Homework 3 Writeup

Instructions

- Describe any interesting decisions you made to write your algorithm.
- Show and discuss the results of your algorithm.
- Feel free to include code snippets, images, and equations.
- **Please make this document anonymous.**

Bayer Image Interpolation

First part of this project is to make the code that interpolating the given bayer image into proper RGB image.

```
def bayer_to_rgb_bilinear(bayer_img):

    height, width = bayer_img.shape
    rgb_img = np.zeros((height + 2, width + 2, 3), dtype=np.uint8)
    # added zero padding of 1 pixel width around the image for
    calculation efficiency
    # Extract bayer image into each channel of rgb image
    # Range starts from 1 due to the padding
    rgb_img[1:-1:2, 1:-1:2, 0] = bayer_img[0::2, 0::2]
    rgb_img[1:-1:2, 2:-1:2, 1] = bayer_img[0::2, 1::2]
    rgb_img[2:-1:2, 1:-1:2, 1] = bayer_img[1::2, 0::2]
    rgb_img[2:-1:2, 2:-1:2, 2] = bayer_img[1::2, 1::2]

    # For the R channel
    rgb_img[1:height:2, 2:width+1:2, 0] = (rgb_img[1:height:2, 1:width
    :2,0]//2 +
    rgb_img[1:height:2, 3:width+2:2,0]//2)
    rgb_img[2:height+1:2, 1:width:2, 0] = (rgb_img[1:height:2, 1:width
    :2,0]//2 +
    rgb_img[3:height+2:2, 1:width:2,0]//2)

    rgb_img[2:height+1:2, 2:width+1:2, 0] = (
        rgb_img[1:height:2, 1:width:2, 0]//4 + rgb_img[1:height:2, 3:
    width+2:2, 0]//4 +
        rgb_img[3:height+2:2, 1:width:2, 0]//4 + rgb_img[3:height+2:2,
     3:width+2:2, 0]//4
    )
    #For the G channel
    rgb_img[1:-1:2, 1:-1:2, 1] = (
        rgb_img[1:-1:2, 0:-2:2, 1]//4 +
        rgb_img[0:-2:2, 1:-1:2, 1]//4 +
        rgb_img[2::2, 1:-1:2, 1]//4 +
        rgb_img[1:-1:2, 2::2, 1]//4
    )
    rgb_img[2:height+1:2, 2:width+1:2, 1] = (
```

```
        rgb_img[2:height+1:2, 1:width:2, 1]//4 +
        rgb_img[1:height:2, 2:width+1:2, 1]//4 +
        rgb_img[3:height+2:2, 2:width+1:2, 1]//4 +
        rgb_img[2:height+1:2, 3:width+2:2, 1]//4
    )

    # For the B channel
    rgb_img[2::2, 1:-2:2, 2] = (rgb_img[2::2, 0:-3:2,2]//2 +
    rgb_img[2::2, 2:-1:2,2]//2)

    rgb_img[1:-2:2, 2::2, 2] = (rgb_img[0:-3:2, 2::2,2]//2 +
    rgb_img[2:-1:2, 2::2,2]//2)

    rgb_img[1:-2:2, 1:-2:2, 2] = (
        rgb_img[0:-3:2, 0:-3:2, 2]//4 +
        rgb_img[0:-3:2, 2:-1:2, 2]//4 +
        rgb_img[2:-1:2, 0:-3:2, 2]//4 +
        rgb_img[2:-1:2, 2:-1:2, 2]//4
    )
    #Remove Padding
    rgb_img = rgb_img[1:-1, 1:-1, :]

    return rgb_img

```

Interesting Implementation Detail

I'll explain my code step by step.

1. First, I made the empty array for result image with 2 pixels bigger than bayer image. This is because I wanted to add zero padding of 1 pixel around the image, for edge calculation.
2. Then I brought the each R,G,B pixels from the bayer image and applied to channels in R,G,B order. To apply padding, i left the first and last pixels zero and filled the pixel value starting from 1.
3. For each channel, used bilinear interpolation to fill the missing pixel values. I used indexing and slicing of Numpy to do this, since it is much faster than naive for loops.
4. For the pixels in edges, I just used the same ways with other pixels, the only difference is that averaged the zero padding together. This makes the edge value smaller than intended, but it could the code more precise.
5. Finally, I removed the zero padding by slicing the result image from the first pixel to the second last pixel.

