I_{LDR}^k[i, j, c])} \quad (1)$$

where the weighting between exposure frames is:

$$w(z) = e^{-4\frac{(z-0.5)^2}{0.5^2}}, \text{ for } Z_{\min} \leq z \leq Z_{\max}.$$

NOTE: This formula assumes all pixels are between 0 and 1.

HDR Tone-Mapping without in-built

```

1 ##### Section 2 –HDR Imaging #####
2 ##### Section 1.2.1 –Calculating the HDR Image #####
3
4 def weight(z = [], *args): # Weight function to select particular pixels
5     while throwing out unrequired ones
6     z=(z-np.min(z))/(np.max(z)-np.min(z))
7     z=np.exp(-4*(np.power((z-0.5),2)/0.25))
8     return z
9
10 def exposure_time(k): # This function takes into account the time
11     between exposures taken by the camera
12     t=(1/2048)*np.power(2,k-1)
13     return t
14
15 #The Saved Processed Exposures in a list
16 filenames=['processed_exposure1.tiff','processed_exposure2.tiff','
17 processed_exposure3.tiff','processed_exposure4.tiff','
18 processed_exposure5.tiff','processed_exposure6.tiff','
19 processed_exposure7.tiff','processed_exposure8.tiff','
20 processed_exposure9.tiff','processed_exposure10.tiff','
21 processed_exposure11.tiff','processed_exposure12.tiff','
22 processed_exposure13.tiff','processed_exposure14.tiff','
23 processed_exposure15.tiff','processed_exposure16.tiff']

```

```

15
16 bgr1 = cv2.imread(filenames[0]) # Reading-in the first image so as to
   initialize the weight and HDR functions in its shape
17 w= np.zeros(bgr1.shape) # Initilazing the weight function as zero
18 hdr=np.zeros(bgr1.shape) # Initilazing the hdr function as zero
19
20 for count ,name in enumerate(filenames ,1): #Iterating through the
   processed filenames
21 bgr = cv2.imread(name)
22 bgr_norm = cv2.normalize(bgr , dst=None , alpha=0 , beta=1 , norm_type=cv2.
   NORM_MINMAX , dtype=cv2.CV_32F) # Normalizing the functions between 0
   and 1
23 w+=weight(bgr_norm) # Calculating the weight across every exposure and
   adding it every time to the final weight function
24 hdr+=weight(bgr_norm)*bgr_norm/exposure_time(count) #HDR Calculation
   formula
25
26 hdr/=w # Normalizing the HDR by dividing by the total weight function
27 cv2.imwrite("hdr1.hdr" , hdr) # Saving te HDR image

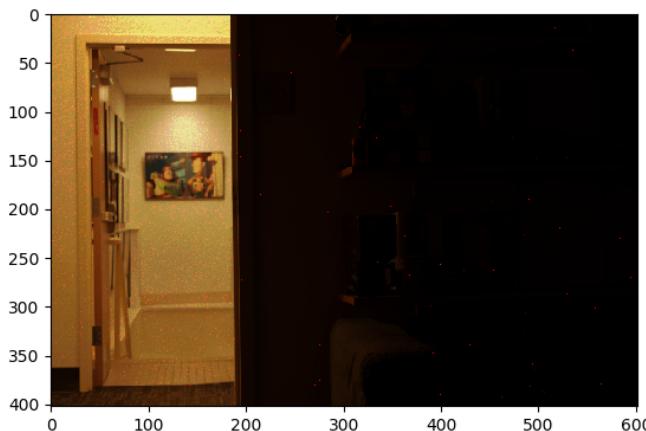
```

3. To display the image, use the tonemapping $I = I_{\text{HDR}}/(1 + I_{\text{HDR}})$. Display and save this image as “HDR_photonemap.png”.

```

1 ##### Section 2 -HDR Imaging #####
2 ### Section 1.2.1 -Displaying with Tonemapping (User-Defined) #####
3
4 ''' Algorithm for Tone Mapping- User Defined '''
5 I= (hdr/(1+hdr)).astype('float32') # Tone-Mapping to obtain the bright-
   right hand side of the image and casting into float32
6
7 cv2.imwrite("HDR_tone.jpg" , I*255) # Saving the Tone-mapped Image

```



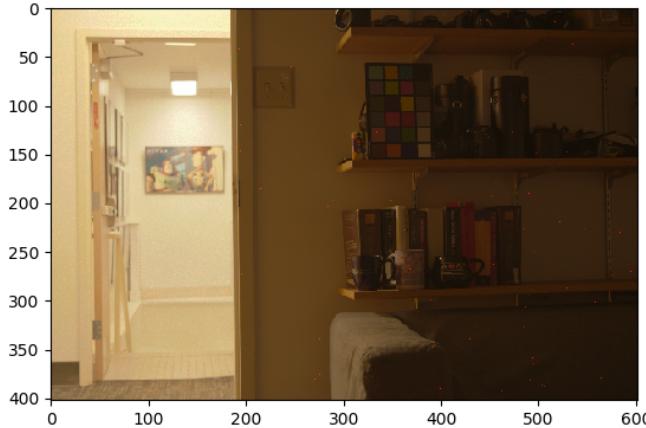
HDR Tone-Mapping using Reinhard Tonemapping (Built-in)

4. Use a built-in tone-mapping from OpenCV. Be careful to cast your images into float32 format for some of the built-in mappings. Play around with the parameters until you find a tonemapping you like. Display and save this image.

```

1 ##### Section 2 -HDR Imaging #####
2 ### Section 1.2.1 -Displaying with Tonemapping (Built-in) #####
3 ##### Reinhard Tonemapping #####
4
5 tonemap_Reinhard = cv2.createTonemapReinhard(3.0, 0, 0, 0)
6 Reinhard = tonemap_Reinhard.process(hdr.astype('float32'))
7 cv2.imwrite("Reinhard.jpg", Reinhard*255)

```



Problem 3. Bilateral Filter

The point of this problem is to implement and play with the bilateral filter.

