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".

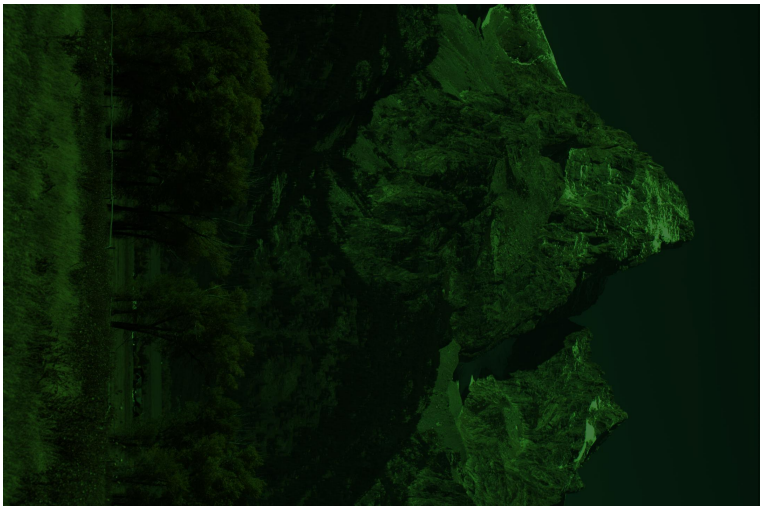
```
############ Section 1 -Demosaicing ################
##### Section 1.2.1 -Read the file by rawpy and save it ############
path = '/home/owner/Documents/HW1_Blackboard/HW1_Blackboard/tetons.nef'
with rawpy.imread(path) as raw:
    rgb = raw.postprocess()
imageio.imsave('tetons original.png', rgb.astype('uint8'))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

original.png

```
############ Section 1 -Demosaicing #################
###### Section 1.2.1 -Read the file by rawpy as a raw-image #############
path = '/home/owner/Documents/HW1_Blackboard/HW1_Blackboard/tetons.nef'
with rawpy.imread(path) as raw:
    rgb = raw.postprocess()

with rawpy.imread(path) as raw:
  raw_image = raw.raw_image.copy()
  raw_image1 = raw.raw_image.copy()
  raw_image2 = raw.raw_image.copy()
  raw_image3 = raw.raw_image.copy()
```

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

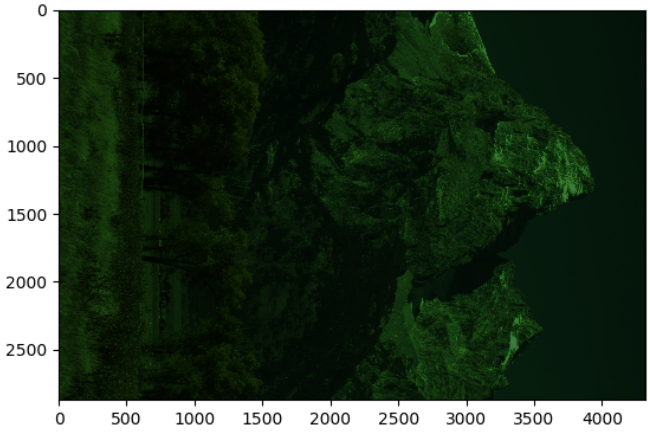
```
    else :
      raw_image1[x,y]=0

'''Algorithm to extract the Blue Pixels as they occur at odd positions
    '''
for x in range(0, raw_image2.shape[0]):
  for y in range(0, raw_image2.shape[1]):
    if ((x%2!=0) and (y%2!=0)):
      raw_image2[x,y]=raw_image2[x,y]
    else :
        raw_image2[x,y]=0


red1= raw_image1.copy() # Red Pixels extractd from the BAYER Matrix
blue1= raw_image2.copy() # Blue Pixels extractd from the BAYER Matrix
green1= raw_image3-red1-blue1 # Green pixels extracted via subtracting
    the Red and Blue Matices from the Original RAW Bayer Pattern

red = (red1/16).astype('float32') # Converting from 12-bit to 8-bit by
    dividing by 2^4
blue = (blue1/16).astype('float32')# Converting from 12-bit to 8-bit by
    dividing by 2^4
green = (green1/16).astype('float32')# Converting from 12-bit to 8-bit
    by dividing by 2^4
down_r2=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_AREA) # Downsampling the red pixels by half
down_b2=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_AREA) # Downsampling the blue pixels by half
down_g2=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_AREA)# Downsampling the green pixels by half
rgbArray2 = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix
    where each of the channels will be stacked at different dimensions
rgbArray2[..., 0] = down_r2  # Red channel in dimension 1
rgbArray2[..., 1] = down_g2 # Green channel in dimension 2
rgbArray2[..., 2] = down_b2 # Blue channel in dimension 3
sub_img = Image.fromarray(rgbArray)
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".

```
############ Section 1 -Demosaicing ################
##### Section 1.3.1 Nearest Neighbours Interpolation ############
down_r2=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_AREA) # Downsampling the red pixels by half
down_b2=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_AREA) # Downsampling the blue pixels by half
down_g2=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_AREA)# Downsampling the green pixels by half
rgbArray2 = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix
    where each of the channels will be stacked at different dimensions
rgbArray2[..., 0] = down_r2  # Red channel in dimension 1
rgbArray2[..., 1] = down_g2 # Green channel in dimension 2
rgbArray2[..., 2] = down_b2 # Blue channel in dimension 3
sub_img = Image.fromarray(rgbArray)
sub_img.save('tetons_subsample.jpeg') # Subsampled Image
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
img2 = Image.fromarray(up_NN)
img2.save('Nearest_Neighbour.jpeg')
plt.imshow(up_NN.astype('uint8'))
plt.show()
```

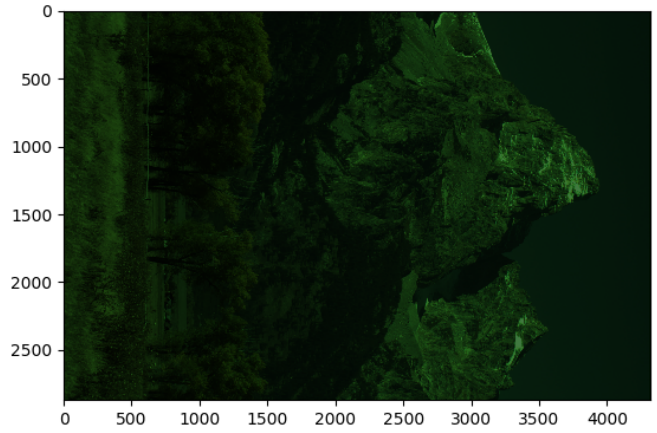


Bilinear Interpolation Demosaicing

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

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

```
down_g=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.
    INTER_LINEAR)# Downsampling the green pixels by half

rgbArray = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix
    where each of the channels will be stacked at different dimensions
rgbArray[..., 0] = down_r # Red channel in dimension 1
rgbArray[..., 1] = down_g # Green channel in dimension 2
rgbArray[..., 2] = down_b # Blue channel in dimension 3

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
img = Image.fromarray(up_BL)
img.save('Bilinear.jpeg')
plt.imshow(up_BL.astype('uint8'))
plt.show()
```



