Computer Vision: Image Demosaicing

Ian Sinclair
ENCE 3620-1

March 30, 2022
Dr. Mohammad H. Mahoor

Abstract

Image demosaicing is the process of reconstructing color information from incomplete samples. In particular, CCD digital cameras have typically captured images by exposing each pixel to only one of three colors (red, blue, or green). Then, a significant amount of color information is missing from the final image. This raises the need for efficient demosaicing algorithms to fill-in the missing pixel data. Here two algorithms outlined or purposed in [1] are implemented and tested against four experimental images. Specifically comparing, a bilinear interpolation and an improved gradient corrected bilinear interpolation method [1]. The analysis is validated by MSE and PSNR values between corresponding experimental and test images.

1 Introduction

Images captured with some single-CCD digital cameras use a color filter array that exposes individual pixels to a single one of red, blue, or green lightwaves following a Bayer pattern grid. This allows multiple wavelengths of information to be captured by a single chip; however, because each pixel has only one color dimension, information is missing from the final image. Demosaicing algorithms can be used to approximate this information. Algorithms implemented here and in [1] are under the assumption that experimental images are captured using a Bayer grid convention.

1.1 Bayer Pattern Grid

A Bayer pattern is a convention that maps a particular pixel on a grid to one of three colors (RBG). Then the intensity of that pixel is used record its corresponding color value.

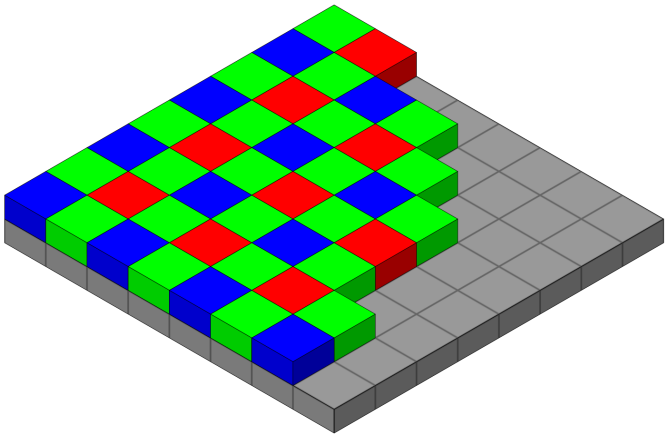

Figure 1: Bayer pattern grid

And so if a 'blue' pixel has intensity of 100, then the blue value of that pixel in the complete image is 100. However, in the original image the red and green values are missing from that pixel.

1.2 Complete Image and Indexing

An image resulting from the Bayer grid pattern will have a single intensity value for each pixel. However, to create an accurately colored image each pixel is desired to store a intensity value for each RGB color. As a result, if coordinate (i, j) for $i, j \in \mathbf{Z}^+$ is on the pixel grid for an image, then $(i, j)[r][b][g]$ corresponds to intensity information for each color. And so here, 'complete image' is used to reference an image where all color intensity values are recorded. (No missing values). Whereas a Bayer image is used to describe an image where each pixel coordinate maps to a single intensity value. $(i, j) \rightarrow k$.

2 Procedure

Within the experiment, two algorithms described in [1] are implemented and tested against four Bayer images. The first algorithm uses Bilinear interpolation to approximate the values of missing color information. The second uses a similar approach with a greater appeal to edge detection via corrected gradient filtering.

2.1 Uploading Bayer pattern images

Four Bayer pattern images are considered,

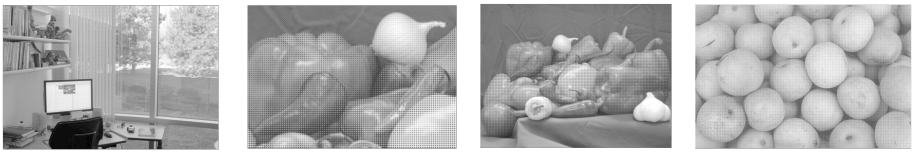

Figure 2: Experimental Bayer Images

Are interpreted as grey-scale intensity images:

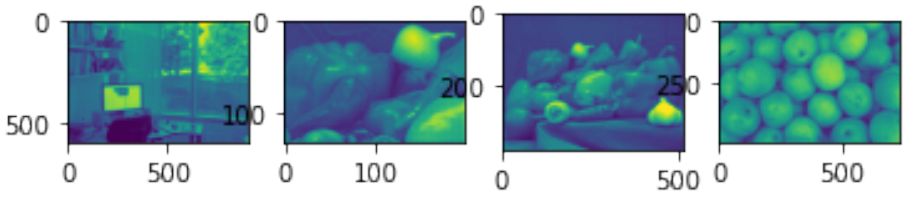

Figure 3: Bayer image interpreted by Matplotlib from single intensity value.

And compared to a corresponding set of test images.

Figure 4: Final validation test image set.

2.2 Accessing Bayer Grid Color Pixels

Critically, because the Bayer images have a single intensity and do not naturally encode which color each pixel represents, it is necessary to use standard convention to access the Bayer grid. This is implemented by symmetry, and used by both algorithms to fill missing color information in each pixel.

```
# All Blue pixels.
for i in range( 0, train.shape[0], 2 ) :
  for j in range( 0, train.shape[1], 2 ) :
      pass # Filled by algorithm specific logic


# All Red pixels.
for i in range( 1, train.shape[0], 2 ) :
  for j in range( 1, train.shape[1], 2 ) :
      pass # Filled by algorithm specific logic

# All Green information
for i in range( 0, train.shape[0], 2 ) :
  for j in range( 1, train.shape[1], 2 ) :
      pass # Filled by algorithm specific logic

for i in range( 1, train.shape[0], 2 ) :
  for j in range( 0, train.shape[1], 2 ) :
      pass # Filled by algorithm specific logic
```

2.3 Method 1: Bilinear Interpolation Demosaicing

The first method in [1] uses symmetric Bilinear interpolation to estimate the missing values from each Bayer image in correspondence to the Bayer grid. Here, by targeting a specific missing color from each pixel, the intensity of that color is approximated by averaging the intensity of adjacent pixels whose color corresponds to that of the target color.

$$\hat{g}(i,j) = \frac{1}{4} \sum_{(m,n)\in\{(-1,0),(1,0),(0,1),(0,-1)\}} g(i+m, j+n) \qquad [1] \qquad (1)$$

Which results in the following filters based on proximity of center pixels to adjacent target colors.

