Homework 3 Writeup

Overview

In this lab, I have to implement image rectification and a basic stereo algorithm. There are 4 tasks given in this homework.

1. Bayer image is converted to RBG image through image interpolation. In this case, only bilinear interpolation was used.
2. Find fundamental matrix using the normalized eight point algorithm.
3. Rectify stereo images by applying homography.
4. Make a cost volume using NCC matching cost function for the two rectified images, then obtain disparity map from the cost volume after aggregate it with a box filter.

First, I will explain codes and then, report the results.

Code Explanation: Bayer to RGB

First, define the width of Bayer image as W and the height as H. And define RGB image to return in advance.

```
W = bayer_img.shape[1]
H = bayer_img.shape[0]

rgb_img = np.zeros((H, W, 3))
```

Next, move the values from Bayer image to RGB image. Of the 4 pixels in each 2x2 windows, the upper left is red, the lower right is blue, and the rest are green.

```
for i in range(H):
  for j in range(W):
    if i % 2 == 0 and j % 2 == 0:
      rgb_img[i,j,0] = bayer_img[i,j]
    elif i % 2 == 0 and j % 2 == 1:
      rgb_img[i,j,1] = bayer_img[i,j]
    elif i % 2 == 1 and j % 2 == 0:
      rgb_img[i,j,1] = bayer_img[i,j]
    else:
      rgb_img[i,j,2] = bayer_img[i,j]
```

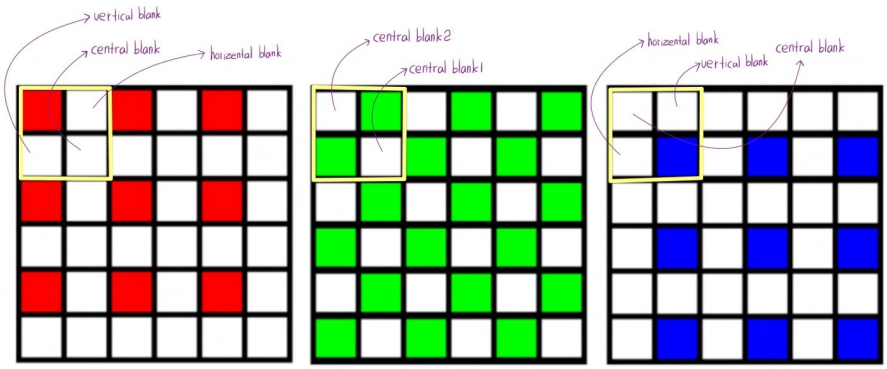

Figure 1: *C*lassified blanks

Next, fill in the blanks in RGB image. Each blank is classified as shown in Figure 1. The values in blanks are calculated using bilnear interporation.(I use round function which uses round to even method)

The code of red filter is as follows.

```
#Horizental blank
for i in range(0,H-1,2):
  for j in range(1,W-2,2):
    rgb_img[i,j,0] =
        round((rgb_img[i,j-1,0]+rgb_img[i,j+1,0])/2)

#Vertical blank
for i in range(1,H-2,2):
  for j in range(0,W-1,2):
    rgb_img[i,j,0] =
        round((rgb_img[i-1,j,0]+rgb_img[i+1,j,0])/2)

#Central blank
for i in range(1,H-2,2):
  for j in range(1,W-2,2):
    rgb_img[i,j,0] =
        round((rgb_img[i-1,j-1,0]+rgb_img[i-1,j+1,0]
      +rgb_img[i+1,j-1,0]+rgb_img[i+1,j+1,0])/4)
```

The code of green filter is as follows.

```
#Central blank 1
for i in range(1,H-2,2):
  for j in range(1,W-2,2):
    rgb_img[i,j,1] = round((rgb_img[i,j-1,1]+rgb_img[i,j+1,1]
      +rgb_img[i-1,j,1]+rgb_img[i+1,j,1])/4)

#Central blank 2
for i in range(2,H-1,2):
```

```
  for j in range(2,W-1,2):
    rgb_img[i,j,1] = round((rgb_img[i,j-1,1]+rgb_img[i,j+1,1]
      +rgb_img[i-1,j,1]+rgb_img[i+1,j,1])/4)
```

The code of blue filter is as follows.

```
#Horizental blank
for i in range(1,H,2):
  for j in range(2,W-1,2):
    rgb_img[i,j,2] =
        round((rgb_img[i,j-1,2]+rgb_img[i,j+1,2])/2)

#Vertical blank
for i in range(2,H-1,2):
  for j in range(1,W,2):
    rgb_img[i,j,2] =
        round((rgb_img[i-1,j,2]+rgb_img[i+1,j,2])/2)

#Central blank
for i in range(2,H-1,2):
  for j in range(2,W-1,2):
    rgb_img[i,j,2] =
        round((rgb_img[i-1,j-1,2]+rgb_img[i-1,j+1,2]
      +rgb_img[i+1,j-1,2]+rgb_img[i+1,j+1,2])/4)
```

The edges of RBG image have to be processed separately. Because it is not regular in that positions. Edge positions can be distinguished as shown in Figure 2. Edges are processed by averaging pixel values around each location.