Edge Based Interpolation

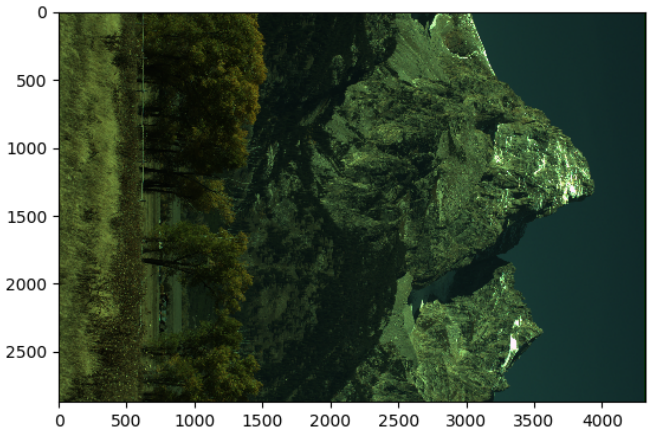
5. Read the paper by Gunturk et al., *Demosaicing: Color Filter Interpolation*, IEEE Signal Processing Magazine, 2005, `http://www.ece.lsu.edu/ipl/papers/IEEE_SPM2005.pdf`. Implement the algorithm of edge-based demosaicing for the green channel in Figure 4. For the R/B channels, feel free to just use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

```
############## Section 1 -Demosaicing ##################
##### Section 1.3.1 Edge Based Demosaicing  #############
# Predicting Green Pixels

for x in range(2, red.shape[0]-2,2): #Predicitng Green Pixels at Red
    pixel positions
  for y in range(2, red.shape[1]-2,2):#Starting from (2,2) so as to
    avoid corners
      dH= (red[x,y-2]+red[x,y+2])/2-red[x,y]
      dV= (red[x-2,y]+red[x+2,y])/2-red[x,y]
      if (dH<dV):
        green[x,y]= (green[x,y-1]+green[x,y+1])/2
      elif (dH>dV) :
```

```
        green[x,y]= (green[x-1,y]+green[x+1,y])/2
      else :
        green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])
    /4


for x in range(3, blue.shape[0]-3,2): #Predicitng Green Pixels at Blue
    pixel positions
  for y in range(3, blue.shape[1]-3,2): #Starting from (3,3) so as to
    avoid corners
    dH= (blue[x,y-2]+blue[x,y+2])/2-blue[x,y]
    dV= (blue[x-2,y]+blue[x+2,y])/2-blue[x,y]
    if (dH<dV):
      green[x,y]= (green[x,y-1]+green[x,y+1])/2
    elif (dH>dV) :
      green[x,y]= (green[x-1,y]+green[x+1,y])/2
    else :
      green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
#### Algorithm to predict Green Pixels at corners and at pixel positions
    not taken into account above ##########
for x in range(1, 2):
  for y in range(1, blue.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(2, red.shape[0]-2, 2):
  for y in range(red.shape[1]-2,red.shape[1]-1):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(red.shape[0]-2, red.shape[0]-1):
  for y in range(2,red.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(1, blue.shape[0]-1,2):
  for y in range(blue.shape[1]-1,blue.shape[1]):
    green[x,y]=(green[x,y-1]+green[x-1,y]+green[x+1,y])/4

for x in range(blue.shape[0]-1, blue.shape[0]):
  for y in range(1,blue.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x-1,y])/4

for x in range(1,blue.shape[0]-1,2):
  for y in range(1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(0, 1):
  for y in range(2, red.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1])/2

for x in range(0, blue.shape[0],2):
  for y in range(0,1):
    green[x,y]=(green[x,y+1])/2

# Predicting Red Pixels by Bilinear Interpolation

for x in range(0, red.shape[0],2):
  for y in range(1, red.shape[1]-1,2):
```

```
    red[x,y]= (red[x,y-1]+red[x,y+1])/2


for x in range(1, red.shape[0]-1,2):
  for y in range(0, red.shape[1],2):
    red[x,y]=(red[x-1,y]+red[x+1,y])/2


for x in range(1, red.shape[0]-1,2):
  for y in range(1, red.shape[1]-1,2):
    red[x,y]=(red[x-1,y]+red[x-1,y+1]+red[x+1,y-1]+red[x+1,y+1])/4

for x in range(red.shape[0]-1, red.shape[0]):
  for y in range(0, red.shape[1],1):
    red[x,y]=(red[x-1,y])/2

for x in range(0, red.shape[0]-1):
  for y in range(4310, red.shape[1],1):
    red[x,y]=525


# Predicting Blue Pixels by Bilinear Interpolation

for x in range(1, blue.shape[0]-1,2):
  for y in range(2, blue.shape[1]-2,2):
    blue[x,y]= (blue[x,y-1]+blue[x,y+1])/2


for x in range(2, blue.shape[0]-2,2):
  for y in range(1, blue.shape[1]-1,2):
    blue[x,y]=(blue[x-1,y]+blue[x+1,y])/2

for x in range(2, blue.shape[0]-2,2):
  for y in range(2, blue.shape[1]-2,2):
    blue[x,y]=(blue[x,y-1]+blue[x,y+1]+blue[x-1,y]+blue[x+1,y])/4

for x in range(1, blue.shape[0],1):
  for y in range(0, blue.shape[1]-2,blue.shape[1]):
    blue[x,y]=(blue[x,y+1])/2

for x in range(1, blue.shape[0],1):
  for y in range(blue.shape[1]-1, blue.shape[1]):
    blue[x,y]=(blue[x,y-1])/2

for x in range(0, blue.shape[0],blue.shape[0]):
  for y in range(0, blue.shape[1]):
    blue[x,y]=(blue[x+1,y])/2

for x in range(blue.shape[0]-2, blue.shape[0],blue.shape[0]):
  for y in range(0, blue.shape[1]):
    blue[x,y]=(blue[x+1,y]+blue[x-1,y])/2

for x in range(blue.shape[0]-1, blue.shape[0],blue.shape[0]):
  for y in range(2, blue.shape[1]-2):
    blue[x,y]=(blue[x-1,y])/2

```

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

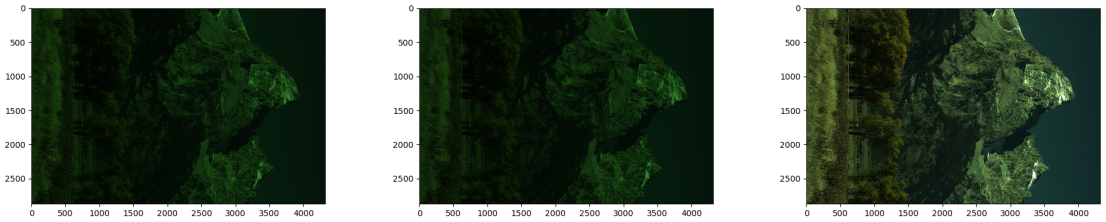


Fig1. Nearest Neighbours Demosaicing,Bilinear Demosaicing and Edge-Based Demosaicing