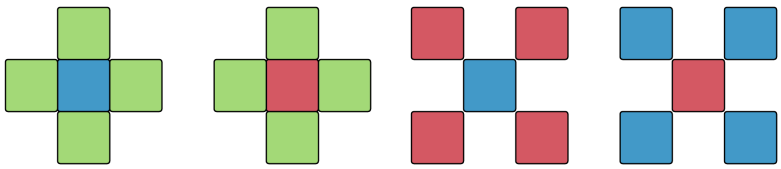

Figure 5: Filters for Red and Blue pixels targeting each other color layer.

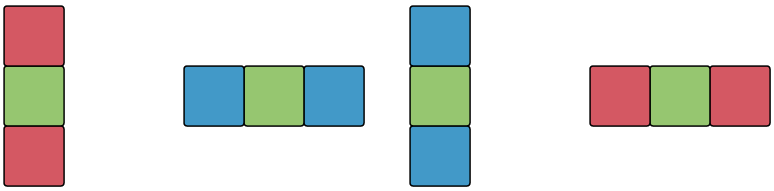

Figure 6: Green pixel targeting Red or Blue depending on their position being vertical or horizontal relative to green.

Note, for each filter, the center pixel inherits the color intensity value corresponding to the color of the adjacent target pixels. Filters are applied to every point on the image plane. Additionally, at image boundaries, filter components that are indexed to points that are not in the image are neglected and the total weight of the filter reconfigured.
Within the implementation All filters are generated from a single function.

```
def Bilinear_Filter( i : int, j : int, Color : str, target : str, train : list ) :
  valid = ['blue', 'red', 'green', 'vertical', 'horizontal']
  if Color not in valid or target not in valid :
    raise ValueError("results: Color/target must be one of %r." % valid)

  den = 0;
  avg = 0;
  if ( Color == 'blue' or Color == 'red' ) and target == 'green' :
    if i + 1 < train.shape[0] :
      den += 1
      avg += train[i+1][j]

    if i - 1 >= 0 :
      den += 1
      avg += train[i-1][j]

    if j + 1 < train.shape[1] :
      den += 1
      avg += train[i][j+1]

    if j - 1 >= 0 :
      den += 1
      avg += train[i][j-1]

    return avg / den

  if ( ( Color == 'blue' or Color == 'red' ) and target != 'green' ) :
    if i + 1 < train.shape[0] :
      if j + 1 < train.shape[1] :
        den += 1
        avg += train[i+1][j+1]

      if j - 1 >= 0 :
        den += 1
        avg += train[i+1][j-1]

    if i - 1 >= 0 :
      if j + 1 < train.shape[1] :
        den += 1
        avg += train[i-1][j+1]

      if j - 1 >= 0 :
        den += 1
        avg += train[i-1][j-1]
    return avg / den


  if Color == 'green' and target == 'vertical' :
    if i + 1 < train.shape[0] :
      den += 1
      avg += train[i+1][j]

    if i - 1 >= 0 :
      den += 1
      avg += train[i-1][j]

    return avg / den

  if Color == 'green' and target == 'horizontal' :
    if j + 1 < train.shape[1] :
      den += 1
      avg += train[i][j+1]

    if j - 1 >= 0 :
      den += 1
      avg += train[i][j-1]

    return avg / den
```

Which results in the following image comparison.

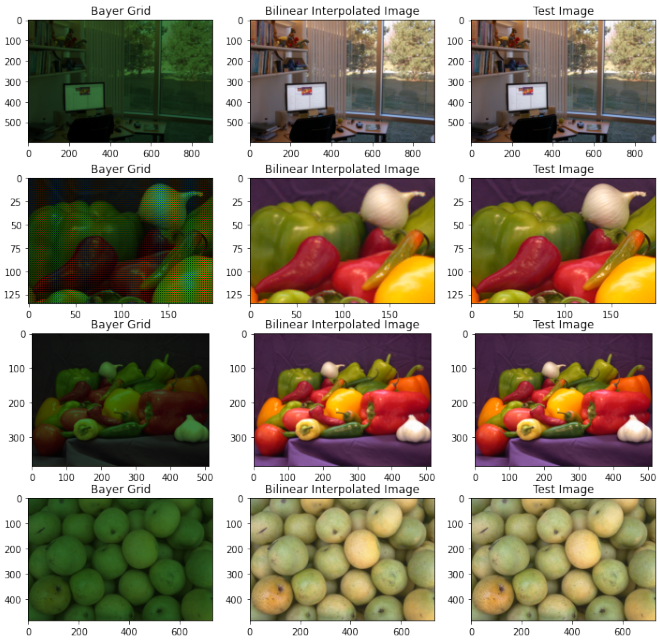

Figure 7: Image comparison of Bilinear interpolation method to raw Bayer and test images.

Importantly, it may be difficult to tell a visual difference between the Bilinearly interpolated images and the test set; however, it still may be possible to improve on the algorithm. [1] Notes that this method does not take advantage of color/intensity correlations between adjacent pixels, and therefore, may be neglecting physical constraints that could improve image quality. To that extend [1] suggests a new method that employed gradient based techniques to detect natural edges in the image.