1. Read in the elephant image in the directory. Using Gaussian blur in OpenCV, blur the image.

```

1 ##### Section 2 -Bilateral Filter #####
2 ### Section 1.2.1 -Read in the Image & #####
3 ##### Gaussian- Blur it #####
4
5 filename='babylephant.jpg'
6 bgr = cv2.imread(filename) # Reading the image
7 bgr1=cv2.resize(bgr,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_CUBIC)
     # Downsampling it for faster computation
8 blur = cv2.GaussianBlur(bgr1,(5,5),10) # Applying Gaussian Blur across
     it with (one case) kernel size 5x5 and sigma 10
9 cv2.imwrite("original_image.png", bgr1) # Saving the original image
10 cv2.imwrite("Guassian_blurred_image.png", blur) # Saving the blurred
      image

```

2. Implement the bilateral filter as described in the lecture slides. Display and save three images side by side: original image, Gaussian blurred image, and bilateral filtered image. You might have to do some parameter searching to find a good bilateral filter. Make sure you show some blow-ups/close details (small pixel windows) to show the effect of the bilateral filter in preserving edges while smoothing noise.

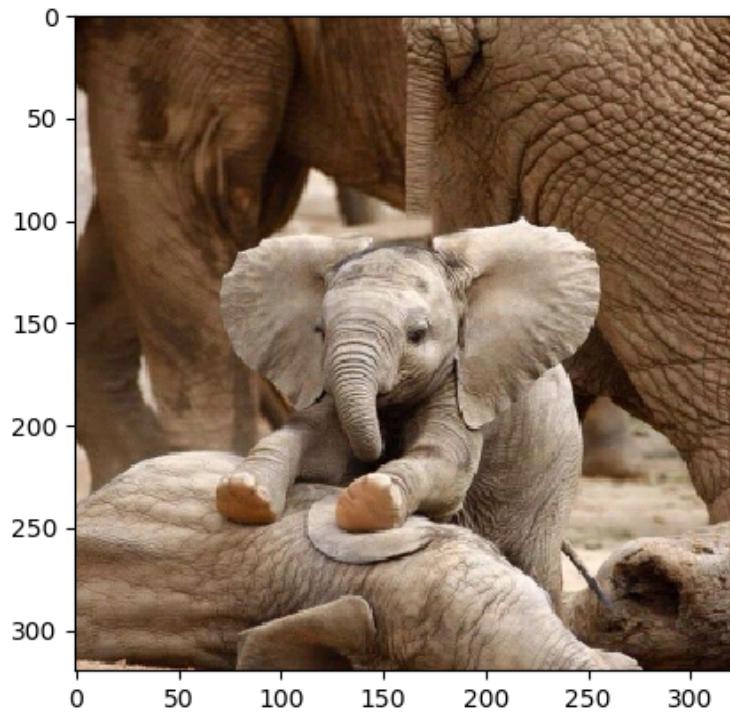


Figure 1: Original Image

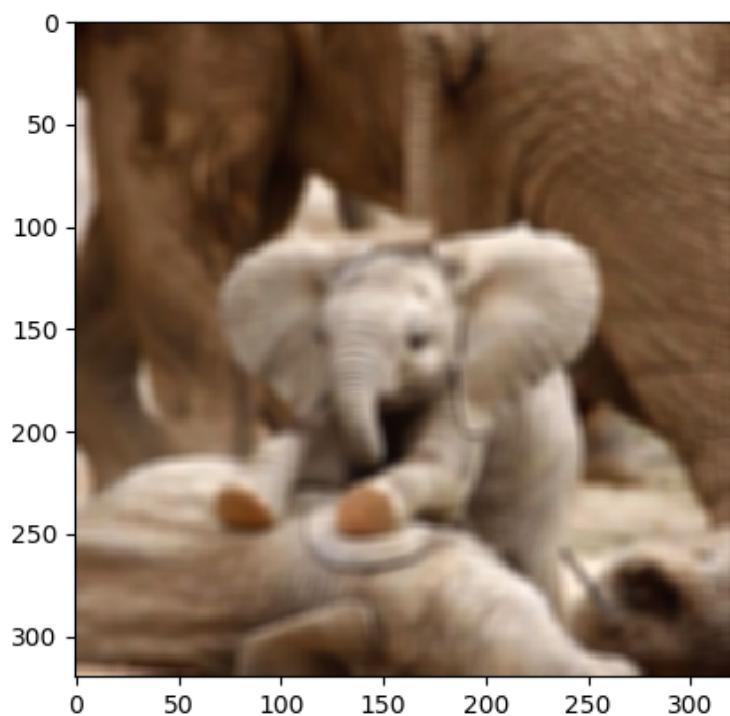


Figure 2: Original Image

```

1 ##### Section 2 -Bilateral Filter #####
2 ### Section 1.2.1 -Implementing Bilateral Filter #####
3 ##### User- Defined #####
4
5     ''' Function to calculate the Bilateral filter '''
6 def designed_bilateral_filter(image,d,sigma_c,sigma_d): # Takes
    parameters-Image, Window Diameter, Variance across color , Variance
    across distance
7     b,g,r = cv2.split(image) # Splitting the image into Red, green & blue
    channels
8     bilatb=np.zeros(b.shape)
9     bilatg=np.zeros(g.shape)
10    bilatr=np.zeros(r.shape)
11
12    m= int((d-1)/2) # Radius calulation , ie , distance form the center
    pixel to the corresponding window
13    for i in range(m,b.shape[0]-m): # Starting window at pixel position (m
    ,m) so as to avoid corners
14        for j in range(m,b.shape[1]-m):
15            Wp1=0 # Total weight initlaized to zero every-time the window
    moves to next pixel
16            Wp2=0
17            Wp3=0
18            for k in range(i-m,m+i+1): # Window function of diameter 'd'
    for l in range(j-m,m+j+1):
19                #print(weight(i,j,k,l))
20                Wp1+=weight(b,i,j,k,l,sigma_c,sigma_d) # Calculation of weight
    function across each channel and adding all of them up (In a
    particular Window)
21                Wp2+=weight(g,i,j,k,l,sigma_c,sigma_d)
22                Wp3+=weight(r,i,j,k,l,sigma_c,sigma_d)
23                bilatb[i,j]+=b[k,l]*weight(b,i,j,k,l,sigma_c,sigma_d) #
    Calculation of denoised pixel at position (i,j) accross every
    channel
24                bilatg[i,j]+=g[k,l]*weight(g,i,j,k,l,sigma_c,sigma_d)
25                bilatr[i,j]+=r[k,l]*weight(r,i,j,k,l,sigma_c,sigma_d)
26                if (k==m+i) and (l==m+j):
27                    bilatb[i,j]=int(round(bilatb[i,j]/Wp1)) # At the end of the
    window, we must divide by the total wweight function across every
    pixel in that window.
28                    bilatg[i,j]=int(round(bilatg[i,j]/Wp2))
29                    bilatr[i,j]=int(round(bilatr[i,j]/Wp3))
30
31    ''' This is to take care of the corner pixels , just replacing the zeros
    with the corresponding pixel values across the channels '''
32    for i in range(0,m):
33        for j in range(0,b.shape[1]):
34            bilatb[i,j]=b[i,j]
35            bilatg[i,j]=g[i,j]
36            bilatr[i,j]=r[i,j]
37    for i in range(b.shape[0]-m-1,b.shape[0]):
38        for j in range(0,b.shape[1]):
39            bilatb[i,j]=b[i,j]
40            bilatg[i,j]=g[i,j]
41            bilatr[i,j]=r[i,j]
42    for i in range(0,b.shape[0]):
43        for j in range(0,m):