Variation between the 3 methods across 2 regions

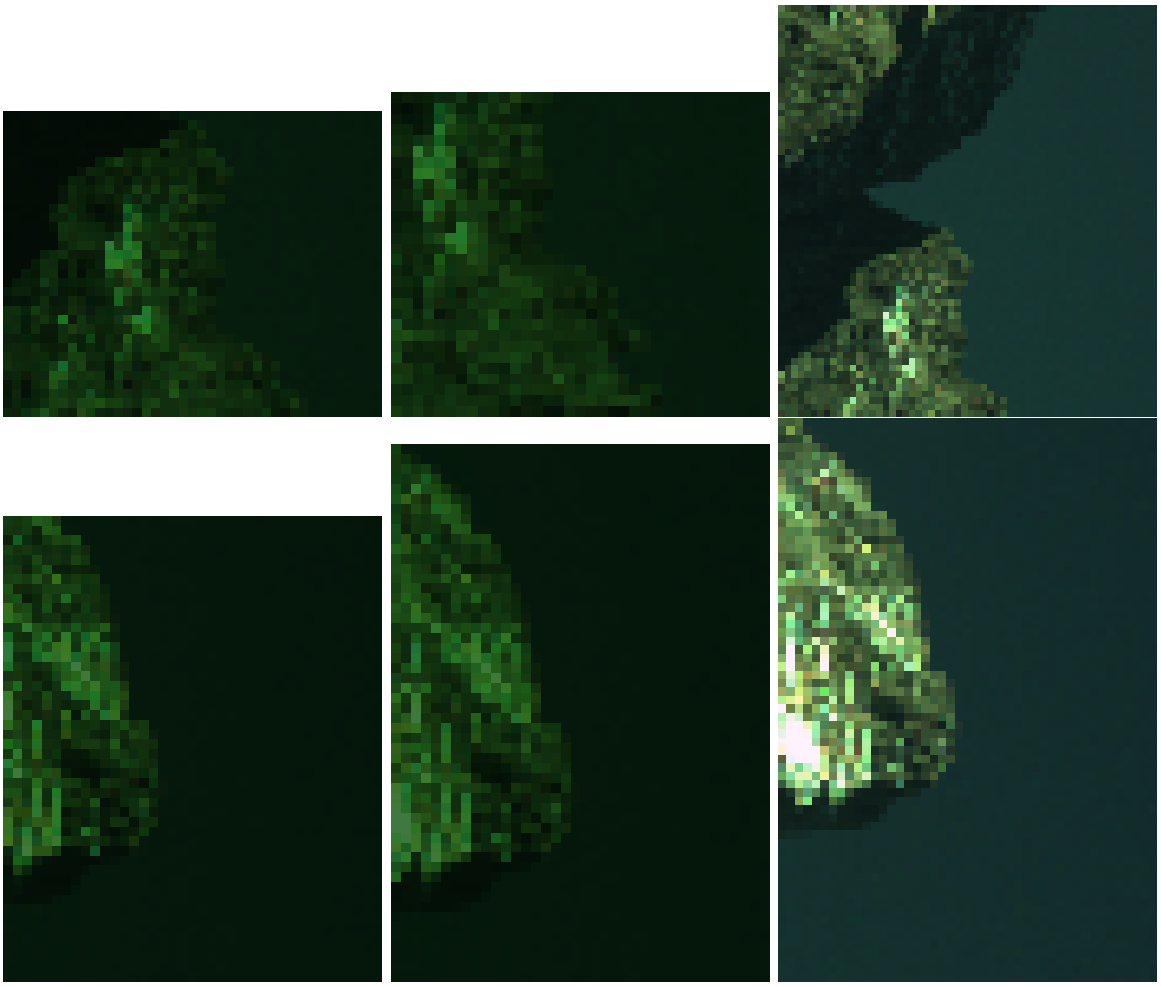


Fig2. Nearest Neighbours Demosaicing,Bilinear Demosaicing and Edge-Based Demosaicing

As we can see, the demosaicing through Nearest neighbours is slightly more pixelated and unclear than the bilinear interpolation demosacing, 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

```
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 $k = 1, \cdots, 16$. For the rest of the problem, you will just work on these processed images.

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

```
### Section 1.2.1 -Read the RAW files by rawpy and save the exposures ###

exposures=['exposure1.nef','exposure2.nef','exposure3.nef','exposure4.nef','exposure5.nef','exposure6.nef','exposure7.nef','exposure8.nef','exposure9.nef','exposure10.nef','exposure11.nef','exposure12.nef','exposure13.nef','exposure14.nef','exposure15.nef','exposure16.nef']
#Above is a list containing the exposures

for count,exp in enumerate(exposures,1): #Iterating through all of the exposures
  with rawpy.imread(exp) as raw:
    rgb = raw.postprocess(gamma=(1,1), no_auto_bright=True, output_bps =16) # Color-balancing and converting the exposures into display-form
  rgb_down=cv2.resize(rgb,None,fx=0.1, fy=0.1, interpolation= cv2.INTER_CUBIC) # Downsampling them for faster computation
  imageio.imsave('processed_exposure'+str(count)+'.tiff',rgb_down) # Saving the processed exposures respectively
```

2. Calculate the HDR image using the following formula:

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

where the weighting between exposure frames is:

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

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

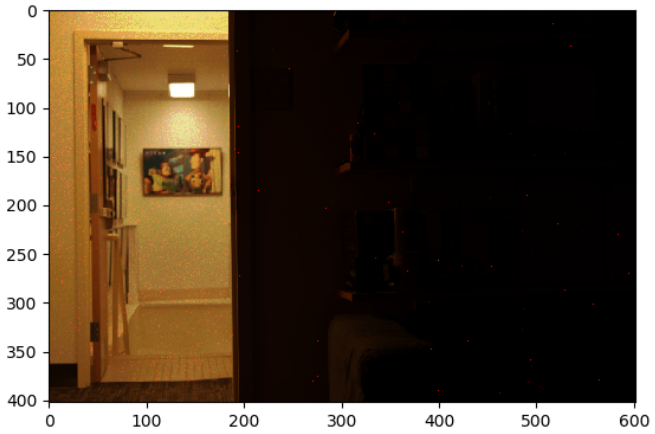
HDR Tone-Mapping without in-built

```
############ Section 2 -HDR Imaging ################
### Section 1.2.1 -Calculating the HDR Image #######

def weight(z = [], *args): # Weight function to select particular pixels while throwing out unrequired ones
  z=(z-np.min(z))/(np.max(z)-np.min(z))
  z=np.exp(-4*(np.power((z-0.5),2)/0.25))
  return z

def exposure_time(k): # This function takes into account the time between exposures taken by the camera
  t=(1/2048)*np.power(2,k-1)
  return t

#The Saved Processed Exposures in a list
filenames=['processed_exposure1.tiff','processed_exposure2.tiff','processed_exposure3.tiff','processed_exposure4.tiff','processed_exposure5.tiff','processed_exposure6.tiff','processed_exposure7.tiff','processed_exposure8.tiff','processed_exposure9.tiff','processed_exposure10.tiff','processed_exposure11.tiff','processed_exposure12.tiff','processed_exposure13.tiff','processed_exposure14.tiff','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_phototonemap.png".

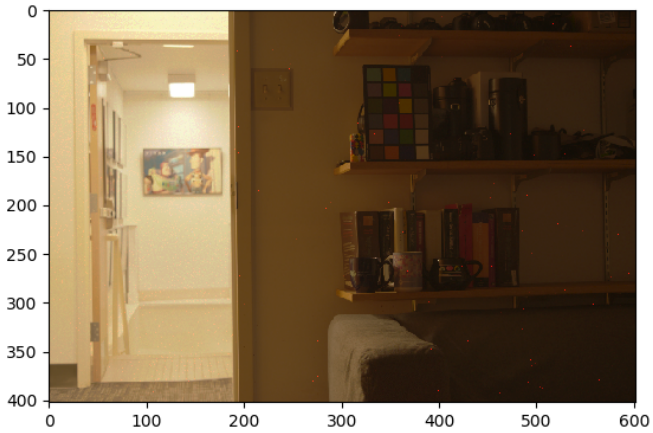
```
1 ############ Section 2 -HDR Imaging ################
2 ### Section 1.2.1 -Displaying with Tonemapping (User-Defined) #######
3
4 ''' Algorithm for Tone Mapping- User Defined '''
5 I= (hdr/(1+hdr)).astype('float32') # Tone-Mapping to obtain the bright-
       right hand side of the image and casting 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.