2.4 Method 2: Gradient Corrected Bilinear Interpolation Demosaicing

This method purposed by [1] takes advantage of the assumption that pixels bound by a small neighborhood should have similar intensity/color information. And if they don't, it is likely that the neighborhood contains an edge in the image. To that extent, [1] purposes an algorithm to weight the intensity of adjacent pixels in a 5×5 grid differently based on the color gradient (distribution) across the grid.
This differential weighting is in general governed by the three following cases.

$$\begin{cases} \hat{g}(i,j) = \hat{g}_b + \alpha\Delta_R(i,j) \\ \hat{r}_1(i,j) = \hat{r}_B + \beta\Delta_G(i,j) \\ \hat{r}_2(i,j) = \hat{r}_B + \gamma\Delta_B(i,j) \\ \Delta_i(i,j) = r(i,j) + \frac{1}{4}\sum_{(m,n)\in\{(2,0),(-2,0),(0,2),(0,-2)\}} r(i+m, r+n) \end{cases} \tag{2}$$

After running a least RMSE analysis on coefficients α, β, γ over a large data-set, [1] determined the optimal scales,

$$\alpha = 1/2, \qquad \beta = 5/8, \qquad \gamma = 3/4 \qquad [1] \tag{3}$$

Which results in the following filters and their implementation.

```
def G_at_RB ( i , j , train ) :
    filter = [ ( 2,0 ) , ( -2,0 ) , ( 0,2 ) , ( 0,-2 ) ]
    interpolate = [ ( 1,0 ) , ( -1,0 ) , ( 0,1 ) , ( 0,-1 ) ]

    center_weight = 4
    adj_weight = 2
    ext_weight = -1
```

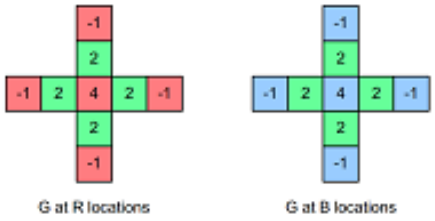

Figure 8: Blue and Red pixel filters targeting Green.

```
8
9     num_active = center_weight
10    sum = center_weight * train[i][j]
11
12    for p in filter :
13      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
14        num_active += ext_weight
15        sum += ext_weight * train[ i + p[0] ][ j + p[1] ]
16
17    for p in interpolate :
18      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
19        num_active += adj_weight
20        sum += adj_weight * train[ i + p[0] ][ j + p[1] ]
21
22    return sum / num_active
```

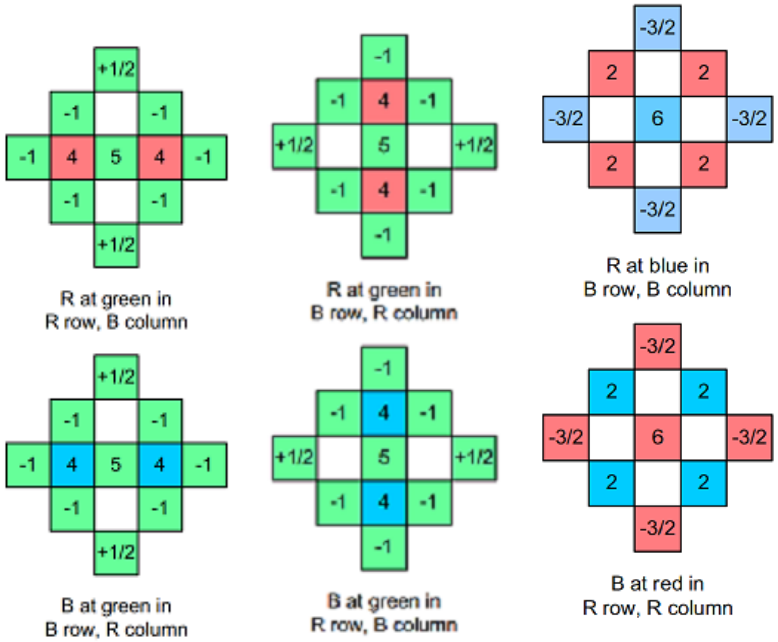

Figure 9: Gradient Corrected Bilinear Interpolated Filters

All filter implementations are consistent with figure 8.
For each filter, the weight on each tile is multiplied by the intensity of that corresponding pixel. Then the sum of all weight intensity pairs is divided by the sum of all weights across the filter to provide the center pixel color estimate.
Examining the 'G at R locations' filter, notice that the filter targets green colors, however, red pixels external to the adjacent green pixels have an weight on the total. Or rather, if there is a significant intensity change across red external filter elements the center pixel will be heavily weighted. This represents gradient correction behavior, and edge detection. Such that, without an edge it is as-

sumed that pixels in a small neighborhood have similar intensity, and so the filter behaves like that of bilinear interpolation. However, at an event of high intensity change like at an edge, the filter adjusts the weights based on the boundary, and so tracks the gradient.
Each filter is then convolved through out the image for corresponding center color and target pairs. Similar to method 1, filter elements that are out of the boundary of the image are neglected and the filter sum adjusted. Ultimately resulting in the following image comparison.

