Mini-project 2

CMPSCI 670, Fall 2019, UMass Amherst
Instructor: Subhransu Maji
TAs: Aruni Roy Chowdhury, Archan Ray

Guidelines

Submission. Submit a *single pdf document* via Gradescope that includes your solutions, figures and code. The latex source file for the homework is provided which you can modify to produce your report. You are welcome to use other typesetting software as long as the final output is a pdf. For readability you may attach the code printouts at the end of the solutions within the same pdf. Note that we will not run your code. Similarly figures should be included in a manner which makes it easy to compare various approaches. Poorly written or formatted reports will make it harder for the TA to evaluate it and may lead to a deduction of credit.

Late policy. You could have 24 hours late submission with a 50% mark down. Late submission beyond 24 hours will not be given *any* credits.

Plagiarism. We might reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers. We expect students to want to learn and not google for answers. See the Universities' guidelines on academic honesty (`https://www.umass.edu/honesty`).

Collaboration. The homework must be done individually, except where otherwise noted in the assignments. 'Individually' means each student must hand in their own answers, and each student must write their own code in the programming part of the assignment. It is acceptable, however, for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

Using other programming languages. We made the starter code available in Python and Matlab. You are free to use other languages such as C, Java, Octave or Julia with the caveat that we won't be able to answer or debug language-questions.

Python requirements. We will be using Python 2.7. The Python starter code requires `scipy`, `numpy` (at least v1.12), and `scikit-image`. If you are not familiar with installing those libraries through some package manager (like `pip`), the easiest way of using them is installing Anaconda.

1 Color image demosaicing [30 points]

Recall that in digital cameras the red, blue, and green sensors are interlaced in the Bayer pattern (Figure 1). The missing values are interpolated to obtain a full color image. In this part you will implement several interpolation algorithms.

Your entry point for this part of the homework is in `evalDemosaicing`. The code loads images from the data directory (in `data/demosaic`), artificially mosaics them (`mosaicImage` file), and provides them as input to the demosaicing algorithm (`demosaicImage` file). The input to the algorithm is a single image im, a NxM array of numbers between 0 and 1. These are measurements in the format shown in Figure 1, i.e., top left `im(1,1)|im[0,0]` is red, `im(1,2)|im[0,1]` is green, `im(2,1)|im[1,0]` is green, `im(2,2)|im[1,1]` is blue, etc. Your goal is to create a single color image C from these measurements.

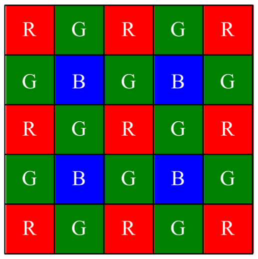

Figure 1: Bayer pattern

By comparing the result with the input we can also compute the error measured as the distance between the estimated image and the true image. This is what the algorithm reports. Running `evalDemosaic` from the Matlab command prompt produces the output shown below:

```
----------------------------------------------------------------------------
#    image              baseline      nn        linear     adagrad
----------------------------------------------------------------------------
1    balloon.jpeg       0.179239    0.179239    0.179239    0.179239
2    cat.jpg            0.099966    0.099966    0.099966    0.099966
3    ip.jpg             0.231587    0.231587    0.231587    0.231587
4    puppy.jpg          0.094093    0.231587    0.231587    0.231587
5    squirrel.jpg       0.121964    0.231587    0.231587    0.231587
6    pencils.jpg        0.183101    0.183101    0.183101    0.183101
7    house.png          0.117667    0.117667   0.117667   0.117667
8    light.png          0.097868    0.097868    0.097868    0.097868
9    sails.png          0.074946    0.074946    0.074946    0.074946
10  tree.jpeg          0.167812   0.167812    0.167812    0.167812
----------------------------------------------------------------------------
    average              0.136824    0.136824    0.136824    0.136824
----------------------------------------------------------------------------
```

Right now only the `demosaicImage(im, 'baseline')` is implemented which simply replaces all the missing values for a channel with the average value of that channel (while using python codes, if you're getting any warnings regarding **iCCP**, do `mogrify *.png` in the `data/demosiac` folder). All the other methods call the baseline algorithm and hence they produce identical results. Implement the following functions in the file:

- **[10 points]** `demosaicImage(im, 'nn')` – nearest-neighbour interpolation.
- **[10 points]** `demosaicImage(im, 'linear')` – linear interpolation.
- **[10 points]** `demosaicImage(im, 'adagrad')` – adaptive gradient interpolation.