- Red: 1 for vertical edge, 2 for horizental edge, 3 for bottom-right edge
- Green: 1 for up-horizental edge, 2 for bottom-horizental edge, 3 for left-vertical edge, 4 for right-vertical edge, 5 for bottom-right edge, 6 for up-left edge
- Blue: 1 for vertical edge, 2 for horizental edge, 3 for up-left edge

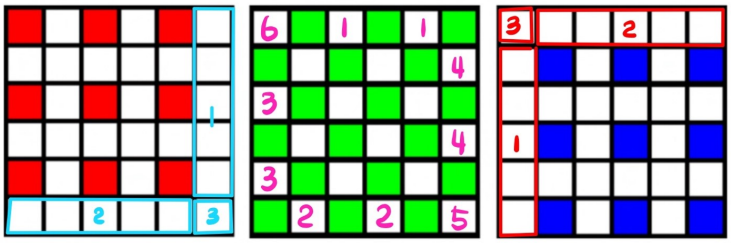

Figure 2: *C*lassified edge blanks

The code of red filter is as follows.

```
#1
for i in range(H-1):
  rgb_img[i,W-1,0] = rgb_img[i,W-2,0]
#2
for j in range(W-1):
  rgb_img[H-1,j,0] = rgb_img[H-2,j,0]
#3
rgb_img[H-1,W-1,0] = rgb_img[H-2,W-2,0]
```

The code of green filter is as follows.

```
#1
for j in range(2,W-1,2):
  rgb_img[0,j,1] =
     round((rgb_img[0,j-1,1]+rgb_img[0,j+1,1]+rgb_img[1,j,1])/3)
#2
for j in range(1,W-2,2):
  rgb_img[H-1,j,1] =
     round((rgb_img[H-1,j-1,1]+rgb_img[H-1,j+1,1]+rgb_img[H-2,j,1])/3)
#3
for i in range(2,H-1,2):
  rgb_img[i,0,1] =
     round((rgb_img[i-1,0,1]+rgb_img[i+1,0,1]+rgb_img[i,1,1])/3)
#4
for i in range(1,H-2,2):
  rgb_img[i,W-1,1] =
     round((rgb_img[i-1,W-1,1]+rgb_img[i+1,W-1,1]+rgb_img[i,W-2,1])/3)
#5
rgb_img[H-1,W-1,1] =
   round((rgb_img[H-1,W-2,1]+rgb_img[H-2,W-1,1])/2)
#6
rgb_img[0,0,1] = round((rgb_img[0,1,1]+rgb_img[1,0,1])/2)
```

The code of blue filter is as follows.

```
#1
for i in range(H-1):
  rgb_img[i,0,2] = rgb_img[i,1,2]
#2
for j in range(W-1):
  rgb_img[0,j,2] = rgb_img[1,j,2]
#3
  rgb_img[0,0,2] = rgb_img[1,1,2]
```

Image type have to be uint8 and return RGB image.

```
rgb_img = rgb_img.astype(np.uint8)
```

Code Explanation: Calculate Fundamental Matrix

First, 8 of input points are selected and points array is transposed. And by normalizing points, normalized points and normalizing transformation are obtained.

```
pts1 = pts1[:8,:].T
pts2 = pts2[:8,:].T
npts1, T1 = normalize_points(pts1, 2)
npts2, T2 = normalize_points(pts2, 2)
```

Next, A on page 16 of hw3_supplementary_slides.pdf is obtained.

```
A = A = np.zeros((8,9))
for i in range(8):
  x1 = npts1[0,i]
  x2 = npts2[0,i]
  y1 = npts1[1,i]
  y2 = npts2[1,i]

  A[i,0] = x1*x2
  A[i,1] = x1*y2
  A[i,2] = x1
  A[i,3] = y1*x2
  A[i,4] = y1*y2
  A[i,5] = y1
  A[i,6] = x2
  A[i,7] = y2
  A[i,8] = 1
```