```

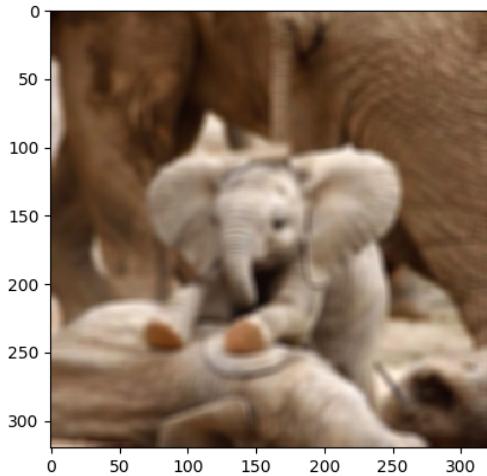


Figure 3: Guassian Blurred Image (Kernel- 9x9)

```

44     bilatb[i,j]=b[i,j]
45     bilatg[i,j]=g[i,j]
46     bilatr[i,j]=r[i,j]
47     for i in range(0,b.shape[0]):
48         for j in range(b.shape[1]-m-1,b.shape[1]):
49             bilatb[i,j]=b[i,j]
50             bilatg[i,j]=g[i,j]
51             bilatr[i,j]=r[i,j]
52
53     bilat1 = np.zeros(image.shape) # Stacking all the channels in matrix
54     bilat1[... , 0] = bilatb # Red Matrix
55     bilat1[... , 1] = bilatg # Green Matrix
56     bilat1[... , 2] = bilatr # Red Matrix
57     return bilat1
58
59 bilat = designed_bilateral_filter(bgr1,5,30,10) # Bilateral Filtered
60     Image with filter size 5x5, variace across color=30, variance across
61     distance=10
62 cv2.imwrite("Denoised_image_BI.png", bilat.astype('uint8')) # Saving the
63     image

```

```

1 ##### Section 2 –Bilateral Filter #####
2 ### Section 1.2.1 –Implement Bilateral Filter #####
3 ##### In–Built Function #####
4
5 filtered_image_OpenCV = cv2.bilateralFilter(bgr1, 5, 30, 10)
6
7 cv2.imwrite("Denoised_image_OpenCV.png",filtered_image_OpenCV)

```

As we can clearly see, the Gaussian blur blurs out even the edges of the image, while the bilateral filter blurs out only the lines on the head, keeping the main edges intact. Thus bilateral works as a better denoising filter

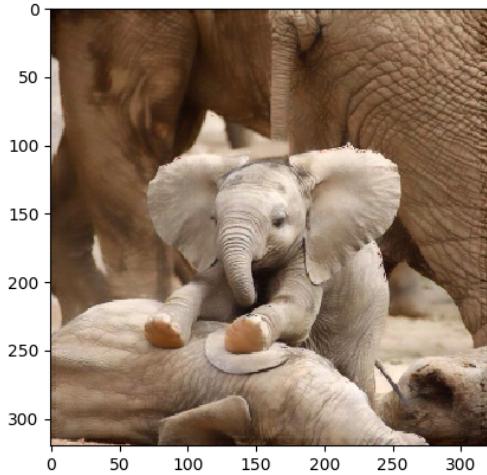


Figure 4: Implemented Bilateral Filtered Image (Filter size=9, Sigma color=30, Sigma distance=50)

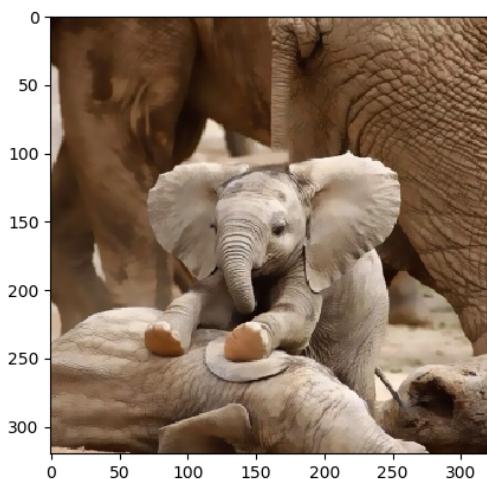


Figure 5: OpenCV Bilateral Filtered Image (Filter size=9, Sigma color=30, Sigma distance=50)

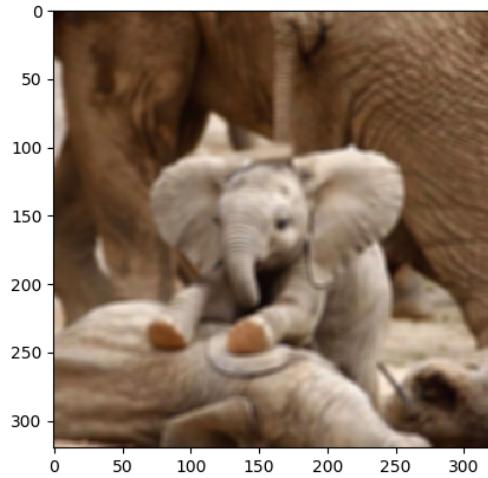


Figure 6: Guassian Blurred Image (Kernel- 7x7)

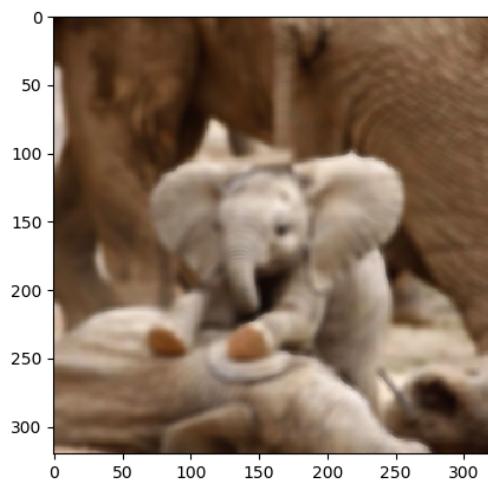


Figure 7: Implemented Bilateral Filtered Image (Filter size=7, Sigma color=500, Sigma distance=50)