In class we discussed how to interpolate the green channel. For the red and blue channels the algorithms will be different since there are half as many pixels with sampled values. For the adaptive gradient method start by interpolating only the green channel and using linear interpolation for the other two channels. Once the overall performance is better, think of a way of improving the interpolation for the red and blue channels. You can even apply different strategies to different pixels for the same channel!

For reference, the baseline method achieves an average error of 0.1392 across the 10 images in the dataset. Your methods you should be able to achieve substantially better results. The linear interpolation method achieves an error of about 0.017. To get full credit for this part you have to

- include your implementation of `demosaicImage`,
- include the output of `evalDemosaic` in a table format shown earlier,
- clearly describe the implementation details.

Tip: You can visualize at the errors by setting the display flag to true in the `runDemosaicing`. Avoid loops for speed in MATLAB/Python. Be careful in handling the boundary of the images.

1.1 My Solution

Here are the errors of my three methods:

```
----------------------------------------------------------------------------
#    image             baseline       nn         linear     adagrad
----------------------------------------------------------------------------
1    balloon.jpeg      0.180700    0.018047    0.013740    0.012922
2    cat.jpg           0.099365    0.023357    0.013701    0.013716
3    ip.jpg            0.199072    0.023273    0.019848    0.017909
4    puppy.jpg         0.093632    0.013210    0.006292    0.006140
5    squirrel.jpg      0.122058    0.038242    0.023286    0.023706
6    pencils.jpg       0.179096    0.027122    0.017621    0.017100
7    house.png         0.116038    0.030960    0.017240    0.015631
8    light.png         0.097154    0.025468    0.017305    0.016551
9    sails.png         0.074997    0.020080    0.013640    0.012876
10  tree.jpeg          0.166748    0.025467    0.015211    0.014293
----------------------------------------------------------------------------
    average            0.132886    0.024523    0.015788    0.015084
----------------------------------------------------------------------------
```

1) Nearest-Neighbour Interpolation

At first, selecting existing pixels at each channel by using slicing operation. (The slicing operation is used in all three methods. In order to save the tutor's reading time, it will not be described in detail in the following two methods)

Copy the original image into three layers, named "mos_img", each layer corresponds to each color channel of RGB.

For the red channel:
Pick the red pixels, i.e. the pixels of odd rows and odd columns (The odd rows and odd columns in here means the real first row, the first column, the third row, the third column...in the matrix).

- Slice the red pixels from the original image and named as "red_value".
- Create a matrix that has the same size as original images, named as "red_channel". All pixel values are -1.
- Given the values of red pixels from "red_value" to "red_channel".

In this way, in the ”red_channel”, all the values of -1 are missing values, and the rest are values of red pixels.

For the blue channel, use the similar method with red channel:
Create ”blue_channel” which missing values are -1 and the rest are values of blue pixels. But the blue pixels are in even columns and even rows.

For the green channel:
It’s easy to represent the positions of red and blue pixels. After removing the red and blue pixels from original images, the rest values are green pixels. In the second layer of ”mos_img”, give -1 to the red and blue pixels. Therefore, the green values can be easily picked by selecting value >= 0.

After the operation of slicing, all the missing values are -1 in three channels. The values >=0 are the original values.

Input different channels into the function of ”assignNearstValue”.