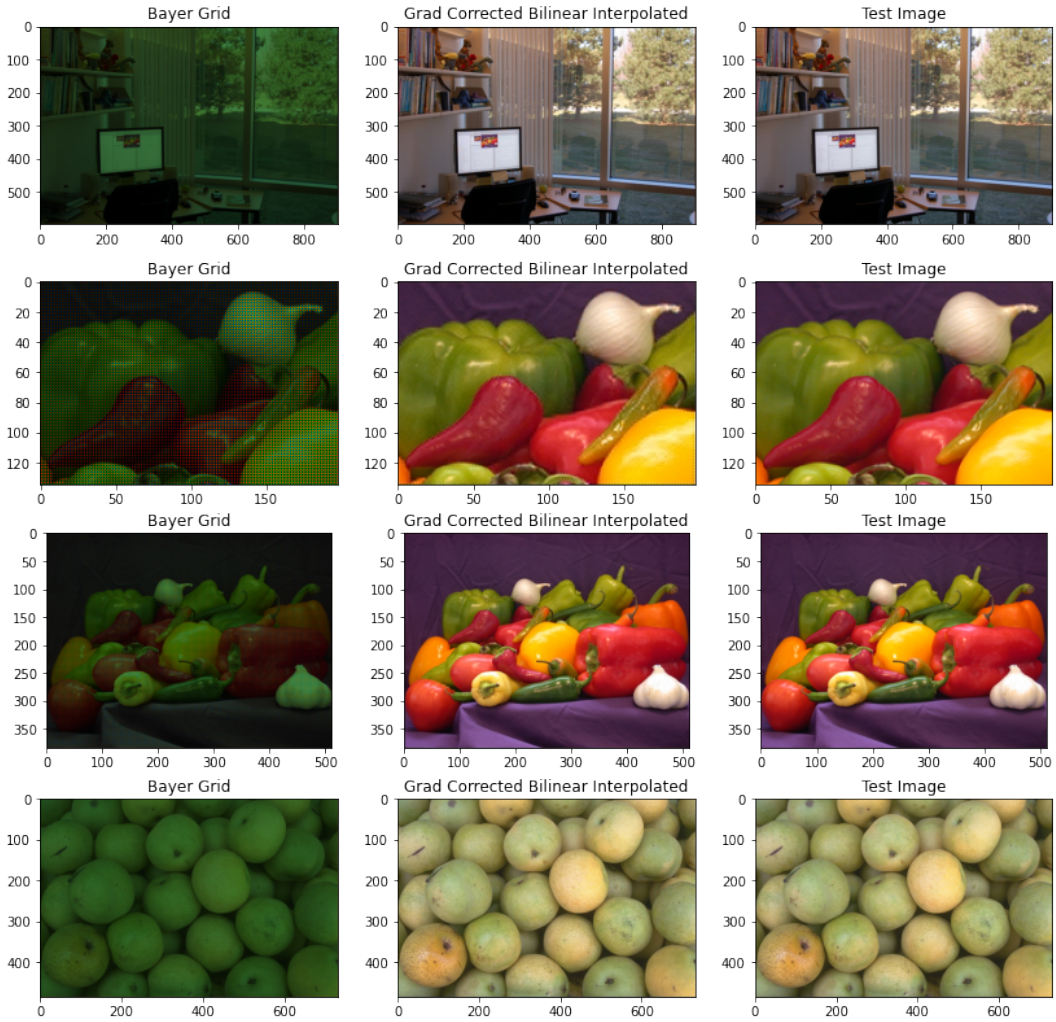

Figure 10: Image comparison of Gradient Corrected Bilinear Interpolation method to raw Bayer and test images.

3 Results

Image validation is handled by both mean square error (MSE) and peak signal to noise ratio (PSNR)

$$MSE = \frac{\sum_{N,M}[I_1(N,M) - I_2(N,M)]^2}{N\dot{M}} \tag{4}$$

$$PSNR = 10\log_{10}(\frac{R^2}{MSE}) \tag{5}$$

Where I_1 is the original image and the I_2 is the demosaiced image. R is the maximum fluctuation in the input image data type.
And so consider the following implementation.

```
1 def MSE_RGB ( train ,test ) :
```

```
    num_elements = 0
    sum_blue = 0
    sum_red = 0
    sum_green = 0

    for i in range(0 , test.shape[0]) :
      for j in range(0 , test.shape[1]) :
        num_elements += 1
        sum_red += ( train[i][j][0] - test[i][j][0] )**2
        sum_green += ( train[i][j][1] - test[i][j][1] )**2
        sum_blue += ( train[i][j][2] - test[i][j][2] )**2

    return sum_red / num_elements, sum_green / num_elements , sum_blue / num_elements
```

```
def PSNR ( MSE , test ) :
  R = 0
  sum = []
  for j in test :
    for k in j :
      sum.append( mean(k) )
  if mean( sum ) > 1 :
    R = 255
  else :
    R = 1

  return  10*math.log10(float(R**2 / MSE))
```

Corresponds to the following RGB MSE and PSNR for each image.

MSE	Bilinear Interpolation			Gradient Corrected Bilinear Interpolation		
	Red	Green	Blue	Red	Green	Blue
Office	0.000769	0.000546	0.001196	0.000494	0.000413	0.000673
Onion	0.000425	0.000198	0.000461	0.000689	0.000196	0.000769
Peppers	0.000243	0.000100	0.000258	0.000250	0.00009	0.000277
Pears	0.000206	0.00009	0.000227	0.00007	0.00003	0.00008
Mean	0.0004113	0.0002353	0.000535	0.0003778	0.0001835	0.000449

Table 1: Table: MSE by color for Bilinear Interpolation and Gradient Corrected Bilinear Interpolation

PSNR	Bilinear Interpolation			Gradient Corrected Bilinear Interpolation		
	Red	Green	Blue	Red	Green	Blue
Office	31.1403	32.6241	29.2200	33.0613	33.8315	31.7141
Onion	33.70975	37.02679	33.3571	31.6126	37.05603	31.1390
Peppers	36.1330	39.9955	35.8835	36.00652	40.3432	35.568704
Pears	36.8423	40.1555	36.43931	41.15710388	45.0625	41.0048
Mean	34.45635	37.45052	33.7250	35.45938	39.07335	34.85667

Table 2: Table: MSE by color for Bilinear Interpolation and Gradient Corrected Bilinear Interpolation