These are two images obatained from bilinear interpolation.

Figure 1: Images obtained from bilienear interpolation

The problem in this implementation is as mentioned, the edge control. I think this could be resolved by using other methods for edges, such as reflection.

Fundamental Matrix Calculation

In this section, with given 8 points on both images I obtained the fundamental matrix, using eight-point algorithm.

```
def calculate_fundamental_matrix(pts1, pts2):
    assert pts1.shape[1] == 2 and pts2.shape[1] == 2
    assert pts1.shape[0] == pts2.shape[0]

    n = pts1.shape[0]


    pts1_normalized, T1 = normalize_points(pts1.T, 2)
    pts2_normalized, T2 = normalize_points(pts2.T, 2)

    # Transpose back to original structure
    pts1_normalized = pts1_normalized.T
    pts2_normalized = pts2_normalized.T

    A = np.zeros((n, 9))
    for i in range(n):
        x1, y1 = pts1_normalized[i]
        x2, y2 = pts2_normalized[i]
        A[i] = [x1*x2, x2*y1, x2, y2*x1, y1*y2, y2, x1, y1, 1]

    U, S, Vt = np.linalg.svd(A)
    f = Vt[-1]
    F = f.reshape(3, 3)

    U, S, Vt = np.linalg.svd(F)
    S[-1] = 0
    F = U @ np.diag(S) @ Vt

    # Denormalize
    F = T2.T @ F @ T1

    return F
```

1. **Normalization** First, I normalized the given points using the built-in fuction $normalize\_points()$. Since the function needs input coordinates with x,y in column, I transposed the given points vector and applied the function. Then I transposed the result of function to obtain original structure.

2. **Constructing Matrix A** Matrix A is constructed using the normalized points. I just simply put each values in the matrix, and the values come from what we learned in class and supplementary slides.

3. **Singular Value Decomposition and Reshaping f** Performed singular value decomposition for matrix A, to obtain f.Since f corresponds to the eigenvector of A^TA with the smallest eigenvalue, I extracted the last row of Vt. Then, reshaped it into 3x3 matrix. Matrix F is 3x3 fundmental matrix obtained from SVD.

4. **Enforce F to have Rank 2, and Recalculate F** Performed singular value decomposition again for matrix F. To enforce rank 2 condition, I set the last row of S(minimum sigular value) zero and recalculated F with new S.

5. **Denormalization** Finally, I denormalized fundamental matrix F with scaling matrix $T1$ and $T2$ to bring it back to the scale of the original points.

