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

```
def transform_fundamental_matrix(F, h1, h2):


    F_mod = np.linalg.inv(h2).T @ F @ np.linalg.inv(h1)

    return F_mod

def rectify_stereo_images(img1, img2, h1, h2):

    height, width = img1.shape[:2]

    pts = np.float32([[0, 0], [width - 1, 0], [width - 1, height - 1],
     [0, height - 1]]).reshape(-1, 1, 2)

    dst1 = cv2.perspectiveTransform(pts, h1)
    dst2 = cv2.perspectiveTransform(pts, h2)
    # Calculate the bounding box dimensions
    x_min1, y_min1 = np.int32(dst1.min(axis=0).ravel())
    x_max1, y_max1 = np.int32(dst1.max(axis=0).ravel())
    x_min2, y_min2 = np.int32(dst2.min(axis=0).ravel())
    x_max2, y_max2 = np.int32(dst2.max(axis=0).ravel())

    x_min = min(x_min1, x_min2)
    y_min = min(y_min1, y_min2)
    x_max = max(x_max1, x_max2)
    y_max = max(y_max1, y_max2)

    # Create new translation matrices based on the maximum bounding
    box
```

```
    T1 = np.array([[1, 0, -x_min + 50],
            [0, 1, -y_min + 50],
            [0, 0, 1]])
    T2 = np.array([[1, 0, -x_min + 50],
                [0, 1, -y_min + 50],
                [0, 0, 1]])

    h1_mod = T1 @ h1
    h2_mod = T2 @ h2

    new_dims = (x_max - x_min + 100, y_max - y_min + 100)


    # Warp the images
    img1_rectified = cv2.warpPerspective(img1, h1_mod, new_dims)
    img2_rectified = cv2.warpPerspective(img2, h2_mod, new_dims)


    return img1_rectified, img2_rectified, h1_mod, h2_mod
```

1. **Defining Corner Points** First, I defined the corner points of the imgae in pts. After calculating new homography, the points are translated to new point with it to obtain the boundary of rectified image.

2. **Calculate Transformed Points** The corner points are transformed using the original homography matrices H and H' for both images. I used $cv2.perspectiveTransform()$ with corner points and homography matrix, to apply transform in the corner points.

3. **Bounding Box Calculation** I obatined the coordinates of two bounding boxes from each

```
x_min1, y_min1 = np.int32(dst1.min(axis=0).ravel())
x_max1, y_max1 = np.int32(dst1.max(axis=0).ravel())
```

4. **Maximum Bounding Box:** A common bounding box is computed that can contain both rectified images.

```
x_min = min(x_min1, x_min2)
y_min = min(y_min1, y_min2)
x_max = max(x_max1, x_max2)
y_max = max(y_max1, y_max2)
```

5. **Translation Matrices:** Translation matrices are calculated to place the transformed images within the maximum bounding box with an added margin.

```
T1 = np.array([[1, 0, -x_min + margin], [0, 1, -y_min + margin], [
```

6. **Modified Homographies:** The original homography matrices are modified using the translation matrices.

```
h1_mod = T1 @ h1
h2_mod = T2 @ h2
```

7. **Image Warping:** Finally, the ‘cv2.warpPerspective‘ function is used to warp the images using the modified homography matrices and obtain the rectified images.

```
img1_rectified = cv2.warpPerspective(img1, h1_mod, new_dims)
img2_rectified = cv2.warpPerspective(img2, h2_mod, new_dims)
```

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)

        for y in range(half_window, h - half_window):
            # Vectorized inner loop
            w1 = img1_gray[y - half_window:y + half_window + 1,
    half_window:w - half_window]
            w2 = img2_shifted[y - half_window:y + half_window + 1,
    half_window:w - half_window]

            mean_w1 = np.mean(w1, axis=0)
            mean_w2 = np.mean(w2, axis=0)

```

```
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      # Cost Aggregation
39      radius = 40
40      epsilon = 0.1
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. **Initialization** First initialized the disparity map and cost volume with zero numpy array.