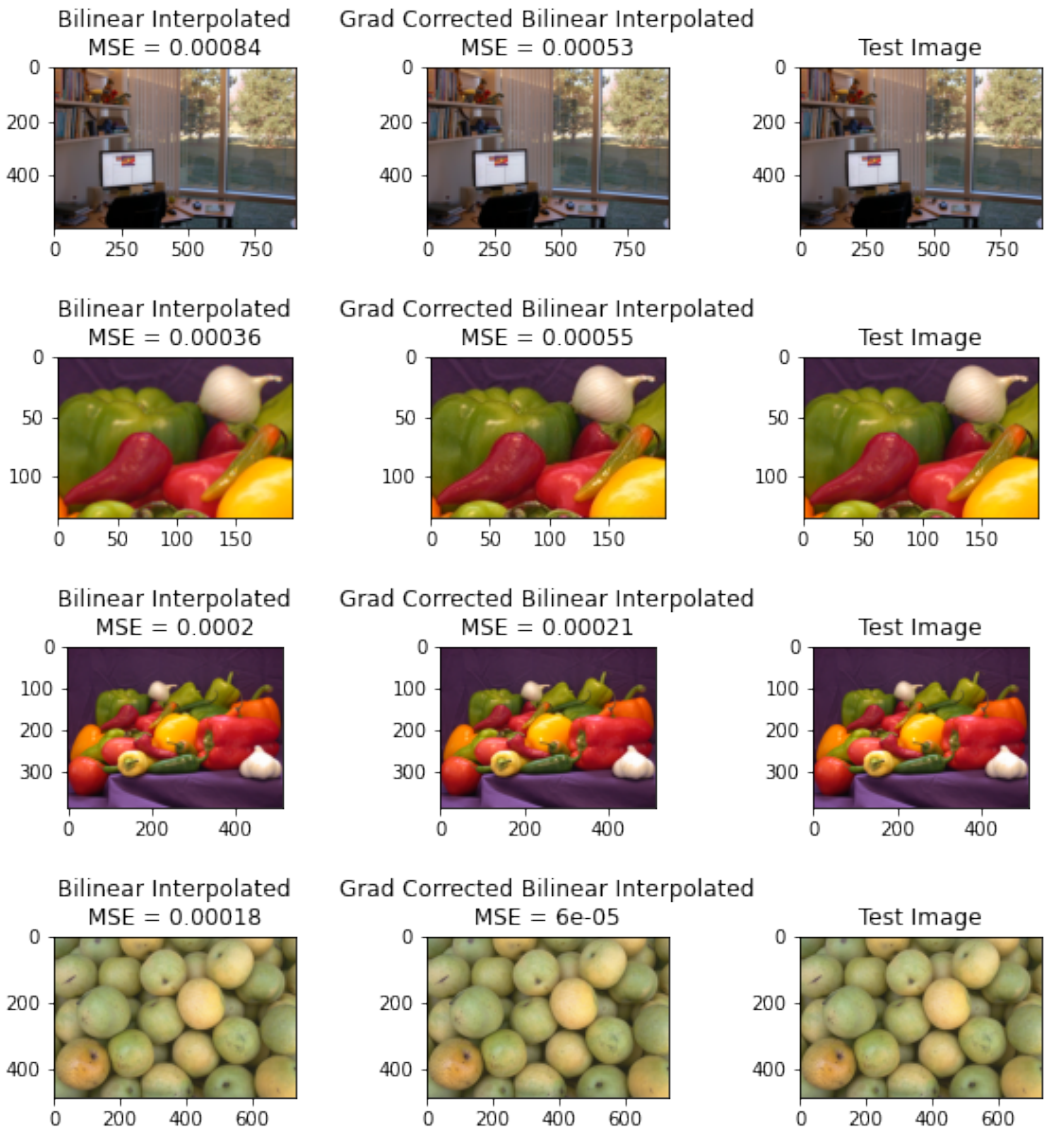

Figure 11: PSNR comparison by image

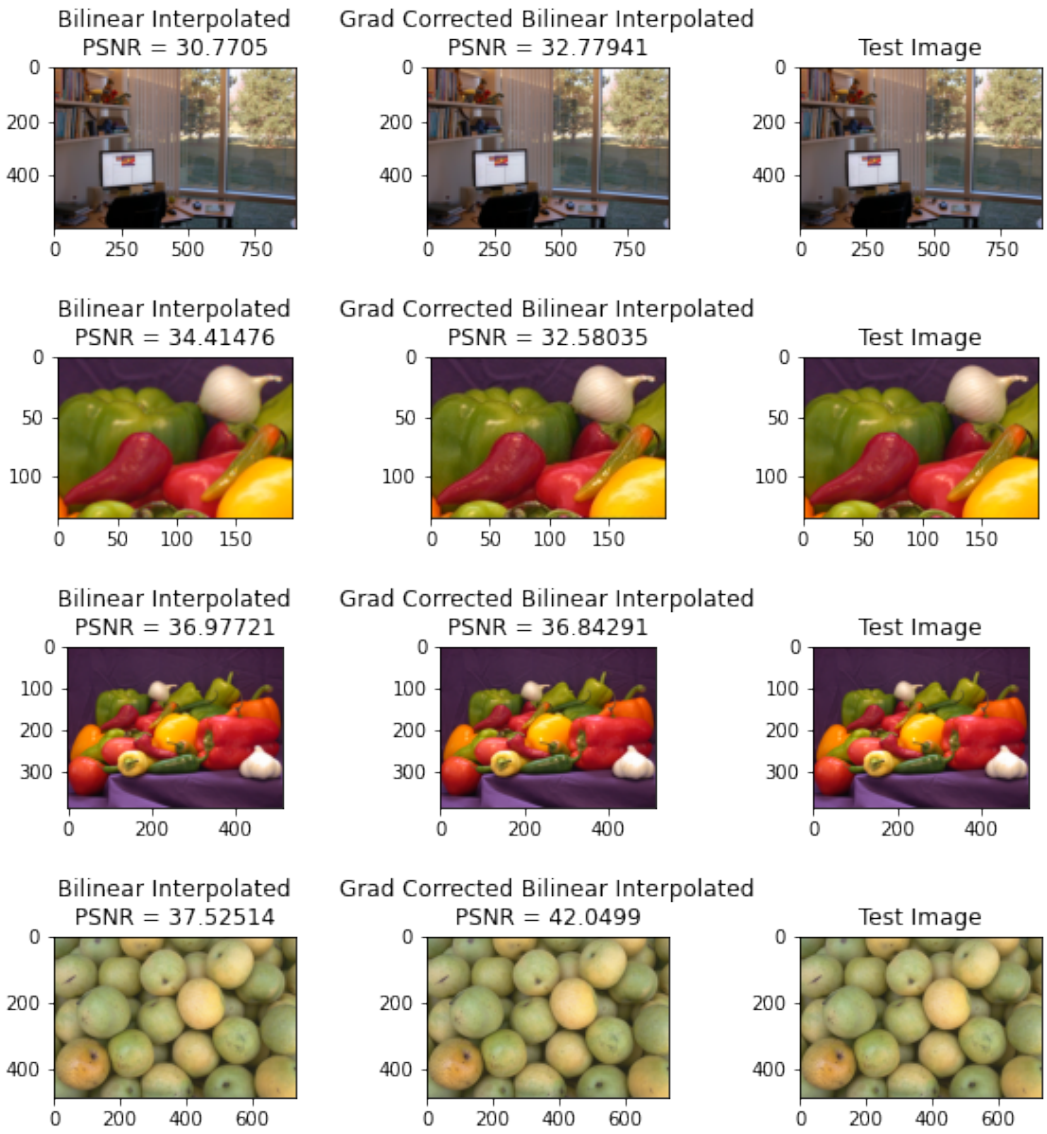

Figure 12: Avg. PSNR comparison by image

4 Conclusion

Here two algorithms discussed in [1] are implemented to demosaice Bayer images. The validity of each algorithm is described by MSE and PSNR for each pixel color category and averaged across all color categories. On average, the gradient corrected bilinear interpolated demosaicing algorithm outperformed the traditional bilinear interpolation demosaicing algorithm. Demonstrating lower average MSE by pixel and across the whole image. Interestingly, the average PSNR varies across the type of image. For images with a low number of edges or low intensity fluctuation in large neighborhoods (Like the office or onion photo), the bilinear interpolation had increased performance. However, with lots of edges, significant changes in intensity in a large neighborhood, the gradient corrected bilinear interpolated demosaicing method out preformed the other algorithm by up to $5.5dB$. Which is consistent with the results gathered from [1]. Both algorithms are comparable; however, the gradient corrected bilinear interpolated demosaicing method seems to on average out preform traditional bilinear interpolation methods.

5 References

[1] Malvar HS, He LW, Cutler R. High-quality linear interpolation for demosaicing of Bayer-patterned color images. InAcoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on 2004 May 17 (Vol. 3, pp. iii-485). IEEE.

6 Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mp
import PIL as pl
import cv2
from statistics import mean
import math

from google.colab import drive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
from google.colab import drive
drive.mount('/content/drive')


auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)


experimental_images = []

image_path = 'drive/MyDrive/officeBayer (1).png'
experimental_images.append( mp.imread(image_path) )
train = mp.imread( image_path )

image_path = 'drive/MyDrive/onionBayer.png'
experimental_images.append( mp.imread(image_path) )

image_path = 'drive/MyDrive/peppersBayer.png'
experimental_images.append( mp.imread(image_path) )

image_path = 'drive/MyDrive/pearsBayer.png'
experimental_images.append( mp.imread(image_path) )


test_images = []

image_path = 'drive/MyDrive/office_4 (1).jpg'
test_images.append( mp.imread( image_path ) )
test = mp.imread( image_path )

image_path = 'drive/MyDrive/onion.png'
test_images.append( mp.imread( image_path ) )

image_path = 'drive/MyDrive/peppers.png'
test_images.append( mp.imread( image_path ) )

image_path = 'drive/MyDrive/pears.png'
test_images.append( mp.imread( image_path ) )


def Bilinear_Filter( i : int, j : int, Color : str, target : str, train : list ) :
  valid = ['blue', 'red', 'green', 'vertical', 'horizontal']
  if Color not in valid or target not in valid :