```
############ Section 2 -HDR Imaging ################
### Section 1.2.1 -Displaying with Tonemapping (Built-in ) #######
############ Reinhard Tonemapping ####################

tonemap_Reinhard = cv2.createTonemapReinhard(3.0, 0,0,0)
Reinhard = tonemap_Reinhard.process(hdr.astype('float32'))
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.

```
############ Section 2 -Bilateral Filter ################
### Section 1.2.1 -Read in the Image & #######
############ Gaussian- Blur it ####################

filename='babyelephant.jpg'
bgr = cv2.imread(filename) # Reading the image
bgr1=cv2.resize(bgr,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_CUBIC)
    # Downsampling it for faster computation
blur = cv2.GaussianBlur(bgr1,(5,5),10) # Applying Gaussian Blur across
   it with (one case) kernel size 5x5 and sigma 10
cv2.imwrite("original_image.png", bgr1) # Saving the original image
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.

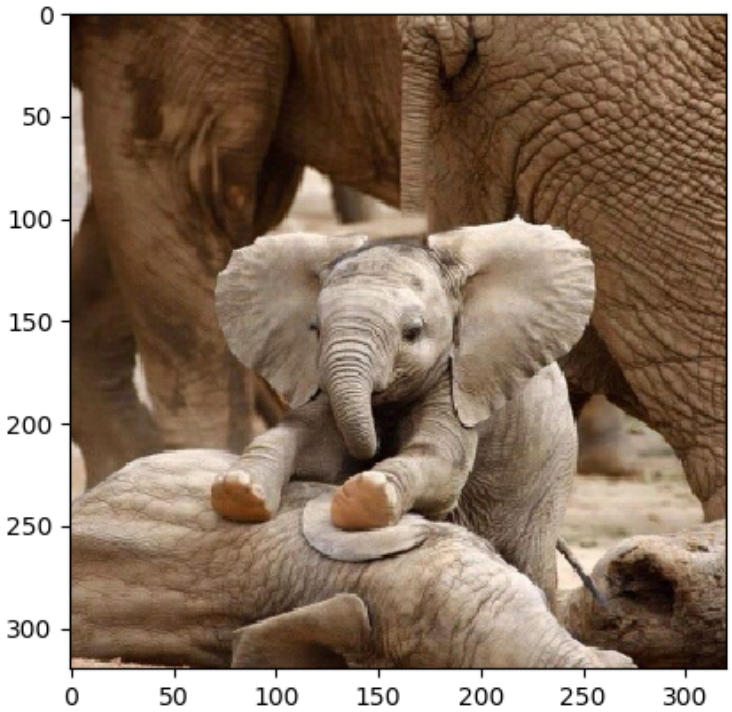


Figure 1: Original Image

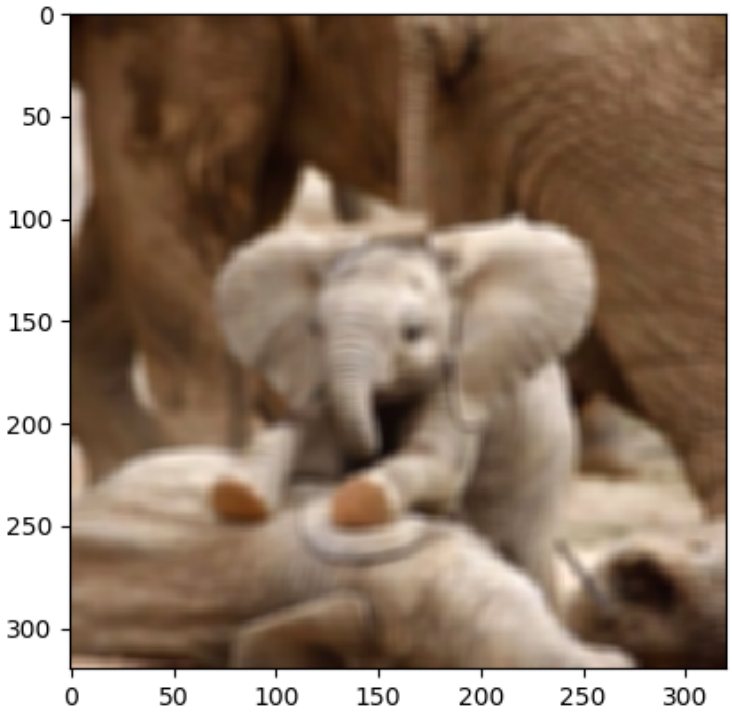


Figure 2: Original Image

```
############ Section 2 -Bilateral Filter ################
### Section 1.2.1 -Implementing Bilateral Filter #######
############ User- Defined  ####################

''' Function to calculate the Bilateral filter'''''
def designed_bilateral_filter(image,d,sigma_c,sigma_d): # Takes parameters-Image, Window Diameter, Variance across color, Variance across distance
  b,g,r = cv2.split(image) # Splitting the image into Red, green & blue channels
  bilatb=np.zeros(b.shape)
  bilatg=np.zeros(g.shape)
  bilatr=np.zeros(r.shape)

  m= int((d-1)/2) # Radius calulation, ie, distance form the center pixel to the corresponding window
  for i in range(m,b.shape[0]-m): # Starting window at pixel position (m,m) so as to avoid corners
    for j in range(m,b.shape[1]-m):
      Wp1=0 # Total weight initlaized to zero every-time the window moves to next pixel
      Wp2=0
      Wp3=0
      for k in range(i-m,m+i+1):     # Window function of diameter 'd'
        for l in range(j-m,m+j+1):
          #print(weight(i,j,k,l))
          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)
          Wp2+=weight(g,i,j,k,l,sigma_c,sigma_d)
          Wp3+=weight(r,i,j,k,l,sigma_c,sigma_d)
          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
          bilatg[i,j]+=g[k,l]*weight(g,i,j,k,l,sigma_c,sigma_d)
          bilatr[i,j]+=r[k,l]*weight(r,i,j,k,l,sigma_c,sigma_d)
          if (k==m+i) and (l==m+j):
            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.
            bilatg[i,j]=int(round(bilatg[i,j]/Wp2))
            bilatr[i,j]=int(round(bilatr[i,j]/Wp3))
''' This is to take care of the corner pixels, just replacing the zeros with the corresponding pixel values across the channels'''
  for i in range(0,m):
    for j in range(0,b.shape[1]):
      bilatb[i,j]=b[i,j]
    bilatg[i,j]=g[i,j]
    bilatr[i,j]=r[i,j]
  for i in range(b.shape[0]-m-1,b.shape[0]):
    for j in range(0,b.shape[1]):
      bilatb[i,j]=b[i,j]
      bilatg[i,j]=g[i,j]
      bilatr[i,j]=r[i,j]
  for i in range(0,b.shape[0]):
    for j in range(0,m):
```

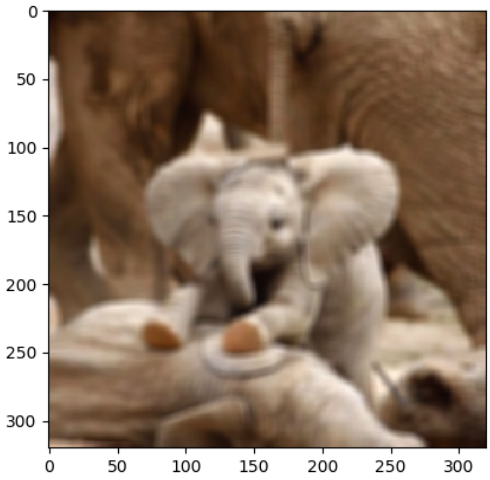


Figure 3: Guassian Blurred Image (Kernel- 9x9)

```
      bilatb[i,j]=b[i,j]
      bilatg[i,j]=g[i,j]
      bilatr[i,j]=r[i,j]
  for i in range(0,b.shape[0]):
    for j in range(b.shape[1]-m-1,b.shape[1]):
      bilatb[i,j]=b[i,j]
      bilatg[i,j]=g[i,j]
      bilatr[i,j]=r[i,j]

  bilat1 = np.zeros(image.shape) # Stacking all the channels in matrix
  bilat1[..., 0] = bilatb # Red Matrix
  bilat1[..., 1] = bilatg # Green Matrix
  bilat1[..., 2] = bilatr # Red Matrix
  return bilat1

bilat = designed_bilateral_filter(bgr1,5,30,10) # Bilateral Filtered
    Image with filter size 5x5, variace across color=30, variance across
     distance=10
cv2.imwrite("Denoised_image_BI.png", bilat.astype('uint8')) # Saving the
    image
```