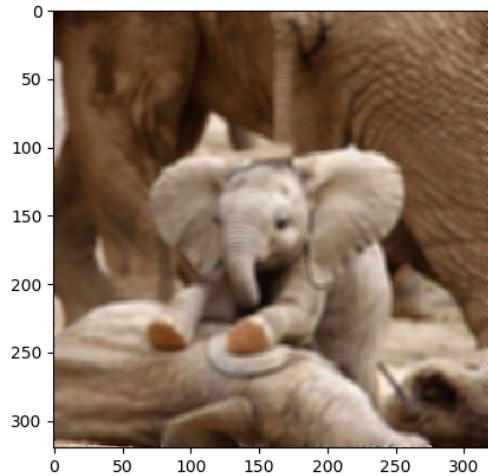


Figure 8: OpenCV Bilateral Filtered Image (Filter size=9, Sigma color=500, Sigma distance=50)

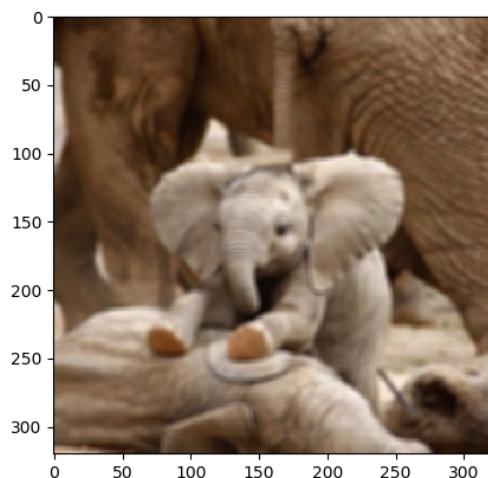


Figure 9: Guassian Blurred Image (Kernel- 5x5)

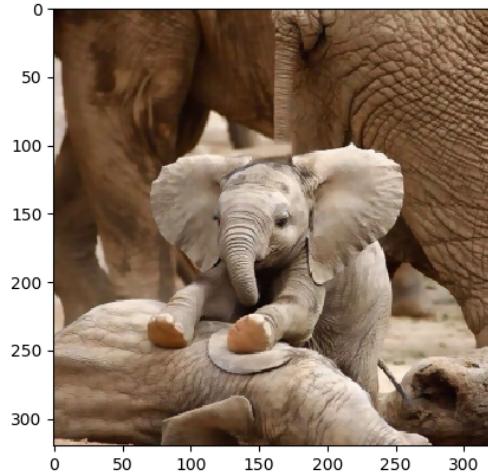


Figure 10: Implemented Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=50)

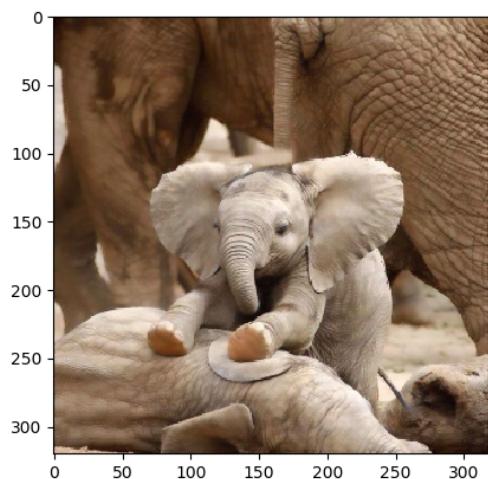


Figure 11: OpenCV Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=50)

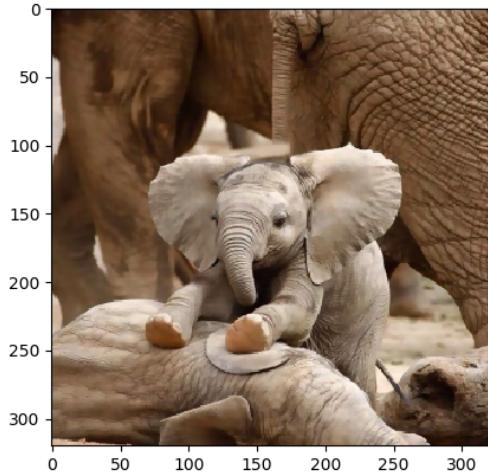


Figure 12: Implemented Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=10)

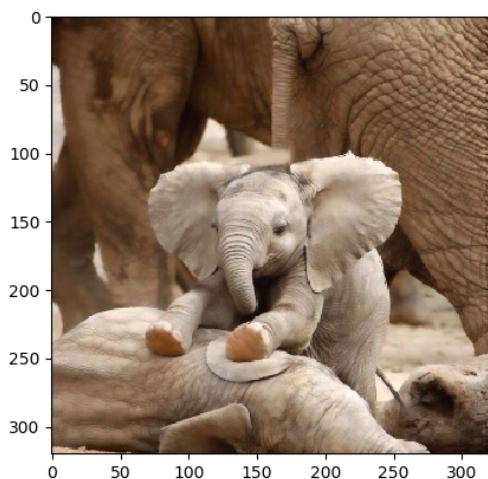


Figure 13: OpenCV Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=10)



Figure 14: Original Cropped image of the top of elephant's head, between the ears

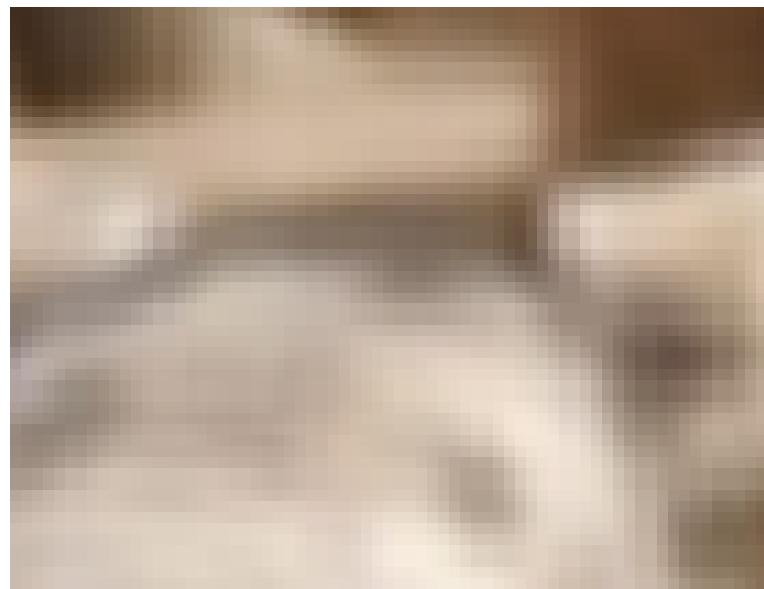


Figure 15: Guassian Blurred Image (Kernel- 5x5)