    raise ValueError("results: Color/target must be one of %r." % valid)

  den = 0;
  avg = 0;
  if ( Color == 'blue' or Color == 'red' ) and target == 'green' :
    if i + 1 < train.shape[0] :
      den += 1
      avg += train[i+1][j]

    if i - 1 >= 0 :
      den += 1
      avg += train[i-1][j]
```

```
71
72      if j + 1 < train.shape[1] :
73        den += 1
74        avg += train[i][j+1]
75
76      if j - 1 >= 0 :
77        den += 1
78        avg += train[i][j-1]
79
80      return avg / den
81
82    if ( ( Color == 'blue' or Color == 'red' ) and target != 'green' ) :
83      if i + 1 < train.shape[0] :
84        if j + 1 < train.shape[1] :
85          den += 1
86          avg += train[i+1][j+1]
87
88        if j - 1 >= 0 :
89          den += 1
90          avg += train[i+1][j-1]
91
92      if i - 1 >= 0 :
93        if j + 1 < train.shape[1] :
94          den += 1
95          avg += train[i-1][j+1]
96
97        if j - 1 >= 0 :
98          den += 1
99          avg += train[i-1][j-1]
100     return avg / den
101
102
103   if Color == 'green' and target == 'vertical' :
104     if i + 1 < train.shape[0] :
105       den += 1
106       avg += train[i+1][j]
107
108     if i - 1 >= 0 :
109       den += 1
110       avg += train[i-1][j]
111
112     return avg / den
113
114   if Color == 'green' and target == 'horizontal' :
115     if j + 1 < train.shape[1] :
116       den += 1
117       avg += train[i][j+1]
118
119     if j - 1 >= 0 :
120       den += 1
121       avg += train[i][j-1]
122
123     return avg / den
124
125
126
127 def Bilinear_Interpolation( Bayer_image , train ) :
128
129   Bilinear_Interpolation = []
130   Bilinear_Interpolation = Bayer_image.copy()
131   #  All Blue pixels.
132   for i in range( 0, train.shape[0], 2 ) :
133     for j in range( 0, train.shape[1], 2 ) :
134       Bilinear_Interpolation[i][j][1] = Bilinear_Filter( i , j , 'blue', 'green', train )
135       Bilinear_Interpolation[i][j][0] = Bilinear_Filter( i , j , 'blue', 'red', train )
136
137
138   #  All Red pixels.
139   for i in range( 1, train.shape[0], 2 ) :
140     for j in range( 1, train.shape[1], 2 ) :
141       Bilinear_Interpolation[i][j][1] = Bilinear_Filter( i , j , 'red', 'green', train )
142       Bilinear_Interpolation[i][j][2] = Bilinear_Filter( i , j , 'red', 'blue', train )
143
144   #  All Green information
145   for i in range( 0, train.shape[0], 2 ) :
146     for j in range( 1, train.shape[1], 2 ) :
147       Bilinear_Interpolation[i][j][0] = Bilinear_Filter( i , j , 'green' , 'vertical' , train )
148       Bilinear_Interpolation[i][j][2] = Bilinear_Filter( i , j , 'green' , 'horizontal' , train )
149
150   for i in range( 1, train.shape[0], 2 ) :
151     for j in range( 0, train.shape[1], 2 ) :
152       Bilinear_Interpolation[i][j][0] = Bilinear_Filter( i , j , 'green' , 'horizontal' , train )
153       Bilinear_Interpolation[i][j][2] = Bilinear_Filter( i , j , 'green' , 'vertical' , train )
154
```