Stereo Rectification

Calculating Disparity Map

The final, and the most complex part of this project is cacluating the disparity map.

```
def calculate_disparity_map(img1, img2):

    # First convert color image to grayscale
    img1_gray = cv2.cvtColor(img1, cv2.COLOR_RGB2GRAY)
    img2_gray = cv2.cvtColor(img2, cv2.COLOR_RGB2GRAY)
    # You have to get disparity (depth) of img1 (left)
    # i.e., I1(u) = I2(u + d(u)),
    # where u is pixel positions (x,y) in each images and d is
    dispairty map.
    # Your code here


    h, w = img1_gray.shape
    half_window = WINDOW_SIZE // 2

    # Initialize disparity map and cost volume
    disparity_map = np.zeros((h, w), dtype=np.float32)
    cost_volume = np.zeros((h, w, DISPARITY_RANGE), dtype=np.float32)

    for d in range(DISPARITY_RANGE):
        print(f"Calculating disparity {d}")

        img2_shifted = np.roll(img2_gray, d, axis=1)
```

```
23
24          for y in range(half_window, h - half_window):
25              # Vectorized inner loop
26              w1 = img1_gray[y - half_window:y + half_window + 1,
    half_window:w - half_window]
27              w2 = img2_shifted[y - half_window:y + half_window + 1,
    half_window:w - half_window]
28
29              mean_w1 = np.mean(w1, axis=0)
30              mean_w2 = np.mean(w2, axis=0)
31
32              numerator = np.sum((w1 - mean_w1) * (w2 - mean_w2), axis
    =0)
33              denominator = np.sqrt(np.sum((w1 - mean_w1)**2, axis=0) *
    np.sum((w2 - mean_w2)**2, axis=0))
34
35              ncc = -1 if denominator == 0 else numerator / denominator
36              cost_volume[y, half_window:w - half_window, d] = -ncc
37
38      # Aggregate costs
39      radius = 40
40      epsilon = 0.2**2  # Epsilon in the Guided Filter
41      gf = cv2.ximgproc.createGuidedFilter(img1_gray, radius, epsilon)
42
43      for d in range(DISPARITY_RANGE):
44          cost_volume[:, :, d] = gf.filter(cost_volume[:, :, d])
45
46      # Get disparity map
47      disparity_map = -np.argmin(cost_volume, axis=2)
48      return disparity_map
```

1.

There are some points that I considered to make this code.

```
1       for d in range(DISPARITY_RANGE):
2           print(f"Calculating disparity {d}")
3
4           img2_shifted = np.roll(img2_gray, d, axis=1)
5
6           for y in range(half_window, h - half_window):
7               for x in range(half_window, w - half_window):
8                   # Extract square windows
9                   w1 = img1_gray[y - half_window:y + half_window + 1, x
    - half_window:x + half_window + 1]
10                  w2 = img2_shifted[y - half_window:y + half_window + 1,
     x - half_window:x + half_window + 1]
11
12                  mean_w1 = np.mean(w1)
13                  mean_w2 = np.mean(w2)
14
15                  numerator = np.sum((w1 - mean_w1) * (w2 - mean_w2))
16                  denominator = np.sqrt(np.sum((w1 - mean_w1)**2) * np.
    sum((w2 - mean_w2)**2))
```

```
17
18                  ncc = -1 if denominator == 0 else numerator /
    denominator
19                  cost_volume[y, x, d] = -ncc
```

This was the original for loop that I made. However, three loops together made a lot of computational time; so I had to reduce the number of loops. In original approach, I calculated NCC with two boxes in each images, so I had to traverse two loops for x and y coordinates. To get rid of this loop, I chose to use the window that has a width with same of image width, and height of $WINDOW\_SIZE$. I calculated the total mean for this area, and applied same calculated NCC value to pixels with coordinate y and all the pixels in the x coordinate of window. I thought that since this approach compares the value of horizontal block, it would not capture the local pixel difference as much as the previous one, but would be much faster.

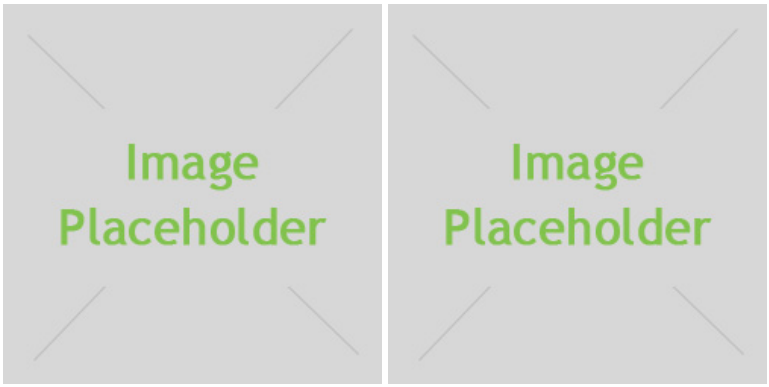

Figure 2: *Left:* My result was spectacular. *Right:* Curious.

My results are summarized in Table 1.

Condition	Time (seconds)
Test 1	1
Test 2	1000

Table 1: Stunning revelation about the efficiency of my code.