Figure 16: Implemented Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=10)



Figure 17: OpenCV Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=10)

Problem 4. Appendix

```
1 ''' Program to Perfrom Demosaicing via 3 methods– Nearest Neighbours ,  
2 Bilinear Interpolation and Edge–Based Interpolation Demosaicing '''  
3  
4 import rawpy  
5 import imageio  
6 import cv2  
7 import numpy as np  
8 import matplotlib.pyplot as plt  
9 import ipdb  
10 from PIL import Image  
11 import colorsys  
12  
13 path = '/home/owner/Documents/HW1_Blackboard/HW1_Blackboard/tetons.nef'  
14 with rawpy.imread(path) as raw:  
15     rgb = raw.postprocess()      # Demosaicing done by the RawPy package to  
16     convert the RGGB Bayer Matrix into a 3–D Image  
17  
18 imageio.imwrite('tetons original.png', (rgb).astype('uint8')) #That Image is  
19 saved  
20  
21 with rawpy.imread(path) as raw: ## To Perform Demosaicing across 3 methods ,  
22     so we need to extract the RAW image (Bayer Matrix)  
23     raw_image = raw.raw_image.copy() # Creating copies of the RAW matrix in 3  
24     different variables so as to extract Red, Green and Blue channels from  
25     them  
26     raw_image1 = raw.raw_image.copy()  
27     raw_image2 = raw.raw_image.copy()  
28     raw_image3 = raw.raw_image.copy()  
29  
30  
31     '''Algorithm to extract the Red Pixels as they occur at only even positions  
32     ,,,,  
33  
34     for x in range(0, raw_image1.shape[0]):  
35         for y in range(0, raw_image1.shape[1]):  
36             if ((x%2==0) and (y%2==0)):  
37                 raw_image1[x,y]=raw_image1[x,y]  
38             else :  
39                 raw_image1[x,y]=0  
40  
41     '''Algorithm to extract the Blue Pixels as they occur at odd positions '''  
42     for x in range(0, raw_image2.shape[0]):  
43         for y in range(0, raw_image2.shape[1]):  
44             if ((x%2!=0) and (y%2!=0)):  
45                 raw_image2[x,y]=raw_image2[x,y]  
46             else :  
47                 raw_image2[x,y]=0  
48  
49 red1= raw_image1.copy() # Red Pixels extractd from the BAYER Matrix
```

```

45 blue1= raw_image2.copy() # Blue Pixels extractd from the BAYER Matrix
46 green1= raw_image3-red1-blue1 # Green pixels extracted via subtracting the
    Red and Blue Matrices from the Original RAW Bayer Pattern
47
48 red = (red1/16).astype('float32') # Converting from 12-bit to 8-bit by
    dividing by 2^4
49 blue = (blue1/16).astype('float32')# Converting from 12-bit to 8-bit by
    dividing by 2^4
50 green = (green1/16).astype('float32')# Converting from 12-bit to 8-bit by
    dividing by 2^4
51
52 '''Algorithm to perform downsampling then upsampling by Bilinear
    Interpolation Demosaicing '',
53 down_r=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR) #
    Downsampling the red pixels by half
54 down_b=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR)#
    Downsampling the blue pixels by half
55 down_g=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR)
    # Downsampling the green pixels by half
56
57 rgbArray = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix where
    each of the channels will be stacked at different dimensions
58 rgbArray [..., 0] = down_r # Red channel in dimension 1
59 rgbArray [..., 1] = down_g # Green channel in dimension 2
60 rgbArray [..., 2] = down_b # Blue channel in dimension 3
61
62 sub_img = Image.fromarray(rgbArray)
63 sub_img.save('tetons_subsample.jpeg') # Subsampled Image
64
65 up_BL=cv2.resize(rgbArray,None,fx=2, fy=2, interpolation= cv2.INTER_LINEAR) #
    Upsampling the newly formed matrix by 2 times to get the original image
66 img = Image.fromarray(up_BL)
67 img.save('Bilinear.jpeg')
68 plt.imshow(up_BL.astype('uint8'))
69 plt.show()
70
71 '''Algorithm to perform downsampling then upsampling by Nearest Neighbours
    Demosaicing '',
72 down_r2=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) #
    Downsampling the red pixels by half
73 down_b2=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) #
    Downsampling the blue pixels by half
74 down_g2=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA)#
    Downsampling the green pixels by half
75 rgbArray2 = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix where
    each of the channels will be stacked at different dimensions
76 rgbArray2 [..., 0] = down_r2 # Red channel in dimension 1
77 rgbArray2 [..., 1] = down_g2 # Green channel in dimension 2
78 rgbArray2 [..., 2] = down_b2 # Blue channel in dimension 3
79 up_NN=cv2.resize(rgbArray2,None,fx=2, fy=2, interpolation= cv2.INTER_AREA)#
    Upsampling the newly formed matrix by 2 times to get the original image
80 img2 = Image.fromarray(up_NN)
81 img2.save('Nearest_Neighbour.jpeg')
82 plt.imshow(up_NN.astype('uint8'))
83 plt.show()
84
```

```

85     '''Algorithm to perform downsampling then upsampling by Edge-Based
86     Demosaicing'','
87 # Predicting Green Pixels
88
89 for x in range(2, red.shape[0]-2,2): #Predicting Green Pixels at Red pixel
    positions
90 for y in range(2, red.shape[1]-2,2):#Starting from (2,2) so as to avoid
    corners
91     dH= (red[x,y-2]+red[x,y+2])/2-red[x,y]
92     dV= (red[x-2,y]+red[x+2,y])/2-red[x,y]
93     if (dH<dV):
94         green[x,y]= (green[x,y-1]+green[x,y+1])/2
95     elif (dH>dV) :
96         green[x,y]= (green[x-1,y]+green[x+1,y])/2
97     else :
98         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
99
100
101 for x in range(3, blue.shape[0]-3,2): #Predicting Green Pixels at Blue pixel
    positions
102 for y in range(3, blue.shape[1]-3,2): #Starting from (3,3) so as to avoid
    corners
103     dH= (blue[x,y-2]+blue[x,y+2])/2-blue[x,y]
104     dV= (blue[x-2,y]+blue[x+2,y])/2-blue[x,y]
105     if (dH<dV):
106         green[x,y]= (green[x,y-1]+green[x,y+1])/2
107     elif (dH>dV) :
108         green[x,y]= (green[x-1,y]+green[x+1,y])/2
109     else :
110         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
111 ##### Algorithm to predict Green Pixels at corners and at pixel positions not
    taken into account above #####
112 for x in range(1, 2):
113     for y in range(1, blue.shape[1]-1,2):
114         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
115
116 for x in range(2, red.shape[0]-2, 2):
117     for y in range(red.shape[1]-2,red.shape[1]-1):
118         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
119
120 for x in range(red.shape[0]-2, red.shape[0]-1):
121     for y in range(2,red.shape[1]-1,2):
122         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
123
124 for x in range(1, blue.shape[0]-1,2):
125     for y in range(blue.shape[1]-1,blue.shape[1]):
126         green[x,y]=(green[x,y-1]+green[x-1,y]+green[x+1,y])/4
127
128 for x in range(blue.shape[0]-1, blue.shape[0]):
129     for y in range(1,blue.shape[1]-1,2):
130         green[x,y]=(green[x,y-1]+green[x-1,y])/4
131
132 for x in range(1,blue.shape[0]-1,2):
133     for y in range(1,2):
134         green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