```
    return Bilinear_Interpolation




Bayer_Grid_Images = []
for i in range( 0 , len( experimental_images ) ) :
  Bayer_Grid_Images.append( init_from_bayer_grid( experimental_images[i], test_images[i] ) )


Bilinear_Interpolated_Images = []
for i in range( 0 , len( Bayer_Grid_Images ) ) :
  Bilinear_Interpolated_Images.append( Bilinear_Interpolation( Bayer_Grid_Images[i] , experimental_images[i] ) )




%matplotlib inline
fig, axs = plt.subplots( len( test_images ) , 3 , figsize=(12, 12) , )
for i in range( 0 , len( test_images )) :
  axs[i , 0].imshow(Bayer_Grid_Images[i])
  axs[i, 0].set_title('Bayer Grid')
  axs[i , 1].imshow( Bilinear_Interpolated_Images[i] )
  axs[i, 1].set_title('Bilinear Interpolated Image')
  axs[i , 2].imshow( test_images[i] )
  axs[i, 2].set_title('Test Image')




def G_at_RB ( i , j , train ) :
  filter = [ ( 2,0 ) , ( -2,0 ) , ( 0,2 ) , ( 0,-2 ) ]
  interpolate = [ ( 1,0 ) , ( -1,0 ) , ( 0,1 ) , ( 0,-1 ) ]

  center_weight = 4
  adj_weight = 2
  ext_weight = -1

  num_active = center_weight
  sum = center_weight * train[i][j]

  for p in filter :
    if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
      num_active += ext_weight
      sum += ext_weight * train[ i + p[0] ][ j + p[1] ]

  for p in interpolate :
    if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
      num_active += adj_weight
      sum += adj_weight * train[ i + p[0] ][ j + p[1] ]

  return sum / num_active




def BR_at_G_Horizontal( i , j , train ) :
  filter = [ ( 0,2 ) , ( -1,1 ) , ( -1,-1 ) , ( 0,-2 ) , ( 1,-1 ) , ( 1,1 ) ]
  filter2 = [ ( 2,0 ) , ( -2,0 ) ]

  interpolate = [ ( 0,1 ) , ( 0,-1 ) ]

  center_weight = 5
  adj_weight = 4
  ext_weight = -1
  ext_weight2 = 1/2


  num_active = center_weight
  sum = center_weight * train[i][j]

  for p in filter :
    if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
      num_active += ext_weight
      sum += ext_weight * train[ i + p[0] ][ j + p[1] ]

  for p in filter2 :
    if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
      num_active += ext_weight2
      sum += ext_weight2 * train[ i + p[0] ][ j + p[1] ]

  for p in interpolate :
    if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
```

```
        num_active += adj_weight
        sum += adj_weight * train[ i + p[0] ][ j + p[1] ]

    return sum / num_active




def BR_at_G_Vertical( i , j , train ) :
    filter = [ ( 2,0 ) , ( -1,1 ) , ( 1,1 ) , ( -2,0 ) , ( 1,-1 ) , ( -1,-1 ) ]
    filter2 = [ ( 0,2 ) , ( 0,-2 ) ]

    interpolate = [ ( 1,0 ) , ( -1,0 ) ]

    center_weight = 5
    adj_weight = 4
    ext_weight = -1
    ext_weight2 = 1/2


    num_active = center_weight
    sum = center_weight * train[i][j]

    for p in filter :
      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
        num_active += ext_weight
        sum += ext_weight * train[ i + p[0] ][ j + p[1] ]

    for p in filter2 :
      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
        num_active += ext_weight2
        sum += ext_weight2 * train[ i + p[0] ][ j + p[1] ]

    for p in interpolate :
      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
        num_active += adj_weight
        sum += adj_weight * train[ i + p[0] ][ j + p[1] ]

    return sum / num_active



def BR_at_RB ( i , j , train ) :
    filter = [ ( 2,0 ) , ( 0,2 ) , ( -2,0 ) , ( 0,-2 ) ]

    interpolate = [ ( 1,1 ) , ( -1,1 ) , ( -1,-1 ) , ( 1,-1 ) ]

    center_weight = 6
    adj_weight = 2
    ext_weight = -3/2

    num_active = center_weight
    sum = center_weight * train[i][j]
    for p in filter :
      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
        num_active += ext_weight
        sum += ext_weight * train[ i + p[0] ][ j + p[1] ]

    for p in interpolate :
      if ( 0 <= i + p[0] < train.shape[0] ) and ( 0 <= j + p[1] < train.shape[1] ) :
        num_active += adj_weight
        sum += adj_weight * train[ i + p[0] ][ j + p[1] ]

    return sum / num_active



def Gradient_corrected_Bilinear_Interpolation ( Bayer_image , train ) :
    Grad_Interpolation = []
    Grad_Interpolation = Bayer_image.copy()

    '''
      0 --> Red
      1 --> Green
      2 --> Blue
    '''

    #  All Blue pixels.
    for i in range( 0, train.shape[0], 2 ) :
      for j in range( 0, train.shape[1], 2 ) :
        Grad_Interpolation[i][j][1] = G_at_RB( i , j , train )
        Grad_Interpolation[i][j][0] = BR_at_RB( i , j , train )


```

```
    # All Red pixels.
    for i in range( 1, train.shape[0], 2 ) :
      for j in range( 1, train.shape[1], 2 ) :
        Grad_Interpolation[i][j][1] = G_at_RB( i , j , train )
        Grad_Interpolation[i][j][2] = BR_at_RB( i , j , train )

    # All Green information
    for i in range( 0, train.shape[0], 2 ) :
      for j in range( 1, train.shape[1], 2 ) :
        Grad_Interpolation[i][j][0] = BR_at_G_Vertical( i , j , train )
        Grad_Interpolation[i][j][2] = BR_at_G_Horizontal( i , j , train )

    for i in range( 1, train.shape[0], 2 ) :
      for j in range( 0, train.shape[1], 2 ) :
        Grad_Interpolation[i][j][0] = BR_at_G_Horizontal( i , j , train )
        Grad_Interpolation[i][j][2] = BR_at_G_Vertical( i , j , train )

    return Grad_Interpolation



Bayer_Grid_Images = []
for i in range( 0 , len( experimental_images ) ) :
  Bayer_Grid_Images.append( init_from_bayer_grid( experimental_images[i], test_images[i] ) )