For each pixel(i,j) in each channel:
If the value of a pixel is missing (value = -1), copy the value from the neighbor pixel. The location of neighbor value is nothing more than the following eight cases:

- If pixel(i,j) is not the most left column, and the value of the left pixel is not equal to -1.
  pixel(i,j) = the value of the left pixel
- If pixel(i,j) is not in the top left corner, and the value in the top left corner is not equal to -1.
  pixel(i,j) = the value of the pixel in the top left corner
- If pixel(i,j) is not on the top row, and the above value is not equal to -1.
  pixel(i,j) = the value of the above pixel
- If pixel(i,j) is not in the top right corner, and the value in the top right corner is not equal to -1.
  pixel(i,j) = the value of the pixel in the top right corner
- If pixel(i,j) is not in the last column, and the value of the pixel on the right is not equal to -1.
  pixel(i,j) = value of pixel on the right
- If pixel(i,j) is not in the last row, and the value of the pixel below is not equal to -1.
  pixel(i,j) = the value of one pixel below
- If pixel(i,j) is not in the bottom right corner, and the value in the bottom right corner is not equal to -1.
  pixel(i,j) = the value of the pixel at the bottom right corner
- If pixel(i,j) is not in the bottom left corner, and the value in the bottom left corner is not equal to -1.
  pixel(i,j) = the value of one pixel at the bottom left corner
- If none of the above is true
  Output ”unknown case”.

2) Linear Interpolation

Using the same slicing method as Nearest Neighbor. The original image is copied to three layers, corresponding to three channels. Selecting the existing pixels at each channel.

For each channel, use np.pad to pad an array outside the matrix, i.e. insert a row in the top and bottom of matrix respectively, insert a column in the left and right of matrix respectively. The value of each inserted pixel is -1. The purpose of this step is to allow the algorithm to calculate marginal missing values. The inserted value is -1, all -1 will be ignored as missing values in the following calculations.

Therefore, the result would not be affected. One of the benefits of this method is that it's not necessary to discuss odd or even, as well as rows or columns during calculations. All channels can be input into the algorithm directly regardless of the position of the missing value.

Ignore the value inserted in the surrounding. Pick the missing value in the position of original each channel, i.e. the pixel value =-1. For each missing value (i, j), pick a 3*3 matrix which (i, j) is at the center of the pixel. Assign all negative values to zero. Then all missing value equals to zero. Because it is possible that the original existing pixel value is 0. But a pixel with a value of 0 should not participate in the calculation of the average. Therefore, a pixel with a value of 0 should be treated as a missing value. This situation should also be considered.

Then count how many pixels have existing valid values (non-missing values) in this matrix. Missing value (i, j) = sum of the values of existing valid pixels / the amount of these pixels. If the amount of existing valid pixels in this 3*3 matrix is 0, in order to avoid the denominator being 0, let the pixel (i, j) is still missing. But this situation cannot exist.

After the above calculation, all the missing values are replaced by the average of the actual values in the surrounding 3*3 matrix. Except for the missing value inserted in the outside of the matrix. Clip off the inserted rows and columns, and return the matrix which has the original image size.

3) Adaptive Gradient Interpolation

Using the same slicing method as Nearest Neighbor. The original image is copied to three layers, corresponding to three channels. Selecting the existing pixels at each channel.

Using the pad array method mentioned in the Linear Interpolation. For each channel, insert a row to the top and bottom of the matrix respectively, insert a column to the left and right of matrix respectively. All the inserted value is -1. Considering that the original valid value maybe 0. These pixels should be treated as missing values and should not be used for calculating the average. Therefore, all negative values are assigned to 0.

Start calculating the adaptive gradient:
If the left and right pixel are not missing values, calculate the left and right difference, ”diff_left_right”. If the top and bottom are not missing values, calculate the difference between the up and down, ”diff_top_down”.

If diff_left_right ¡ diff_top_down, sum the value of the pixels left and right / 2
If diff_left_right ¿ diff_top_down, sum the value of the pixels above and down / 2

If the two cases above are not satisfied, use the Linear Interpolation method instead, i.e. pick a 3*3 matrix around the missing value pixel. Count how many pixels have existing valid values (non-missing values) in this matrix. Missing value (i, j) = sum of the values of existing valid pixels / the amount of these pixels. If the amount of existing valid pixels in this 3*3 matrix is 0, in order to avoid the denominator being 0, let the pixel (i, j) is still missing.

After the calculation above, except for the missing value added in the periphery of the matrix, all the missing values are either replaced by the average of the left and right or replaced by the average of the up and down. If the values of the up and down, or left and right are missing, The central missing value is replaced by the average of the existing values in the surrounding 3*3 matrix. Clip off the inserted rows and columns, and return the matrix which has the original image size.