Get eigenvector of

$$A^T A$$

And get f that is eigenvector correspoinding the smallest eigenvalue (on page 16 of hw3_supplementary_slides.pdf). Then, get F reshaping f (on page 15 of same file).

```
eig_val, eig_vec= np.linalg.eig(A.T@A)
f = eig_vec[:,-1]
F = np.reshape(f,(3,3)).T
```

Using SVD, get U, S, V. By eliminating minimum singular value for S, get new F.

```
U, pre_S, VT = np.linalg.svd(F)
S = np.zeros((3,3))
S[0,0] = pre_S[0]
S[1,1] = pre_S[1]
F = U@S@VT
```

Get fundamental matrix in original coordinates and return.

```
fundamental_matrix = T2.T@F@T1
```

Code Explanation: Rectify Stereo Images

First, rectify each corner of image.

```
H1,W1,C1 = img1.shape
points1 =
    np.array([[0,0],[0,H1-1],[W1-1,H1-1],[W1-1,0]]).reshape(-1,1,2)
points1 = points1.astype(np.float32)
points1 = cv2.perspectiveTransform(points1,h1)

H2,W2,C2 = img1.shape
points2 =
    np.array([[0,0],[0,H2-1],[W2-1,H2-1],[W2-1,0]]).reshape(-1,1,2)
points2 = points2.astype(np.float32)
points2 = cv2.perspectiveTransform(points2,h2)
```

Next, find the range of x and y coordinates that can appear when two images are merged. That is, the smallest and highest values of each are calculated.

```
W_max =
    np.max([points1[0,0,0],points1[1,0,0],points1[2,0,0],points1[3,0,0]
   ,points2[0,0,0],points2[1,0,0],points2[2,0,0],points2[3,0,0]])
W_min =
    np.min([points1[0,0,0],points1[1,0,0],points1[2,0,0],points1[3,0,0]
   ,points2[0,0,0],points2[1,0,0],points2[2,0,0],points2[3,0,0]])
H_max =
    np.max([points1[0,0,1],points1[1,0,1],points1[2,0,1],points1[3,0,1]
   ,points2[0,0,1],points2[1,0,1],points2[2,0,1],points2[3,0,1]])
H_min =
    np.min([points1[0,0,1],points1[1,0,1],points1[2,0,1],points1[3,0,1]
   ,points2[0,0,1],points2[1,0,1],points2[2,0,1],points2[3,0,1]])
```

Rectified points obtained above may be negative. Then displayed images may be cropped. Therefore, images are adjusted using shift_matrix so that they always have positive values. At this time, 10 is added to give a little gap.

The boundary obtained above is used to show the entire images. I gave a little gap (10 pixels) when showing the whole images. Therefore, the width of the displayed image becomes W_max-W_min+20, and the height becomes H_max-H_min+20.

The obtained rectified images are returned.

```
shift_matrix =
    np.array([[1,0,-1*W_min+10],[0,1,-1*H_min+10],[0,0,1]])
rectified_W = int(W_max-W_min+20)
```

```
rectified_H = int(H_max-H_min+20)
img1_rectified = cv2.warpPerspective(img1, h1@shift_matrix,
    (rectified_W,rectified_H))
img2_rectified = cv2.warpPerspective(img2, h2@shift_matrix,
    (rectified_W,rectified_H))
```

Code Explanation: Caculate Disparity Map.py

First, find height and width of the image.

```
H = img_left.shape[0]
W = img_left.shape[1]
```

Next, images are padded to be applied well at the corners of the images when obtaining NCC.

```
pad_num = int((window_size-1)/2)
img_left = np.pad(img_left,((pad_num, pad_num),(pad_num,
    pad_num)), 'constant', constant_values = 0)
img_right = np.pad(img_right,((pad_num, pad_num),(pad_num,
    pad_num)), 'constant', constant_values = 0)
```

Next, find the cost volume. After calculating NCC according to the disparity in each pixel, put it in cost_vol.

At this time, if disparity increases and exceeds the boundary of image, the calculation is not performed. In this case, -1 (the lowest value) is added to cost_vol so that the disparity is not selected.

```
cost_vol = np.zeros((H, W, max_disparity))

for i in range(H):
  for j in range(W):
    window_left = img_left[i:i+window_size, j:j+window_size]
    E_left = np.mean(window_left)
    window_left = window_left - E_left
    abs_left = np.sqrt(np.sum(window_left**2))
    for k in range(max_disparity):
      if j+k <= W-1:
        window_right = img_right[i:i+window_size,
            j+k:j+k+window_size]
        E_right = np.mean(window_right)
        window_right = window_right - E_right
        abs_right = np.sqrt(np.sum(window_right**2))

        left_dot_right = np.sum(np.multiply(window_left,
            window_right))
```

```
            cost_vol[i,j,k] = left_dot_right / (abs_left + 1e-8)
                / (abs_right + 1e-8)
        else:
            cost_vol[i,j,k] = -1
```

Smooth cost volume using a box filter

```
kernel = np.ones((window_size, window_size))/(window_size**2)
cost_vol = cv2.filter2D(cost_vol, -1, kernel, borderType=0)
```

Find the disparity that has the highest value in each pixel and return it.

```
disparity_map = np.argmax(cost_vol,axis=2)
```

Result: Bayer To RGB

The images captured by two cameras are as follows.