Grad_Corrected_Bilinear_Interpolated_Images = []
for i in range( 0 , len( Bayer_Grid_Images ) ) :
  Grad_Corrected_Bilinear_Interpolated_Images.append( Gradient_corrected_Bilinear_Interpolation( Bayer_Grid_Images[i] ,
       experimental_images[i] ) )



%matplotlib inline
fig, axs = plt.subplots( len( test_images ) , 3 , figsize=(14, 14) , )
for i in range( 0 , len( test_images )) :
  axs[i , 0].imshow(Bayer_Grid_Images[i])
  axs[i, 0].set_title('Bayer Grid')
  axs[i , 1].imshow( Grad_Corrected_Bilinear_Interpolated_Images[i] )
  axs[i, 1].set_title('Grad Corrected Bilinear Interpolated')
  axs[i , 2].imshow( test_images[i] )
  axs[i, 2].set_title('Test Image')



def con_255_01 ( test ) :
  validate = np.zeros(test.shape)
  for i in range(0 , test.shape[0]) :
    for j in range(0 , test.shape[1]) :
      for k in range(0 , test.shape[2]) :
        validate[i][j][k] = test[i][j][k]/255

  return validate



def MSE( train , test ) :
  num_elements = 0
  sum = 0
  for i in range(0 , test.shape[0]) :
    for j in range(0 , test.shape[1]) :
      for k in range(0 , test.shape[2]) :
        num_elements += 1
        sum += ( train[i][j][k] - test[i][j][k] ) ** 2
  return sum / num_elements



def MSE_RGB ( train ,test ) :
  num_elements = 0
  sum_blue = 0
  sum_red = 0
  sum_green = 0

  for i in range(0 , test.shape[0]) :
    for j in range(0 , test.shape[1]) :
      num_elements += 1
      sum_red += ( train[i][j][0] - test[i][j][0] )**2
```

```
        sum_green += ( train[i][j][1] - test[i][j][1] )**2
        sum_blue += ( train[i][j][2] - test[i][j][2] )**2

    return sum_red / num_elements, sum_green / num_elements , sum_blue / num_elements



def PSNR ( MSE , test ) :
    R = 0
    sum = []
    for j in test :
        for k in j :
            sum.append( mean(k) )
    if mean( sum ) > 1 :
        R = 255
    else :
        R = 1

    return  10*math.log10(float(R**2 / MSE))


validate = []
for i in range( 0 , len( test_images ) ) :
    sum = []
    for j in test_images[i] :
        for k in j :
            sum.append( mean(k) )
    if mean( sum ) > 1 :
        validate.append( con_255_01( test_images[i] ) )
    else :
        validate.append( test_images[i] )



Bilinear_Interpolation_MSE = []
Bilinear_Interpolation_PSNR = []
Bilinear_Interpolation_MSE_RGB = []
for i in range ( 0 , len( Bilinear_Interpolated_Images )) :
    Bilinear_Interpolation_MSE.append( MSE( Bilinear_Interpolated_Images[i] , validate[i] ) )
    Bilinear_Interpolation_MSE_RGB.append( MSE_RGB( Bilinear_Interpolated_Images[i] , validate[i] ) )

for i in range ( 0 , len ( Bilinear_Interpolation_MSE )) :
    Bilinear_Interpolation_PSNR.append( PSNR( Bilinear_Interpolation_MSE[i] , validate[i] ) )


Grad_Corrected_Bilinear_Interpolation_MSE = []
Grad_Corrected_Bilinear_Interpolation_PSNR = []
Grad_Corrected_Bilinear_Interpolation_MSE_RGB = []
for i in range ( 0 , len( Grad_Corrected_Bilinear_Interpolated_Images )) :
    Grad_Corrected_Bilinear_Interpolation_MSE.append( MSE( Grad_Corrected_Bilinear_Interpolated_Images[i] , validate[i] ) )
    Grad_Corrected_Bilinear_Interpolation_PSNR.append( PSNR(Grad_Corrected_Bilinear_Interpolation_MSE[-1] , validate[i] ))
    Grad_Corrected_Bilinear_Interpolation_MSE_RGB.append( MSE_RGB( Grad_Corrected_Bilinear_Interpolated_Images[i] , validate[i] )  )


 %matplotlib inline
fig, axs = plt.subplots( len( test_images ) , 3 , figsize=( 10, 10) )
fig.subplots_adjust(hspace = 0.8)
for i in range( 0 , len( test_images )) :
    axs[i , 0].imshow(Bilinear_Interpolated_Images[i])
    axs[i, 0].set_title('Bilinear Interpolated' + '\nMSE = ' + str(round(Bilinear_Interpolation_MSE[i], 5)))
    axs[i , 1].imshow( Grad_Corrected_Bilinear_Interpolated_Images[i] )
    axs[i, 1].set_title('Grad Corrected Bilinear Interpolated' + '\nMSE = ' + str(round(Grad_Corrected_Bilinear_Interpolation_MSE[i],
        5)))
    axs[i , 2].imshow( test_images[i] )
    axs[i, 2].set_title('Test Image')



%matplotlib inline
fig, axs = plt.subplots( len( test_images ) , 3 , figsize=( 10, 10) )
fig.subplots_adjust(hspace = 0.8)
for i in range( 0 , len( test_images )) :
    axs[i , 0].imshow(Bilinear_Interpolated_Images[i])
    axs[i, 0].set_title('Bilinear Interpolated' + '\nPSNR = ' + str(round(Bilinear_Interpolation_PSNR[i], 5)))
    axs[i , 1].imshow( Grad_Corrected_Bilinear_Interpolated_Images[i] )
    axs[i, 1].set_title('Grad Corrected Bilinear Interpolated' + '\nPSNR = ' + str(round(Grad_Corrected_Bilinear_Interpolation_PSNR[i
        ], 5)))
    axs[i , 2].imshow( test_images[i] )
    axs[i, 2].set_title('Test Image')
```