2 Depth from disparity [35 points]

In this part of the homework, you will use a pair of images to compute a depth image of the scene. You will do this by computing a disparity map. A disparity map is an image that stores the displacement that leads every pixel in an image X to its corresponding pixel in the image X'. The depth image is inversely proportional to the disparity, as illustrated in Figure 2.

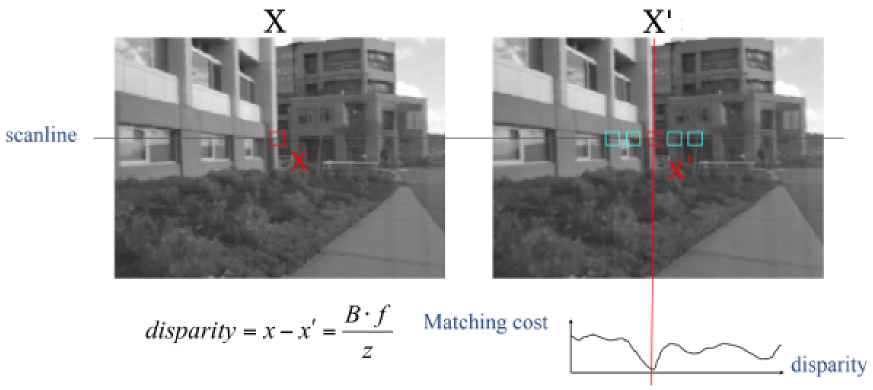

Figure 2: Visual description of depth map computation from a pair of images.

In order to compute the disparity map, you will have to associate each pixel x in X with the corresponding pixel x' in X'. This association can be computed by finding the image patch centered in x' that that has the smallest SSD to the patch centered in x. However, it is not necessary to compute SSD for every patch in X': for this homework, you can assume that the images planes are parallel to each other and have the same focal length, which means that you only need to search for the patches computed along the same epipolar line, as illustrated in Figure 2.

1. **[20 points] Compute depth image.** For this part, you will implement the function `depthFromStereo`. This function receives two images of the same scene and a patch size for SSD computation. It returns an image containing the relative depth. You can assume that the images planes are parallel to each other and have the same focal length. Once you compute the disparity image by finding patch correspondences along the same epipolar line, you can compute the depth through the following equation:

$$x - x' = \frac{Bf}{\text{depth}} \tag{1}$$

   where B is the baseline and f is the focal length. Notice that we don't have access to B and f values, but those are constant throughout both images. This means that you will not be able to compute the true depth image, just the relative depth.

   The entry code for this part is `evalDisparity`. The code loads a pair of images and you will implement a function that estimates the depth of each pixel in the first image, which you can visualize as a grayscale image. Assume $Bf = 1$ in your calculation. Show results for both pair of images included in the data directory.

   To get good results you might have to experiment with the window size and the choice of patch similarity measure. Try a few different ones and report what worked well for you. Include a short discussion of on what parts of the image does the approach work well and where it fails.

   My Answer

   Assume window size=3, the left image is img1, and the right image is img2. Assume r be the radius of the window, i.e. the distance between the center of the window and the row and column will be r.

Read RGB images with rgb2gray and convert them to grayscale images.

Create a two-dimensional zero array named depth. The shape of depth is m*n which get from img2. Pads arrays which thickness is r around img1 and img2. That is pad an array which thickness is r before the first row and after the last row, before the first column and right the last column.

In this way, the range of the row of original image1 becomes $(r, m + r)$, and the range of the column becomes $(r, n+r)$. In the left image img1, pick a patch which centered on pixel (i, j) and window size is 3. X can be donated to $pad\_img1[(i - r) : (i + r + 1), (j - r) : (j + r + 1)]$ in code.

In the right image img2, pick a patch which window size is also 3. Since X and X′ are on the same epipolar line on the same height. Therefore, the line where X and X′ are located can be represented by $(i - r) : (i + r + 1)$. X′ needs to move back and forth on the horizontally direction the epipolar lines to find the position with the smallest SSD with X.