2. **Calculating NCC, part 1** For all disparity d, for each window, I calculated NCC. First I shifted $img2$ by d with $np.roll()$. The reason I used this function is for computational efficiency (vectorized calculation) and for edge constraints. Then I calculated the mean of each window in $img1$ and $img2$. For the window size i used total width of the image and and the height of the window for each y coordinate. This is explained in more detail below.

3. **Calculating NCC, part 2** Then i calculated the numerator and denominator of NCC with respect to the NCC equation. To get rid of the case if denominator is zero, I set the ncc value -1 in that case. Finally, I put cacluated NCC cost in the $cost\_volume$ matrix for all y coordinates of image.

4. **Cost Aggregation** For cost aggregation, I used guided filter. $cv2.ximgproc.createGuidedFilter()$ creates a guided filter with given image, and parameters of window size $radius$ and regularization term $epsilon$. I first created the guided filter with reference image $img1$, and then applied it to each layers of the cost volume traversing through disparities.

5. **Get Disparity Map** Finally, I obtained the disparity map by choosing the disparity of pixels with minimum cost. I used $np.argmin()$ along the disparity axis of the cost volume. The reason why I made the value negative is because when I didn't, the result disparity map showed just an opposite result from the answer in slides. The deep area had blue color, and shallow area had red color. When I put minus on the value, bad pixel error decreased tremendously. I was unable to figure out why this happen, so I had to just choose the way with lower error.

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

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

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

        for y in range(half_window, h - half_window):
            for x in range(half_window, w - half_window):
                # Extract square windows
                w1 = img1_gray[y - half_window:y + half_window + 1, x - half_window:x + half_window + 1]
                w2 = img2_shifted[y - half_window:y + half_window + 1, x - half_window:x + half_window + 1]

                mean_w1 = np.mean(w1)
                mean_w2 = np.mean(w2)

                numerator = np.sum((w1 - mean_w1) * (w2 - mean_w2))
                denominator = np.sqrt(np.sum((w1 - mean_w1)**2) * np.sum((w2 - mean_w2)**2))

                ncc = -1 if denominator == 0 else numerator / denominator
                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. The computation result is in 2.

Choosing the Hyperparameters

With $WINDOW\_SIZE$ of 30 and $DISPARITY\_RANGE$ of 40, the result was as below;

```
Calculating disparity 39
[ PASS ] Disparity computation time: 7.04 seconds
[ 15 pts ] EPE: 2.1975
[ 15 pts ] Bad pixel ratio: 18.48%
[ PASS (d) ]
```

Figure 2: Computation Result with window size=30 and disparity range=40

Choosing the disprity range was quite easy; I first set the max disparity a large value, such as 100. Then I checked out that the most of the disparity are located between 0 to

50. I choosed 40 as max disparity to get rid of the outliers and to calculate bit tightly for large disparities.

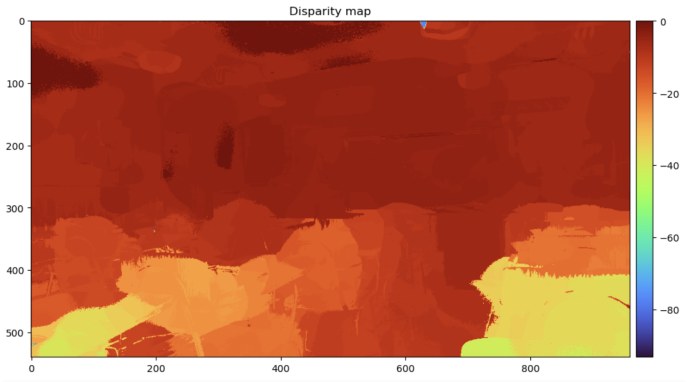

Figure 3: Disparity map with max disparity 100. We could notice that most of disparity is smaller than 50.

Choosing window size was bit confusing. At the beginning I thought the small value will give better result, but oppositely, large size of window made better result.