```

```

135 for x in range(0, 1):
136     for y in range(2, red.shape[1]-1,2):
137         green[x,y]=(green[x,y-1]+green[x,y+1])/2
138
139 for x in range(0, blue.shape[0],2):
140     for y in range(0,1):
141         green[x,y]=(green[x,y+1])/2
142
143 # Predicting Red Pixels by Bilinear Interpolation
144
145 for x in range(0, red.shape[0],2):
146     for y in range(1, red.shape[1]-1,2):
147         red[x,y]=(red[x,y-1]+red[x,y+1])/2
148
149
150 for x in range(1, red.shape[0]-1,2):
151     for y in range(0, red.shape[1],2):
152         red[x,y]=(red[x-1,y]+red[x+1,y])/2
153
154
155 for x in range(1, red.shape[0]-1,2):
156     for y in range(1, red.shape[1]-1,2):
157         red[x,y]=(red[x-1,y]+red[x-1,y+1]+red[x+1,y-1]+red[x+1,y+1])/4
158
159 for x in range(red.shape[0]-1, red.shape[0]):
160     for y in range(0, red.shape[1],1):
161         red[x,y]=(red[x-1,y])/2
162
163 for x in range(0, red.shape[0]-1):
164     for y in range(4310, red.shape[1],1):
165         red[x,y]=525
166
167
168 # Predicting Blue Pixels by Bilinear Interpolation
169
170 for x in range(1, blue.shape[0]-1,2):
171     for y in range(2, blue.shape[1]-2,2):
172         blue[x,y]=(blue[x,y-1]+blue[x,y+1])/2
173
174
175 for x in range(2, blue.shape[0]-2,2):
176     for y in range(1, blue.shape[1]-1,2):
177         blue[x,y]=(blue[x-1,y]+blue[x+1,y])/2
178
179 for x in range(2, blue.shape[0]-2,2):
180     for y in range(2, blue.shape[1]-2,2):
181         blue[x,y]=(blue[x,y-1]+blue[x,y+1]+blue[x-1,y]+blue[x+1,y])/4
182
183 for x in range(1, blue.shape[0],1):
184     for y in range(0, blue.shape[1]-2,blue.shape[1]):
185         blue[x,y]=(blue[x,y+1])/2
186
187 for x in range(1, blue.shape[0],1):
188     for y in range(blue.shape[1]-1, blue.shape[1]):
189         blue[x,y]=(blue[x,y-1])/2
190

```

```

191 for x in range(0, blue.shape[0],blue.shape[0]):  

192     for y in range(0, blue.shape[1]):  

193         blue[x,y]=(blue[x+1,y])/2  

194  

195 for x in range(blue.shape[0]-2, blue.shape[0],blue.shape[0]):  

196     for y in range(0, blue.shape[1]):  

197         blue[x,y]=(blue[x+1,y]+blue[x-1,y])/2  

198  

199 for x in range(blue.shape[0]-1, blue.shape[0],blue.shape[0]):  

200     for y in range(2, blue.shape[1]-2):  

201         blue[x,y]=(blue[x-1,y])/2  

202  

203  

204 edge_interpolation = np.zeros((2868,4320,3), 'uint8')#Stacking the Red, Green  

    and Blue Matrices into a Matrix  

205 edge_interpolation [..., 0] = red  

206 edge_interpolation [..., 1] = green  

207 edge_interpolation [..., 2] = blue  

208 img3 = Image.fromarray(edge_interpolation)  

209 img3.save('Edge_Interpolation.jpeg')  

210 plt.imshow(edge_interpolation.astype('uint8'))  

211 plt.show()  

212  

213 plt.subplot(131), plt.imshow(up_NN.astype('uint8'))  

214 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])  

215 plt.subplot(132), plt.imshow(up_BL.astype('uint8'))  

216 plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])  

217 plt.subplot(333), plt.imshow(edge_interpolation.astype('uint8'))  

218 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])  

219 plt.show()  

220  

221 ,,  

222 crop_img1 = up_NN[3000:4000, 0:1000]  

223 plt.imshow(crop_img1.astype('uint8'))  

224 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])  

225 plt.show()  

226  

227 crop_img2 = up_BL[3000:4000, 0:1000]  

228 plt.imshow(crop_img2.astype('uint8'))  

229 plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])  

230 plt.show()  

231  

232 crop_img3 = edge_interpolation[3000:4000, 0:1000]  

233 plt.imshow(crop_img3.astype('uint8'))  

234 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])  

235 plt.show()  

236  

237 crop_img11 = up_NN[3000:4000, 2000:]  

238 plt.imshow(crop_img11.astype('uint8'))  

239 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])  

240 plt.show()  

241  

242 crop_img12 = up_BL[3000:4000, 2000:]  

243 plt.xticks([]), plt.yticks([])  

244 plt.subplot(133), plt.imshow(crop_img13.astype('uint8'))  

245 plt.show()