```
############ Section 2 -Bilateral Filter ################
### Section 1.2.1 -Implement Bilateral Filter #######
############ In-Built Function ####################

filtered_image_OpenCV = cv2.bilateralFilter(bgr1, 5, 30, 10)

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

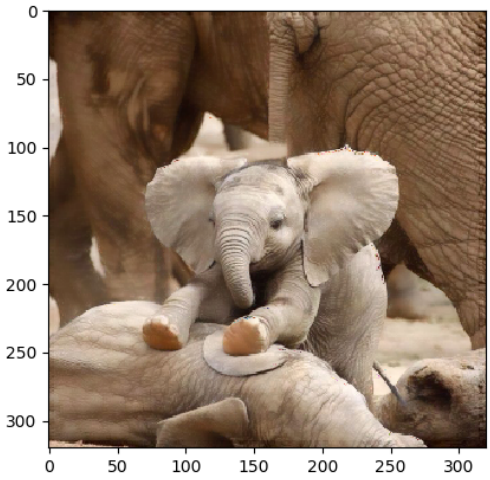


Figure 4: Implemented Bilateral Filtered Image (Filter size=9, Sigma color=30, Sigma distance=50)

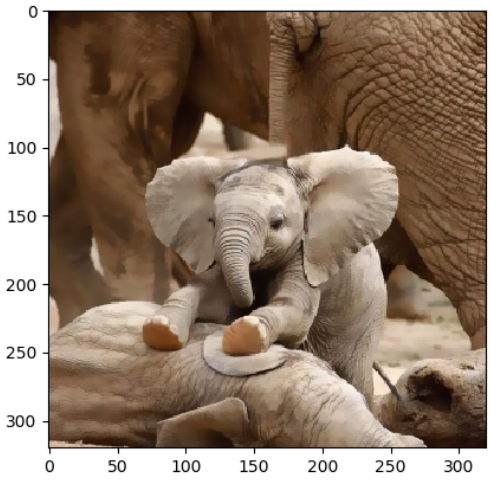


Figure 5: OpenCV Bilateral Filtered Image (Filter size=9, Sigma color=30, Sigma distance=50)

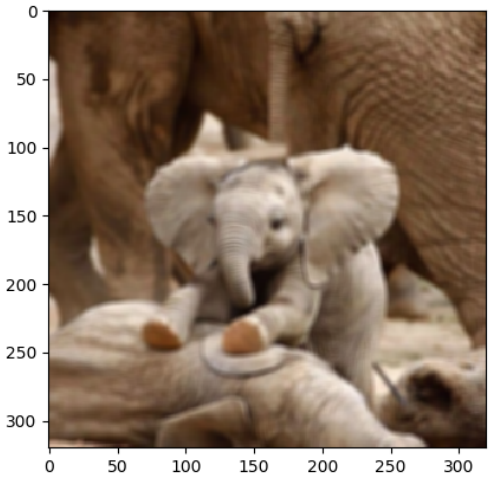


Figure 6: Guassian Blurred Image (Kernel- 7x7)

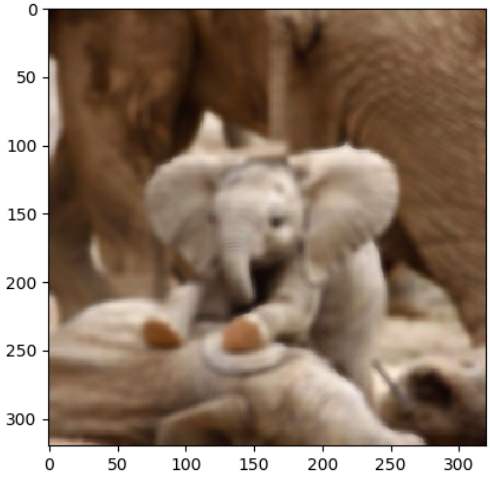


Figure 7: Implemented Bilateral Filtered Image (Filter size=7, Sigma color=500, Sigma distance=50)

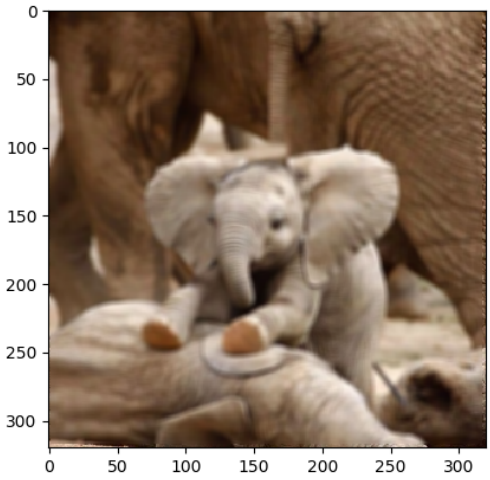


Figure 8: OpenCV Bilateral Filtered Image (Filter size=9, Sigma color=500, Sigma distance=50)

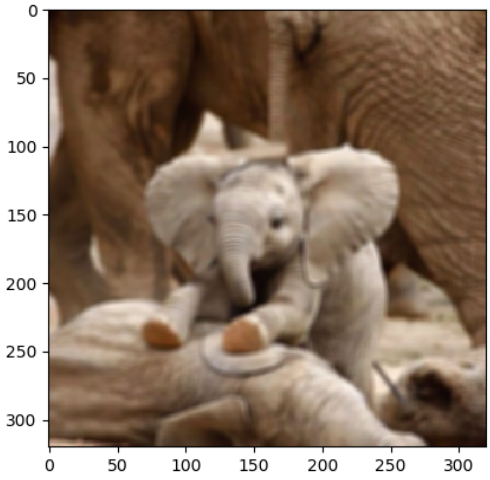


Figure 9: Guassian Blurred Image (Kernel- 5x5)

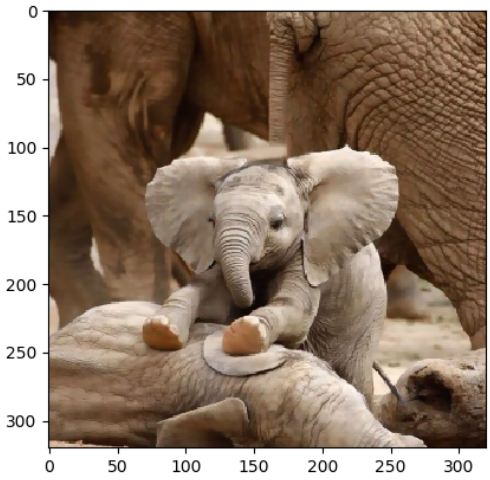


Figure 10: Implemented Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=50)

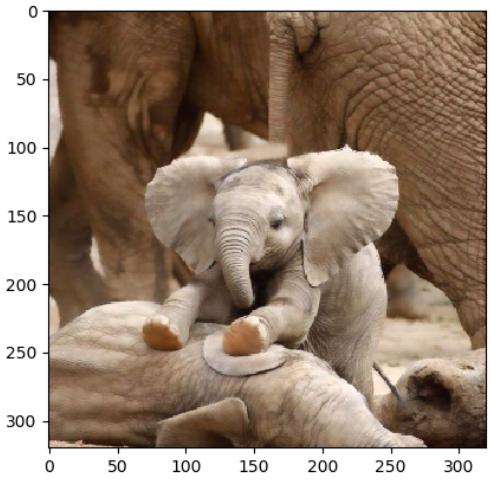


Figure 11: OpenCV Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=50)