Because there is parallax when the left eye and the right eye look at the same object. That means the projection captured by the right eye is slightly to the left of the projection captured by the left eye. Assuming that the parallax max_diff = 55, create an empty array with a length of max_diff and a width of 1 to represent the parallax. Assuming that the patch in the right image is moving from right to left, then d in range$(j - 1, j - max_diff - 1, -1)$. Then the patch in the right picture can be represented as $pad\_img2[(i - r) : (i + r + 1), (d - r) : (d + r + 1)]$.
Then the SSD can be expressed as measure$[j - 1 - k] = np.sum((left - right) * *2)$.
Use $np.argmin$ to return the index of the smallest SSD.
$Disparity = j - (j - np.argmin(measure))$.
Depth is inversely proportional to disparity and can be calculated by:
$depth[i - r, j - r] = -np.log(1.0/disparity)$.

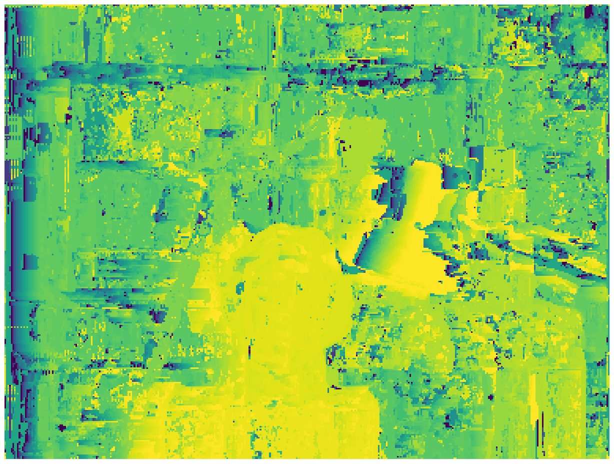
Figure 3: Output of disparity

2. **[5 points] Discussion.** In the first mini-project you implemented a method for depth estimation using photometric stereo. Discuss advantages and disadvantages of stereo over photometric stereo.

   Advantage:

   - Stereo method doesn't have too many requirements on camera hardware. Low cost and easy operation. Just take pictures for the same object twice in different positions. In contrast, photometric stereo has requirements on both camera and light. Photometric stereo is required to take pictures by changing the direction of the light.

- The Stereo method uses ambient light as illumination. It is suitable both indoors and outdoors. It is no need to strictly employ an imaging system in which the illumination is so carefully controlled in the laboratory environment.
- Photometric stereo doesn't work for shiny objects, semi-translucent objects, shadows, and interreflections. But the stereo method is unaffected by these problems.
- The logic of Stereo method is simple and easy to understand. Stereo's method recover depth by finding image coordinate x' that corresponds to x. In addition to finding the corresponding point X', other variables such as baseline, focal length, etc. are easy to measure. It's much easier than know albedo and normal directions.

Disadvantage:

- Stereo method is very expensive. High computational complexity. In order to find the coordinate x' that corresponds to x, this method needs to calculate many matrixes and compare the errors among X' and X. Although Photometric stereo is complex in logic but less in computation.

- Stereo method doesn't work well when monotonous lack of texture scenes. Since the binocular stereo vision method performs image matching according to visual features, such as gradient. It is difficult to match the scenes lacking visual features (such as sky, white wall, desert, etc.). It will lead to matching failures or large errors. Photometric stereo is not limited by texture.

- Stereo method is sensitive to ambient light. It relies on natural light in the environment to capture images. Due to environmental factors such as changes in illumination angle and changes in illumination intensity. The brightness difference between the two images maybe large, which is a great challenge to the matching algorithm. Besides that, the algorithm doesn't work well when the illumination is strong (overexposure) and when it is dark.

- The baseline (the spacing between two cameras) of camera limits the measurement range. The larger the baseline, the farther the measurement range is; the smaller the baseline, the closer the measurement range. Therefore, the baseline limits the measurement range of the depth camera to some extent.

3. **[10 points] Informative patches**. Not all regions are equally informative for matching. As we discussed in class, one way to measure how informative a patch is using its second-moment matrix. For a pixel (u, v) this is defined as:

$$M = \begin{bmatrix} \sum_{x,y} I_x I_x & \sum_{x,y} I_x I_y \\ \sum_{x,y} I_x I_y & \sum_{x,y} I_y I_y \end{bmatrix}, \tag{2}$$

where the summation is over all pixels (x, y) within a window around (u, v). I_x and I_y are the gradients of the image and can be computed using pixel differences, e.g., $I_x(u, v) = I(u + 1, v) - I(u, v)$ and $I_y(u, v) = I_y(u, v+1) - I(u, v)$. The eigenvalues of M indicate if the window is uniform, an edge, or a corner.