```

```

246
247 crop_img13 = edge_interpolation[3000:4000, 2000:]
248 plt.imshow(crop_img13.astype('uint8'))
249 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])
250 plt.show()
251
252 crop_img21 = up_NN[2500:3500, 1500:2500]
253 plt.imshow(crop_img21.astype('uint8'))
254 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])
255 plt.show()
256
257 crop_img22 = up_BL[2500:3500, 1500:2500]
258 plt.imshow(crop_img22.astype('uint8'))
259 plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])
260 plt.show()
261
262 crop_img23 = edge_interpolation[2500:3500, 1500:2500]
263 plt.imshow(crop_img23.astype('uint8'))
264 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])
265 plt.show()
266
267
268 plt.subplot(131), plt.imshow(crop_img1.astype('uint8'))
269 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])
270 plt.subplot(132), plt.imshow(crop_img2.astype('uint8'))
271 plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])
272 plt.subplot(133), plt.imshow(crop_img3.astype('uint8'))
273 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])
274 plt.show()
275 plt.subplot(131), plt.imshow(crop_img11.astype('uint8'))
276 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])
277 plt.subplot(132), plt.imshow(crop_img12.astype('uint8'))
278 plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])
279 plt.subplot(133), plt.imshow(crop_img13.astype('uint8'))
280 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])
281 plt.show()
282 plt.subplot(131), plt.imshow(crop_img21.astype('uint8'))
283 plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])
284 plt.subplot(132), plt.imshow(crop_img22.astype('uint8'))
285 plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])
286 plt.subplot(133), plt.imshow(crop_img23.astype('uint8'))
287 plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])
288 plt.show()
289
290 ,,

```

HDR Imaging

```

1 '''Program for HDR Imaging of exposures taken by a camera and tonemapping the
   HDR for display'''
2
3 import rawpy
4 import imageio
5 import cv2
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import ipdb

```

```

9 from PIL import Image
10
11 def weight(z = [] , *args): # Weight function to select particular pixels
12     while throwing out unrequired ones
13     z=(z-np.min(z))/(np.max(z)-np.min(z))
14     z=np.exp(-4*(np.power((z-0.5),2)/0.25))
15     return z
16
17 def exposure_time(k): # This function takes into account the time between
18     exposures taken by the camera
19     t=(1/2048)*np.power(2,k-1)
20     return t
21
22 exposures=['exposure1.nef','exposure2.nef','exposure3.nef','exposure4.nef',''
23             'exposure5.nef','exposure6.nef','exposure7.nef','exposure8.nef','exposure9
24             .nef','exposure10.nef','exposure11.nef','exposure12.nef','exposure13.nef
25             ','exposure14.nef','exposure15.nef','exposure16.nef']
26 #Above is a list containing the exposures
27
28 for count,exp in enumerate(exposures,1): #Iterating through all of the
29     exposures
30     with rawpy.imread(exp) as raw:
31         rgb = raw.postprocess(gamma=(1, 1), no_auto_bright=True, output_bps
32             =16) # Color-balancing and converting the exposures into display-form
33         rgb_down=cv2.resize(rgb,None,fx=0.1, fy=0.1, interpolation= cv2.INTER_CUBIC
34             ) # Downsampling them for faster computation
35         imageio.imsave('processed_exposure'+str(count)+'.tiff',rgb_down) # Saving
36             the processed exposures respectively
37
38 '''Algorithm for obtaining the HDR Image'''
39 #The Saved Processed Exposures in a list
40 filenames=['processed_exposure1.tiff','processed_exposure2.tiff',''
41             'processed_exposure3.tiff','processed_exposure4.tiff','processed_exposure5
42             .tiff','processed_exposure6.tiff','processed_exposure7.tiff',''
43             'processed_exposure8.tiff','processed_exposure9.tiff',''
44             'processed_exposure10.tiff','processed_exposure11.tiff',''
45             'processed_exposure12.tiff','processed_exposure13.tiff',''
46             'processed_exposure14.tiff','processed_exposure15.tiff',''
47             'processed_exposure16.tiff']
48
49 bgr1 = cv2.imread(filenames[0]) # Reading-in the first image so as to
50     initialize the weight and HDR functions in its shape
51 w= np.zeros(bgr1.shape) # Initilazing the weight function as zero
52 hdr=np.zeros(bgr1.shape) # Initilazing the hdr function as zero
53
54 for count,name in enumerate(filenames,1): #Iterating through the processed
55     filenames
56     bgr = cv2.imread(name)
57     bgr_norm = cv2.normalize(bgr, dst=None, alpha=0, beta=1, norm_type=cv2.
58         NORM_MINMAX, dtype=cv2.CV_32F) # Normalizing the functions between 0 and
59         1
60     w+=weight(bgr_norm) # Calculating the weight across every exposure and
61         adding it every time to the final weight function
62     hdr+=weight(bgr_norm)*bgr_norm/exposure_time(count) #HDR Calculation
63         formula
64
65

```

```

43 hdr/=w # Normalizing the HDR by dividing by the total weight function
44 cv2.imwrite("hdr1.hdr", hdr) # Saving te HDR image
45
46 ''' Algorithm for Tone Mapping- User Defined '''
47 I= (hdr/(1+hdr)).astype('float32') # Tone-Mapping to obtain the bright-right
48     hand side of the image and casting into float32
49 cv2.imwrite("HDR-tone.jpg", I*255) # Saving the Tone-mapped Image
50
51 ''' Algorithm for Tone Mapping- OpenCV by Reinhard Tone-Mapping '''
52 tonemap_Reinhard = cv2.createTonemapReinhard(3.0, 0,0,0) # Creating the
53     reinhard tonemapping function
54 Reinhard = tonemap_Reinhard.process(hdr.astype('float32')) # Storing the
55     result in 'Reinhard'
56 cv2.imwrite("Reinhard.jpg", Reinhard*255) # Saving the result
57
58 plt.imshow(cv2.cvtColor(I, cv2.COLOR_BGR2RGB))
59 plt.show()