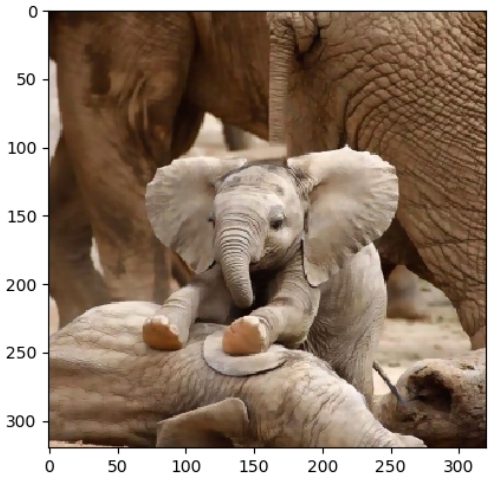


Figure 12: Implemented Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=10)

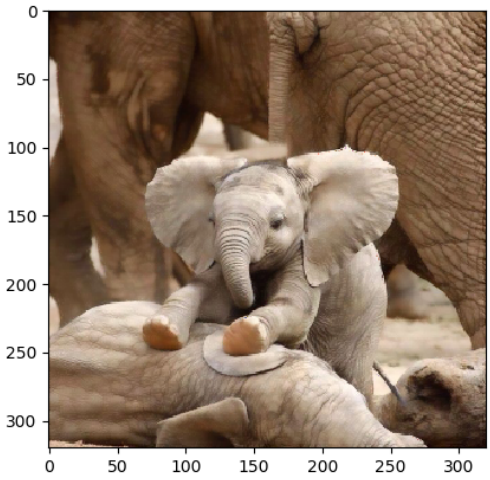


Figure 13: OpenCV Bilateral Filtered Image (Filter size=5, Sigma color=30, Sigma distance=10)

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

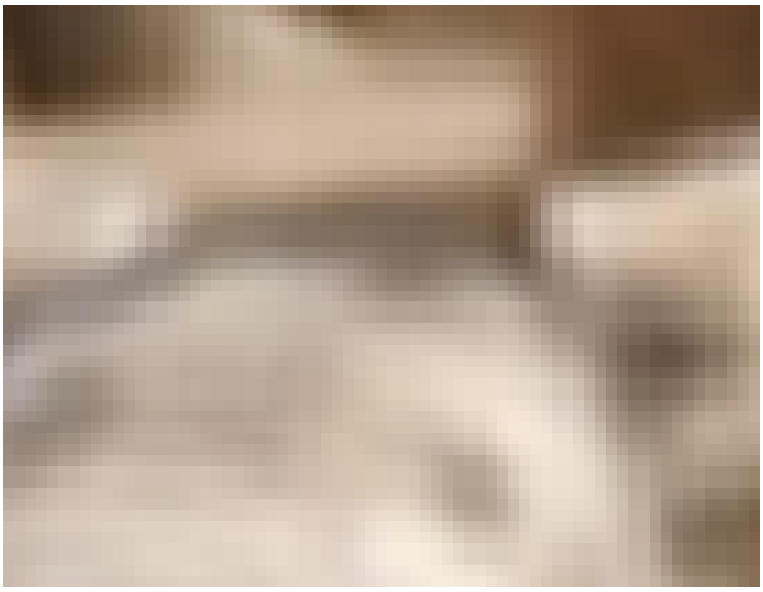


Figure 15: Guassian Blurred Image (Kernel- 5x5)

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*

```
''' Program to Perfrom Demosaicing via 3 methods- Nearest Neighbours, Bilinear Interpolation and Edge-Based Interpolation Demosaicing'''

import rawpy
import imageio
import cv2
import numpy as np
import matplotlib.pyplot as plt
import ipdb
from PIL import Image
import colorsys


path = '/home/owner/Documents/HW1_Blackboard/HW1_Blackboard/tetons.nef'
with rawpy.imread(path) as raw:
    rgb = raw.postprocess()    # Demosaicing done by the RawPy package to convert the RGGB Bayer Matrix into a 3-D Image

imageio.imsave('tetons original.png', (rgb).astype('uint8')) #That Image is saved

with rawpy.imread(path) as raw: ## To Perform Demoisaicing across 3 methods, so weneed to extract the RAw image (Bayer Matrix)
  raw_image = raw.raw_image.copy() # Creating copies of the RAW matrix in 3 different variables so as to extract Reg, Green and Blue chennels from them
  raw_image1 = raw.raw_image.copy()
  raw_image2 = raw.raw_image.copy()
  raw_image3 = raw.raw_image.copy()



'''Algorithm to extract the Red Pixels as they occur at only even positions '''
for x in range(0, raw_image1.shape[0]):
  for y in range(0, raw_image1.shape[1]):
    if ((x%2==0) and (y%2==0)):
      raw_image1[x,y]=raw_image1[x,y]
    else :
      raw_image1[x,y]=0

'''Algorithm to extract the Blue Pixels as they occur at odd positions'''
for x in range(0, raw_image2.shape[0]):
  for y in range(0, raw_image2.shape[1]):
    if ((x%2!=0) and (y%2!=0)):
      raw_image2[x,y]=raw_image2[x,y]
    else :
        raw_image2[x,y]=0


red1= raw_image1.copy() # Red Pixels extractd from the BAYER Matrix
```

```
blue1= raw_image2.copy() # Blue Pixels extractd from the BAYER Matrix
green1= raw_image3-red1-blue1 # Green pixels extracted via subtracting the Red and Blue Matices from the Original RAW Bayer Pattern

red = (red1/16).astype('float32') # Converting from 12-bit to 8-bit by dividing by 2^4
blue = (blue1/16).astype('float32')# Converting from 12-bit to 8-bit by dividing by 2^4
green = (green1/16).astype('float32')# Converting from 12-bit to 8-bit by dividing by 2^4

'''Algorithm to perform downsampling then upsampling by Bilinear Interpolation Demosaicing'''
down_r=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR) # Downsampling the red pixels by half
down_b=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR)# Downsampling the blue pixels by half
down_g=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_LINEAR) # Downsampling the green pixels by half

rgbArray = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix where each of the channels will be stacked at different dimensions
rgbArray[..., 0] = down_r # Red channel in dimension 1
rgbArray[..., 1] = down_g # Green channel in dimension 2
rgbArray[..., 2] = down_b # Blue channel in dimension 3

sub_img = Image.fromarray(rgbArray)
sub_img.save('tetons_subsample.jpeg') # Subsampled Image

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
img = Image.fromarray(up_BL)
img.save('Bilinear.jpeg')
plt.imshow(up_BL.astype('uint8'))
plt.show()

'''Algorithm to perform downsampling then upsampling by Nearest Neighbours Demosaicing'''
down_r2=cv2.resize(red,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) # Downsampling the red pixels by half
down_b2=cv2.resize(blue,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA) # Downsampling the blue pixels by half
down_g2=cv2.resize(green,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_AREA)# Downsampling the green pixels by half
rgbArray2 = np.zeros((1434,2160,3), 'uint8') # Creating a zero matrix where each of the channels will be stacked at different dimensions
rgbArray2[..., 0] = down_r2  # Red channel in dimension 1
rgbArray2[..., 1] = down_g2 # Green channel in dimension 2
rgbArray2[..., 2] = down_b2 # Blue channel in dimension 3
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
img2 = Image.fromarray(up_NN)
img2.save('Nearest_Neighbour.jpeg')
plt.imshow(up_NN.astype('uint8'))
plt.show()