Let λ_1 and λ_2 be the eigenvalues of the matrix M. Write a function to compute the quantity $R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$ for $k = 0.03$ for each pixel given an image and window radius r. Visualize this quantity for the images tsukuba_im1 and poster_im2 for radius $r = 5$ and $r = 10$ as a heatmap (or grayscale image). What does the heatmap indicate? Describe this qualitatively.

Include your implementation in the submission (call the method visualizeInformation.) You may find the following identities useful for calculating R. Given a 2×2 matrix $M = [a\ b; c\ d]$, the $det(M) = \lambda_1\lambda_2 = ad - bc$ and $trace(M) = \lambda_1 + \lambda_2 = a + d$.

To get full credit for this part:

- include your implementation of `depthFromStereo`.
- include the computed depth image of the two image pairs we provided `poster` and `tsubuka`. Report which window size, similarity/distance function (if you found a better one over SSD) you used as well a discussion of failure modes.
- include the discussion comparing stereo and photometric stereo for depth estimation.
- include the implementation of `visualizeInformation`
- show a visualization of R for each image for two window sizes
- include discussion what does R indicate.

Tip: The relative depth for the `tsubuka` scene was shown in one of the lectures. The window-based is the one you are implementing. You can use that as reference for debugging purposes.

3 Report Writing and Presentation [10 points]

Please follow the guidelines for writing a good report. Graders will penalize reports that are poorly written and fail to present the results in a reasonable manner.

4 Extensions [10 points]

Implement at least one of the following to get up to 10 points. You can implement multiple for extra credit!

- **Transformed color spaces for demosaicing.** Try your demosaicing algorithms by first interpolating the green channel and then transforming the red and blue channels R $\leftarrow$ R/G and B $\leftarrow$ B/G, i.e., dividing by the green channel and then transforming them back after interpolation. Try other transformations such as logarithm of the ratio, etc (note: you have to apply the appropriate inverse transform). Does this result in better images measured in terms of the mean error? Why is this a good/bad idea?
- **Evaluate alternative sampling patterns.** Come up with a way of finding the optimal 2x2 pattern for color sampling. You can further simplify this problem by fixing the interpolation algorithm to linear and only using non-panchromatic cells (i.e, no white cells). A brute-force approach is to simply enumerate all the $3^4 = 81$ possible patterns and evaluate the error on a set of images and pick the best. What patterns work well? What are their properties (e.g., ratio of red, blue, green, cells)?