```

Bilateral

```

1 ''' Program to perform Bilateral Filtering on an image and compare the
2     results with the OpenCV defined function and Gaussian Blur '''
3 import rawpy
4 import imageio
5 import cv2
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import ipdb
9 import math
10 from PIL import Image
11
12 def distance(x, y, p, q): # Distance Function between two points- Euclidean
13     return np.sqrt((x-p)**2 + (y-q)**2)
14
15 def gaussian(x, sigma): # Gaussian Function with zero mean and variance
16     sigma
17     return (1.0 /np.sqrt(2 * math.pi * (sigma ** 2)))* math.exp(- (x ** 2)
18     / (2 * sigma ** 2))
19
20 def weight(image, i, j, k, l, sigma_c, sigma_d): # Weight function to calculate
21     the gaussians across variation in pixel positions and intensities
22     fg = gaussian(image[k][l] - image[i][j], sigma_c) # Function to spread
23     the image intensities across a window in a gaussian curve
24     gs = gaussian(distance(i, j, k, l), sigma_d) # Function to spread the
25     variation in pixel positions from a particular pixel across a window in
26     a gaussian curve
27     w = fg * gs # Dot product of the two functions
28
29     return w
30
31 filename='babylephant.jpg'
32 bgr = cv2.imread(filename) # Reading the image
33 bgr1=cv2.resize(bgr,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_CUBIC) #
34     Downsampling it for faster computation
35 blur = cv2.GaussianBlur(bgr1,(5,5),10) # Applying Gaussian Blur across it

```

```

    with (one case) kernel size 5x5 and sigma 10
27 cv2.imwrite("original_image.png", bgr1) # Saving the original image
28 cv2.imwrite("Guassian-blurred-image.png", blur) # Saving the blurred image
29
30     ''' Function to calculate the Bilateral filter '''
31 def designed_bilateral_filter(image,d,sigma_c,sigma_d): # Takes parameters-
32     Image , Window Diameter , Variance across color , Variance across distance
33     b,g,r = cv2.split(image) # Splitting the image into Red, green & blue
34     channels
35     bilatb=np.zeros(b.shape)
36     bilatg=np.zeros(g.shape)
37     bilatr=np.zeros(r.shape)
38
39     m= int((d-1)/2) # Radius calculation , ie , distance form the center pixel
40     to the corresponding window
41     for i in range(m,b.shape[0]-m): # Starting window at pixel position (m,m)
42         so as to avoid corners
43         for j in range(m,b.shape[1]-m):
44             Wp1=0 # Total weight initlaized to zero every-time the window moves
45             to next pixel
46             Wp2=0
47             Wp3=0
48             for k in range(i-m,m+i+1): # Window function of diameter 'd'
49                 for l in range(j-m,m+j+1):
50                     #print(weight(i,j,k,l))
51                     Wp1+=weight(b,i,j,k,l,sigma_c,sigma_d) # Calculation of weight
52                     function across each channel and adding all of them up (In a particular
53                     Window)
54                     Wp2+=weight(g,i,j,k,l,sigma_c,sigma_d)
55                     Wp3+=weight(r,i,j,k,l,sigma_c,sigma_d)
56                     bilatb[i,j]+=b[k,l]*weight(b,i,j,k,l,sigma_c,sigma_d) #
57                     Calculation of denoised pixel at position (i,j) accross every channel
58                     bilatg[i,j]+=g[k,l]*weight(g,i,j,k,l,sigma_c,sigma_d)
59                     bilatr[i,j]+=r[k,l]*weight(r,i,j,k,l,sigma_c,sigma_d)
60                     if (k==m+i) and (l==m+j):
61                         bilatb[i,j]=int(round(bilatb[i,j]/Wp1)) # At the end of the
62                         window, we must divide by the total wweight function across every pixel
63                         in that window.
64                         bilatg[i,j]=int(round(bilatg[i,j]/Wp2))
65                         bilatr[i,j]=int(round(bilatr[i,j]/Wp3))
66     ''' This is to take care of the corner pixels , just replacing the zeros
67     with the corresponding pixel values across the channels '''
68     for i in range(0,m):
69         for j in range(0,b.shape[1]):
70             bilatb[i,j]=b[i,j]
71             bilatg[i,j]=g[i,j]
72             bilatr[i,j]=r[i,j]
73     for i in range(b.shape[0]-m-1,b.shape[0]):
74         for j in range(0,b.shape[1]):
75             bilatb[i,j]=b[i,j]
76             bilatg[i,j]=g[i,j]
77             bilatr[i,j]=r[i,j]
78     for i in range(0,b.shape[0]):
79         for j in range(0,m):
80             bilatb[i,j]=b[i,j]
81             bilatg[i,j]=g[i,j]

```

```

71     bilatr[i,j]=r[i,j]
72     for i in range(0,b.shape[0]):
73         for j in range(b.shape[1]-m-1,b.shape[1]):
74             bilatb[i,j]=b[i,j]
75             bilatg[i,j]=g[i,j]
76             bilatr[i,j]=r[i,j]
77
78 bilat1 = np.zeros(image.shape) # Stacking all the channels in matrix
79 bilat1[:, :, 0] = bilatb # Red Matrix
80 bilat1[:, :, 1] = bilatg # Green Matrix
81 bilat1[:, :, 2] = bilatr # Red Matrix
82 return bilat1
83
84 bilat = designed_bilateral_filter(bgr1,5,30,10) # Bilateral Filtered Image
85     with filter size 5x5, variace across color=30, variance across distance
86     =10
87
88 #bilat1=cv2.resize(bilat,None,fx=2, fy=2, interpolation= cv2.INTER_CUBIC)
89
90
91 filtered_image_OpenCV = cv2.bilateralFilter(bgr1, 5, 30, 10) # Bilaterally
92     filtered image by OpenCV
93
94 cv2.imwrite("Denoised_image_BI.png", bilat.astype('uint8')) # Saving the
95     image
96
97 plt.imshow(cv2.cvtColor(bgr1.astype('uint8'), cv2.COLOR_BGR2RGB))
98 plt.show()
99 plt.imshow(cv2.cvtColor(blur.astype('uint8'), cv2.COLOR_BGR2RGB))
100 plt.show()
101 plt.imshow(cv2.cvtColor(filtered_image_OpenCV.astype('uint8'), cv2.
102     COLOR_BGR2RGB))
103 plt.show()
104 plt.imshow(cv2.cvtColor(bilat.astype('uint8'), cv2.COLOR_BGR2RGB))
105 plt.show()

```