```

```
'''Algorithm to perform downsampling then upsampling by Edge-Based
    Demosaicing'''
# Predicting Green Pixels

for x in range(2, red.shape[0]-2,2): #Predicitng Green Pixels at Red pixel
    positions
  for y in range(2, red.shape[1]-2,2):#Starting from (2,2) so as to avoid
    corners
      dH= (red[x,y-2]+red[x,y+2])/2-red[x,y]
      dV= (red[x-2,y]+red[x+2,y])/2-red[x,y]
      if (dH<dV):
        green[x,y]= (green[x,y-1]+green[x,y+1])/2
      elif (dH>dV) :
        green[x,y]= (green[x-1,y]+green[x+1,y])/2
      else :
        green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4


for x in range(3, blue.shape[0]-3,2): #Predicitng Green Pixels at Blue pixel
    positions
  for y in range(3, blue.shape[1]-3,2): #Starting from (3,3) so as to avoid
    corners
    dH= (blue[x,y-2]+blue[x,y+2])/2-blue[x,y]
    dV= (blue[x-2,y]+blue[x+2,y])/2-blue[x,y]
    if (dH<dV):
      green[x,y]= (green[x,y-1]+green[x,y+1])/2
    elif (dH>dV) :
      green[x,y]= (green[x-1,y]+green[x+1,y])/2
    else :
      green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4
#### Algorithm to predict Green Pixels at corners and at pixel positions not
    taken into account above ###########
for x in range(1, 2):
  for y in range(1, blue.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(2, red.shape[0]-2, 2):
  for y in range(red.shape[1]-2,red.shape[1]-1):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(red.shape[0]-2, red.shape[0]-1):
  for y in range(2,red.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

for x in range(1, blue.shape[0]-1,2):
  for y in range(blue.shape[1]-1,blue.shape[1]):
    green[x,y]=(green[x,y-1]+green[x-1,y]+green[x+1,y])/4

for x in range(blue.shape[0]-1, blue.shape[0]):
  for y in range(1,blue.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x-1,y])/4

for x in range(1,blue.shape[0]-1,2):
  for y in range(1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1]+green[x-1,y]+green[x+1,y])/4

```

```
for x in range(0, 1):
  for y in range(2, red.shape[1]-1,2):
    green[x,y]=(green[x,y-1]+green[x,y+1])/2

for x in range(0, blue.shape[0],2):
  for y in range(0,1):
    green[x,y]=(green[x,y+1])/2

# Predicting Red Pixels by Bilinear Interpolation

for x in range(0, red.shape[0],2):
  for y in range(1, red.shape[1]-1,2):
    red[x,y]= (red[x,y-1]+red[x,y+1])/2


for x in range(1, red.shape[0]-1,2):
  for y in range(0, red.shape[1],2):
    red[x,y]=(red[x-1,y]+red[x+1,y])/2


for x in range(1, red.shape[0]-1,2):
  for y in range(1, red.shape[1]-1,2):
    red[x,y]=(red[x-1,y]+red[x-1,y+1]+red[x+1,y-1]+red[x+1,y+1])/4

for x in range(red.shape[0]-1, red.shape[0]):
  for y in range(0, red.shape[1],1):
    red[x,y]=(red[x-1,y])/2

for x in range(0, red.shape[0]-1):
  for y in range(4310, red.shape[1],1):
    red[x,y]=525


# Predicting Blue Pixels by Bilinear Interpolation

for x in range(1, blue.shape[0]-1,2):
  for y in range(2, blue.shape[1]-2,2):
    blue[x,y]= (blue[x,y-1]+blue[x,y+1])/2


for x in range(2, blue.shape[0]-2,2):
  for y in range(1, blue.shape[1]-1,2):
    blue[x,y]=(blue[x-1,y]+blue[x+1,y])/2

for x in range(2, blue.shape[0]-2,2):
  for y in range(2, blue.shape[1]-2,2):
    blue[x,y]=(blue[x,y-1]+blue[x,y+1]+blue[x-1,y]+blue[x+1,y])/4

for x in range(1, blue.shape[0],1):
  for y in range(0, blue.shape[1]-2,blue.shape[1]):
    blue[x,y]=(blue[x,y+1])/2

for x in range(1, blue.shape[0],1):
  for y in range(blue.shape[1]-1, blue.shape[1]):
    blue[x,y]=(blue[x,y-1])/2

```

```
for x in range(0, blue.shape[0],blue.shape[0]):
  for y in range(0, blue.shape[1]):
    blue[x,y]=(blue[x+1,y])/2

for x in range(blue.shape[0]-2, blue.shape[0],blue.shape[0]):
  for y in range(0, blue.shape[1]):
    blue[x,y]=(blue[x+1,y]+blue[x-1,y])/2

for x in range(blue.shape[0]-1, blue.shape[0],blue.shape[0]):
  for y in range(2, blue.shape[1]-2):
    blue[x,y]=(blue[x-1,y])/2


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

plt.subplot(131), plt.imshow(up_NN.astype('uint8'))
plt.title('Nearest Neighbour '), plt.xticks([]), plt.yticks([])
plt.subplot(132), plt.imshow(up_BL.astype('uint8'))
plt.title('Bilinear Interpolation '), plt.xticks([]), plt.yticks([])
plt.subplot(333), plt.imshow(edge_interpolation.astype('uint8'))
plt.title('Edge-Based'), plt.xticks([]), plt.yticks([])
plt.show()

'''
crop_img1 = up_NN[3000:4000, 0:1000]
plt.imshow(crop_img1.astype('uint8'))
plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])
plt.show()

crop_img2 = up_BL[3000:4000, 0:1000]
plt.imshow(crop_img2.astype('uint8'))
plt.title('Bilinear Interpolation'), plt.xticks([]), plt.yticks([])
plt.show()

crop_img3 = edge_interpolation[3000:4000, 0:1000]
plt.imshow(crop_img3.astype('uint8'))
plt.title('Edge-Base'), plt.xticks([]), plt.yticks([])
plt.show()

crop_img11 = up_NN[3000:4000, 2000:]
plt.imshow(crop_img11.astype('uint8'))
plt.title('Nearest Neighbour'), plt.xticks([]), plt.yticks([])
plt.show()

crop_img12 = up_BL[3000:4000, 2000:]
plt.xticks([]), plt.yticks([])
plt.subplot(133), plt.imshow(crop_img13.astype('uint8'))
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-Base'), 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
```

```
from PIL import Image

def weight(z = [], *args): # Weight function to select particular pixels while throwing out unrequired ones
  z=(z-np.min(z))/(np.max(z)-np.min(z))
  z=np.exp(-4*(np.power((z-0.5),2)/0.25))
  return z

def exposure_time(k): # This function takes into account the time between exposures taken by the camera
  t=(1/2048)*np.power(2,k-1)
  return t

exposures=['exposure1.nef','exposure2.nef','exposure3.nef','exposure4.nef','exposure5.nef','exposure6.nef','exposure7.nef','exposure8.nef','exposure9.nef','exposure10.nef','exposure11.nef','exposure12.nef','exposure13.nef','exposure14.nef','exposure15.nef','exposure16.nef']
#Above is a list containing the exposures

for count,exp in enumerate(exposures,1): #Iterating through all of the exposures
  with rawpy.imread(exp) as raw:
    rgb = raw.postprocess(gamma=(1,1), no_auto_bright=True, output_bps=16) # Color-balancing and converting the exposures into display-form
  rgb_down=cv2.resize(rgb,None,fx=0.1, fy=0.1, interpolation= cv2.INTER_CUBIC) # Downsampling them for faster computation
  imageio.imsave('processed_exposure'+str(count)+'.tiff',rgb_down) # Saving the processed exposures respectively

'''Algorithm for obtaining the HDR Image'''
#The Saved Processed Exposures in a list
filenames=['processed_exposure1.tiff','processed_exposure2.tiff','processed_exposure3.tiff','processed_exposure4.tiff','processed_exposure5.tiff','processed_exposure6.tiff','processed_exposure7.tiff','processed_exposure8.tiff','processed_exposure9.tiff','processed_exposure10.tiff','processed_exposure11.tiff','processed_exposure12.tiff','processed_exposure13.tiff','processed_exposure14.tiff','processed_exposure15.tiff','processed_exposure16.tiff']

bgr1 = cv2.imread(filenames[0]) # Reading-in the first image so as to initialize the weight and HDR functions in its shape
w= np.zeros(bgr1.shape) # Initilazing the weight function as zero
hdr=np.zeros(bgr1.shape) # Initilazing the hdr function as zero

for count,name in enumerate(filenames,1): #Iterating through the processed filenames
  bgr = cv2.imread(name)
  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
  w+=weight(bgr_norm) # Calculating the weight across every exposure and adding it every time to the final weight function
  hdr+=weight(bgr_norm)*bgr_norm/exposure_time(count) #HDR Calculation formula