5 Code

1. Color Image Demosaicing

1.1 Nearest-Neighbour Interpolation

```
def assignNearstValue(img_channel):
    img_channel_temp = img_channel.copy() #make sure new calculated value doesn't
        affect other

    for i in range(img_channel_temp.shape[0]):
        for j in range(img_channel_temp.shape[1]):
            if (img_channel[i, j] == -1): # value needs to be calculated
                # use left value
                if (i - 1 >= 0 and img_channel[i - 1, j] != -1):
```

```
                img_channel_temp[i, j] = img_channel[i - 1, j]
            # use left upper value
            elif (i - 1 >= 0 and j - 1 >= 0 and img_channel[i - 1, j - 1] != -1):
                img_channel_temp[i, j] = img_channel[i - 1, j - 1]
            # use upper value
            elif (j - 1 >= 0 and img_channel[i, j - 1] != -1):
                img_channel_temp[i, j] = img_channel[i, j - 1]
            # use right upper value
            elif (i + 1 < img_channel_temp.shape[0] and j - 1 <
                 img_channel_temp.shape[1] and img_channel[
                i + 1, j - 1] != -1):
                img_channel_temp[i, j] = img_channel[i + 1, j - 1]
            # use right value
            elif (i + 1 < img_channel_temp.shape[0] and img_channel[i + 1, j] !=
                 -1):
                img_channel_temp[i, j] = img_channel[i + 1, j]

            # use bottom value
            elif (j + 1 < img_channel_temp.shape[1] and img_channel[i, j+1] !=
                 -1):
                img_channel_temp[i, j] = img_channel[i , j+1]
            # use right bottom value
            elif (i + 1 < img_channel_temp.shape[0] and j + 1 <
                 img_channel_temp.shape[1] and img_channel[
                i + 1, j + 1] != -1):
                img_channel_temp[i, j] = img_channel[i + 1, j + 1]
            # use left bottom value
            elif (i - 1 >= 0 and j + 1 >= 0 and img_channel[i - 1, j + 1] != -1):
                img_channel_temp[i, j] = img_channel[i - 1, j + 1]
            else:
                print("unknown case")

    return img_channel_temp


def demosaicNN(img):
    '''Nearest neighbor demosaicing.

    Args:
        img: np.array of size NxM.
    '''

    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
    image_height, image_width = img.shape

    # red channel
    red_values = img[0:image_height:2, 0:image_width:2]
    red_channel = np.ones((image_height, image_width)) * (-1) # default value as -1
    red_channel[0:image_height:2, 0:image_width:2] =
        red_values[0:red_values.shape[0], 0: red_values.shape[1]]
    mos_img[:, :, 0] = assignNearstValue(red_channel)

    # blue channel
    blue_values = img[1:image_height:2, 1:image_width:2]
    blue_channel = np.ones((image_height, image_width)) * (-1) # default value as
        -1
    blue_channel[1:image_height:2, 1:image_width:2] =
        blue_values[0:blue_values.shape[0], 0: blue_values.shape[1]]
    mos_img[:, :, 2] = assignNearstValue(blue_channel)
    # green channel
    green_channel = mos_img[:, :, 1]
    green_channel[0:image_height:2, 0:image_width:2] = -1
    green_channel[1:image_height:2, 1:image_width:2] = -1
    mos_img[:, :, 1] = assignNearstValue(green_channel)

    return mos_img
```

1.2 Linear Interpolation

```
def pad_with(vector, pad_width, iaxis, kwargs):
    pad_value = kwargs.get('padder', -1)
    vector[:pad_width[0]] = pad_value
    vector[-pad_width[1]:] = pad_value

def assignMeanValue(img_channel):
    img_channel_padded = np.pad(img_channel, pad_width=1, mode=pad_with) # pad img
        channel with -1 around
    img_channel_temp = img_channel_padded.copy() # make sure new calculated value
        doesn't affect other
    for i in range(1, img_channel_padded.shape[0]-1): # ignore padded rows
        for j in range(1, img_channel_padded.shape[1] -1): # ignore padded columns
            if(img_channel_padded[i, j] == -1): # value needs to be calculated
                neighbour_pixels = img_channel_padded[i-1: i+2, j-1:j+2].copy() # 3*3
                    matrix
                neighbour_pixels[neighbour_pixels<0] = 0 # assign all negative values
                    to zero
                valid_num_of_neighbour_pixels = np.count_nonzero(neighbour_pixels)
                if(valid_num_of_neighbour_pixels == 0): # avoid divide by zero
                    img_channel_temp[i, j] = 0
                else:
                    img_channel_temp[i, j] = np.sum(neighbour_pixels) /
                        valid_num_of_neighbour_pixels

    return img_channel_temp[1:img_channel_temp.shape[0]-1,
        1:img_channel_temp.shape[1]-1] # added padding values should not be
        considered

def demosaicLinear(img):
    '''Nearest neighbor demosaicing.

    Args:
        img: np.array of size NxM.
    '''
    mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
    image_height, image_width = img.shape

    # red channel
    red_values = img[0:image_height:2, 0:image_width:2]
    red_channel = np.ones((image_height, image_width)) * (-1) # default value as -1
    red_channel[0:image_height:2, 0:image_width:2] =
        red_values[0:red_values.shape[0], 0: red_values.shape[1]]
    mos_img[:, :, 0] = assignMeanValue(red_channel)

    # blue channel
    blue_values = img[1:image_height:2, 1:image_width:2]
    blue_channel = np.ones((image_height, image_width)) * (-1) # default value as
```

```
        -1
    blue_channel[1:image_height:2, 1:image_width:2] =
        blue_values[0:blue_values.shape[0], 0: blue_values.shape[1]]
    mos_img[:, :, 2] = assignMeanValue(blue_channel)

    # green channel
    green_channel = mos_img[:, :, 1]
    green_channel[0:image_height:2, 0:image_width:2] = -1
    green_channel[1:image_height:2, 1:image_width:2] = -1
    mos_img[:, :, 1] = assignMeanValue(green_channel)

    return mos_img
```