Window Size	Bad Pixel Ratio (percent)	Computation Timeseconds
10	22.74	4.75
20	19.67	5.64
30	18.48	6.87
40	18.32	7.60
50	18.56	8.68

Table 1: Window Size vs Bad Pixel Ratio, all other conditions identical.

As noticed in 1, the bad pixel ratio start to increase again around 50. I choosed 30 for window size, because bit larger value would give better error ratio but worse in compuational time.

Next, I choosed the parameters for the guided filter, $radius$ and $epsilon$. For $radius$, I used 40, and for epsilon I used 0.1. Actually the change in this parameter affected the result a lot than I expected, so i just choosed the value empirically. What I noticed that in this project, more smoothing usually leads to better error ratio. Since larger radius and smaller epsilon leads to more smoothing and less edge preservation, it produced better result in error ratio. I think this would vary with different ground truth disparity maps.

Sub-Pixel Disparity

I additionally implemented the Sub-Pixel Disparity.

```python
#Sub-Pixel Disparity
limit = 1e-5  # Smoothing term to prevent division by zero
max_subpixel_correction = 0.5

for y in range(h):
    for x in range(w):
        d = disparity_map[y, x]
        if d == 0 or d == DISPARITY_RANGE - 1:
            continue
        C0 = cost_volume[y, x, d-1]
        C1 = cost_volume[y, x, d]
        C2 = cost_volume[y, x, d+1]

        denominator = 2*(C2 + C0 - 2*C1)

        # Check if the denominator is too small or zero
        if np.abs(denominator) < limit:
            continue
        subpixel_correction = (C2 - C0) / denominator
        subpixel_correction = np.clip(subpixel_correction, -
    max_subpixel_correction, max_subpixel_correction)

        disparity_map[y, x] = d + subpixel_correction
```

1. **Limiting the range of Value** $epsilon$ and $max\_subpixel\_correction$ is both the parameter to limit the range of value for subpixel. This will be explained later.

2. **Parabolic Fitting** I applied parabolic fitting for each coordinates and three disparities in a row, with regards to this equation:

$$d_{\text{sub-pixel}} = d + \frac{C(d-1) - C(d+1)}{2(C(d-1) - 2C(d) + C(d+1))}$$

   which is a parabolic fitting crossing three points.

   Since the denominator value could go to zero, I excluded the pixels with denominator value with smaller than limit.

3. **Max Correction** Even after I applied limit, some of the denominator values were too small, making those sub-pixel disparity very large. Thus, I set the max correction value and applied $np.clip()$ to limit the value of correction.

After applying Sub-Pixel Disparity, the bad pixel ratio **decreased about 0.08 percent**. Though my implementation is not perfect, with a better correction It would give a better result with correction. The disparity map before and after applying sub-pixel disparity is shown in 4.

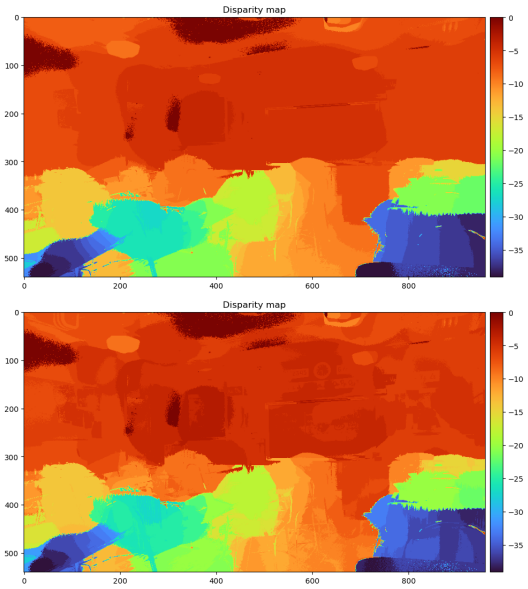

Figure 4: Top: Withoud Sub-Pixel, Bottom: With Sub-Pixel.