```

```
hdr/=w # Normalizing the HDR by dividing by the total weight function
cv2.imwrite("hdr1.hdr", hdr) # Saving te HDR image

''' Algorithm for Tone Mapping- User Defined'''
I= (hdr/(1+hdr)).astype('float32') # Tone-Mapping to obtain the bright-right hand side of the image and casting into float32

cv2.imwrite("HDR_tone.jpg", I*255) # Saving the Tone-mapped Image

''' Algorithm for Tone Mapping- OpenCV by Reinhard Tone-Mapping '''
tonemap_Reinhard = cv2.createTonemapReinhard(3.0, 0,0,0) # Creating the reinhard tonemapping function
Reinhard = tonemap_Reinhard.process(hdr.astype('float32')) # Storing the result in 'Reinhard'
cv2.imwrite("Reinhard.jpg", Reinhard*255) # Saving the result

plt.imshow(cv2.cvtColor(I, cv2.COLOR_BGR2RGB))
plt.show()
plt.imshow(cv2.cvtColor(Reinhard, cv2.COLOR_BGR2RGB))
plt.show()
```

Bilateral

```
''' Program to perform Bilateral Filtering on an image and compare the results with the OpenCV defined function and Gaussian Blur'''
import rawpy
import imageio
import cv2
import numpy as np
import matplotlib.pyplot as plt
import ipdb
import math
from PIL import Image

def distance(x, y, p, q): # Distance Function between two points- Euclidean
    return np.sqrt((x-p)**2 + (y-q)**2)

def gaussian(x, sigma): # Gaussian Function with zero mean and variance sigma
    return (1.0 /np.sqrt(2 * math.pi * (sigma ** 2)))* math.exp(- (x ** 2) / (2 * sigma ** 2))

def weight(image,i,j,k,l,sigma_c,sigma_d): # Weight function to calculate the gaussians across variation in pixel positions and intensities
  fg = gaussian(image[k][l] - image[i][j], sigma_c) # Function to spread the image intensities across a window in a gaussian curve
  gs = gaussian(distance(i, j, k, l), sigma_d) # Function to spread the varaition in pixel positions from a particular pixel across a window in a gaussian curve
  w = fg * gs # Dot product of the two functions
  return w

filename='babyelephant.jpg'
bgr = cv2.imread(filename) # Reading the image
bgr1=cv2.resize(bgr,None,fx=0.5, fy=0.5, interpolation= cv2.INTER_CUBIC) # Downsampling it for faster computation
blur = cv2.GaussianBlur(bgr1,(5,5),10) # Applying Gaussian Blur across it
```

```
    with (one case) kernel size 5x5 and sigma 10
cv2.imwrite("original_image.png", bgr1) # Saving the original image
cv2.imwrite("Guassian_blurred_image.png", blur) # Saving the blurred image

''' Function to calculate the Bilateral filter '''''
def designed_bilateral_filter(image,d,sigma_c,sigma_d): # Takes parameters- Image, Window Diameter, Variance across color, Variance across distance
  b,g,r = cv2.split(image) # Splitting the image into Red, green & blue channels
  bilatb=np.zeros(b.shape)
  bilatg=np.zeros(g.shape)
  bilatr=np.zeros(r.shape)

  m= int((d-1)/2) # Radius calulation, ie, distance form the center pixel to the corresponding window
  for i in range(m,b.shape[0]-m): # Starting window at pixel position (m,m) so as to avoid corners
    for j in range(m,b.shape[1]-m):
      Wp1=0 # Total weight initlaized to zero every-time the window moves to next pixel
      Wp2=0
      Wp3=0
      for k in range(i-m,m+i+1):     # Window function of diameter 'd'
        for l in range(j-m,m+j+1):
          #print(weight(i,j,k,l))
          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)
          Wp2+=weight(g,i,j,k,l,sigma_c,sigma_d)
          Wp3+=weight(r,i,j,k,l,sigma_c,sigma_d)
          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
          bilatg[i,j]+=g[k,l]*weight(g,i,j,k,l,sigma_c,sigma_d)
          bilatr[i,j]+=r[k,l]*weight(r,i,j,k,l,sigma_c,sigma_d)
          if (k==m+i) and (l==m+j):
            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.
            bilatg[i,j]=int(round(bilatg[i,j]/Wp2))
            bilatr[i,j]=int(round(bilatr[i,j]/Wp3))
''' This is to take care of the corner pixels, just replacing the zeros with the corresponding pixel values across the channels'''
  for i in range(0,m):
    for j in range(0,b.shape[1]):
      bilatb[i,j]=b[i,j]
    bilatg[i,j]=g[i,j]
    bilatr[i,j]=r[i,j]
  for i in range(b.shape[0]-m-1,b.shape[0]):
    for j in range(0,b.shape[1]):
      bilatb[i,j]=b[i,j]
      bilatg[i,j]=g[i,j]
      bilatr[i,j]=r[i,j]
  for i in range(0,b.shape[0]):
    for j in range(0,m):
      bilatb[i,j]=b[i,j]
      bilatg[i,j]=g[i,j]
```

```
        bilatr[i,j]=r[i,j]
    for i in range(0,b.shape[0]):
      for j in range(b.shape[1]-m-1,b.shape[1]):
        bilatb[i,j]=b[i,j]
        bilatg[i,j]=g[i,j]
        bilatr[i,j]=r[i,j]

    bilat1 = np.zeros(image.shape) # Stacking all the channels in matrix
    bilat1[..., 0] = bilatb # Red Matrix
    bilat1[..., 1] = bilatg # Green Matrix
    bilat1[..., 2] = bilatr # Red Matrix
    return bilat1

bilat = designed_bilateral_filter(bgr1,5,30,10) # Bilateral Filtered Image
    with filter size 5x5, variace across color=30, variance across distance
    =10

#bilat1=cv2.resize(bilat,None,fx=2, fy=2, interpolation= cv2.INTER_CUBIC)

cv2.imwrite("Denoised_image_BI.png", bilat.astype('uint8')) # Saving the
    image


filtered_image_OpenCV = cv2.bilateralFilter(bgr1, 5, 30, 10) # Bilaterally
    filtered image by OpenCV

cv2.imwrite("Denoised_image_OpenCV.png",filtered_image_OpenCV)

plt.imshow(cv2.cvtColor(bgr1.astype('uint8'), cv2.COLOR_BGR2RGB))
plt.show()
plt.imshow(cv2.cvtColor(blur.astype('uint8'), cv2.COLOR_BGR2RGB))
plt.show()
plt.imshow(cv2.cvtColor(filtered_image_OpenCV.astype('uint8'), cv2.
    COLOR_BGR2RGB))
plt.show()
plt.imshow(cv2.cvtColor(bilat.astype('uint8'), cv2.COLOR_BGR2RGB))
plt.show()
```