1.3 Adaptive Gradient Interpolation

```
def assignLeastMeanValue(img_channel):
    img_channel_padded = np.pad(img_channel, pad_width=1, mode=pad_with) # pad img
        channel with -1 around
    img_channel_temp = img_channel_padded.copy() # make sure new calculated value
        doesn't affect other
    for i in range(1, img_channel_padded.shape[0] - 1): # ignore padded rows
        for j in range(1, img_channel_padded.shape[1] - 1): # ignore padded columns
            if (img_channel_padded[i, j] == -1): # value needs to be calculated
                neighbour_pixels = img_channel_padded[i - 1: i + 2, j - 1:j +
                    2].copy() # 3*3 matrix
                neighbour_pixels[neighbour_pixels < 0] = 0 # assign all negative
                    values to zero
                diff_left_right = 100
                diff_top_down = 100
                if(neighbour_pixels[1,0]>0 and neighbour_pixels[1,2]>0 ): # we can
                    calculate the difference between left and right pixels
                    diff_left_right = abs(neighbour_pixels[1,0] -
                        neighbour_pixels[1,2])
                if (neighbour_pixels[0, 1] > 0 and neighbour_pixels[2, 1] > 0): # we
                    can calculate the difference between top and down pixels
                    diff_top_down = abs(neighbour_pixels[0,1] - neighbour_pixels[2,1])

                if(diff_left_right < diff_top_down): # we can use adagrad method, and
                    it should base on horizontal direction
                    img_channel_temp[i, j] = (neighbour_pixels[1, 0] +
                        neighbour_pixels[1, 2]) / 2
                elif(diff_left_right > diff_top_down): # we can use adagrad method,
                    and it should base on vertical direction
                    img_channel_temp[i, j] = (neighbour_pixels[0, 1] +
                        neighbour_pixels[2, 1]) / 2
                else: # we can not use adagrad method, use linear method instead
                    valid_num_of_neighbour_pixels = np.count_nonzero(neighbour_pixels)
                    if (valid_num_of_neighbour_pixels == 0): # avoid divide by zero
                        img_channel_temp[i, j] = 0
                    else:
                        img_channel_temp[i, j] = np.sum(neighbour_pixels) /
                            valid_num_of_neighbour_pixels

    return img_channel_temp[1:img_channel_temp.shape[0] - 1,
        1:img_channel_temp.shape[1] - 1] # added padding values should not be
        considered

def demosaicAdagrad(img):
```

```
'''Nearest neighbor demosaicing.

Args:
   img: np.array of size NxM.
'''
mos_img = np.tile(img[:, :, np.newaxis], [1, 1, 3])
image_height, image_width = img.shape

# red channel
red_values = img[0:image_height:2, 0:image_width:2]
red_channel = np.ones((image_height, image_width)) * (-1) # default value as -1
red_channel[0:image_height:2, 0:image_width:2] =
    red_values[0:red_values.shape[0], 0: red_values.shape[1]]
mos_img[:, :, 0] = assignLeastMeanValue(red_channel)

# blue channel
blue_values = img[1:image_height:2, 1:image_width:2]
blue_channel = np.ones((image_height, image_width)) * (-1) # default value as
    -1
blue_channel[1:image_height:2, 1:image_width:2] =
    blue_values[0:blue_values.shape[0], 0: blue_values.shape[1]]
mos_img[:, :, 2] = assignLeastMeanValue(blue_channel)

# green channel
green_channel = mos_img[:, :, 1]
green_channel[0:image_height:2, 0:image_width:2] = -1
green_channel[1:image_height:2, 1:image_width:2] = -1
mos_img[:, :, 1] = assignLeastMeanValue(green_channel)

return mos_img
```