Luisa Chiu

10/7/2023

CPE/EE 428

Homework 1: RAW Processing

First, I looked into the Bayer pattern and figured out the 4 case statements for blue pixels, red pixels, green pixels with red adjacent, and green pixels with blue adjacent. I created a test array to manipulate the values and see if my code works as I hard coded each function.

In the demosaic() function, I normalized the image using the .astype() function and zero padding to account for the edges using the np.pad() function. Next, I created three empty arrays (for red, blue, and green channels) that were the same size as in input array/normalized array using np.empty(). Using nested for loops, I was able to iterate through each row and column of the image, find the respective averages for each color and pixel using np.mean() on the padded array, and used indexing to put them in the three empty color arrays. Next, I stacked the three RGB arrays and used the .astype() function to convert it into an array of 32-bit floating point values and returned the demosaiced image. The testing I conducted by hand can be seen in Attachment 1, and the testing I conducted through code can be seen in Attachment 2.\

In the white\_balance() function, I grabbed the first, second, and third values of each row and column of the input image using indexing to find the red, green, and blue channels. Using the np.mean() function, I found the scalar values for each of the 3 channels by dividing the mean values with 0.5. Next, I multiplied each of the colors with their respective scalar values. I tested the code to see if it works by individually finding the mean values of all the new RGB arrays. Lastly, I stacked the new three arrays, converted them back to a float32 data type, and returned the white balanced image.

In the curve\_and\_quantize() function, I put the input image array to the power of the inverse gamma value. Next, I clipped it using np.clip() to limit the values from 0 to 1. Next, I multiplied this clipped image by 255 and used .astsype() to convert it to an array with unsigned 8-bit integers. Finally, I returned the output image.

I used these functions in the process.py file and used imageio.imwrite() to output the processed image as the required file types .png and .jpeg: “Processed\_Image.png” and “Processed\_Image.jpeg”. At the very end, I tested the provided .DNG files to see if my code was correct.

Attachment 1:

A math paper with math equations

Description automatically generated with medium confidence

Attachment 1(cont’d):

A paper with math equations and numbers

Description automatically generated

Attachment 2:

'''

# Demosaic Test Code

#    normalized\_img = raw\_image.astype(np.float32) / (2\*\*16 - 1)

    original = np.array([[100, 20, 30, 40], [80, 10, 70, 20], [10, 20, 30, 20], [200, 70, 80, 90]])

    padded = np.pad(original, pad\_width=1, mode='constant', constant\_values=0)

    print("Padded Array:")

    print(padded)

    row = padded.shape[0] # num of rows

    col = padded.shape[1] # num of col

    Red = []

    Blue = []

    Green = []

    Red\_array= np.empty((original.shape[0], original.shape[1]))

    Blue\_array= np.empty((original.shape[0], original.shape[1]))

    Green\_array= np.empty((original.shape[0], original.shape[1]))

    for i in range(1, row-1, 1):

        for j in range(1, col-1, 1):

            # At red pixel

            if i%2 != 0 and j%2 != 0:

                Red.append(padded[i, j])

                Red\_array[i-1 ,j-1] = padded[i, j]

                Green.append((1/4)\*(padded[i+1, j] + padded[i-1, j] + padded[i, j-1] + padded[i, j+1]))

                Green\_array[i-1 ,j-1] = (1/4)\*(padded[i+1, j] + padded[i-1, j] + padded[i, j-1] + padded[i, j+1])

                Blue.append((1/4)\*(padded[i+1, j+1] + padded[i-1, j-1]+ padded[i-1, j+1] + padded[i+1, j-1]))

                Blue\_array[i-1 ,j-1] = (1/4)\*(padded[i+1, j+1] + padded[i-1, j-1]+ padded[i-1, j+1] + padded[i+1, j-1])

            # At green pixel with red adjacent

            elif i%2 != 0 and j%2 == 0:

                Red.append((1/2)\*(padded[i, j-1] + padded[i, j+1]))

                Red\_array[i-1,j-1] = (1/2)\*(padded[i, j-1] + padded[i, j+1])

                Green.append(padded[i,j])

                Green\_array[i-1 ,j-1] = padded[i,j]

                Blue.append((1/2)\*(padded[i-1, j]+padded[i+1, j]))

                Blue\_array[i-1 ,j-1] = (1/2)\*(padded[i-1, j]+padded[i+1, j])

            # At green pixel with blue adjacent

            elif i%2 == 0 and j%2 != 0:

                Red.append((1/2)\*(padded[i-1, j]+padded[i+1, j]))

#                np.append(Red\_array, (1/2)\*(padded[i-1, j]+padded[i+1, j]))

                Red\_array[i-1,j-1] = (1/2)\*(padded[i-1, j]+padded[i+1, j])

Attachment 2 (cont’d):

                Green.append(padded[i,j])

                Green\_array[i-1 ,j-1] = padded[i,j]

                Blue.append((1/2)\*(padded[i, j-1] + padded[i, j+1]))

                Blue\_array[i-1 ,j-1] = (1/2)\*(padded[i, j-1] + padded[i, j+1])

            # At blue pixel

            elif i%2 == 0 and j%2 == 0:

                Red.append((1/4)\*(padded[i+1, j+1] + padded[i-1, j-1]+ padded[i-1, j+1] + padded[i+1, j-1]))

#                np.append(Red\_array, (1/4)\*(padded[i+1, j+1] + padded[i-1, j-1]+ padded[i-1, j+1] + padded[i+1, j-1]))

                Red\_array[i-1,j-1] = (1/4)\*(padded[i+1, j+1] + padded[i-1, j-1]+ padded[i-1, j+1] + padded[i+1, j-1])

                Green.append((1/4)\*(padded[i+1, j] + padded[i-1, j] + padded[i, j-1] + padded[i, j+1]))

                Green\_array[i-1 ,j-1] = (1/4)\*(padded[i+1, j] + padded[i-1, j] + padded[i, j-1] + padded[i, j+1])

                Blue.append(padded[i, j])

                Blue\_array[i-1 ,j-1] = padded[i, j]

#print(Red)

#print(np.array(Red))

print()

print("Red array:")

print(Red\_array)

print("Blue array:")

print(Blue\_array)

print("Green array:")

print(Green\_array)

c = np.stack((Red\_array, Green\_array, Blue\_array), axis = 2)

print("Stacked array:")

print(c)

print(c.shape)

print(c.dtype)

'''