Figure 3: *Left:* Camera 1 *Right:* Camera 2

When applying bayer_to_rgb.py, results are as follows.

Figure 4: *Left:* Image 1 *Right:* Image 2

Result: Rectify Stereo Images

When applying rectify_stereo_images.py, results are as follows.

Figure 5: *Left:* Rectified Image 1 *Right:* Rectified Image 2

Result: Calculate Disparity Map

When applying calculate_disparity_map.py, results are as follows.

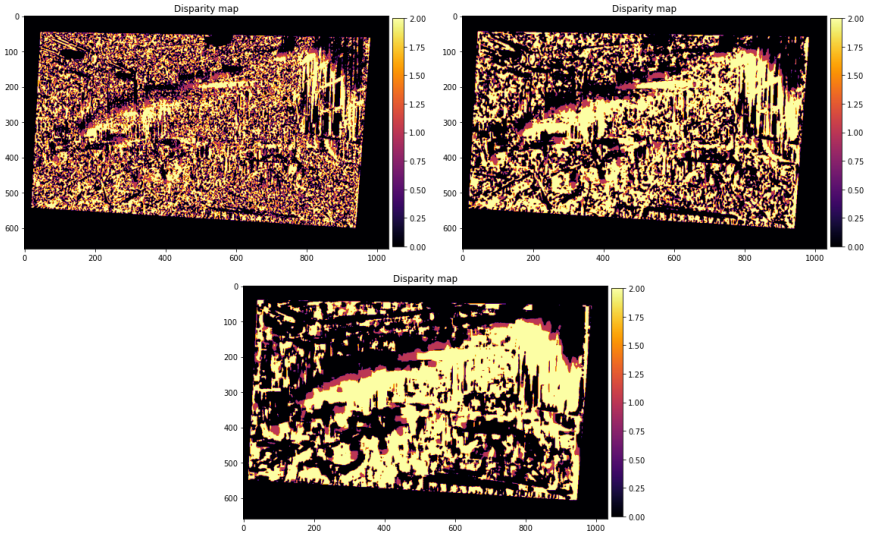

Figure 6: Max disparity is 3 and Window size is 3(Left), 7(Right), 15(Bottom)

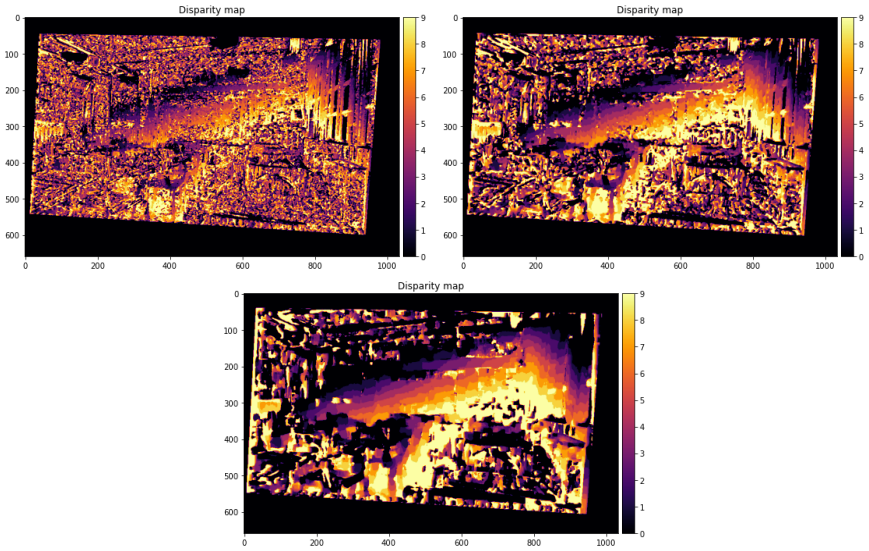

Figure 7: Max disparity is 10 and Window size is 3(Left), 7(Right), 15(Bottom)

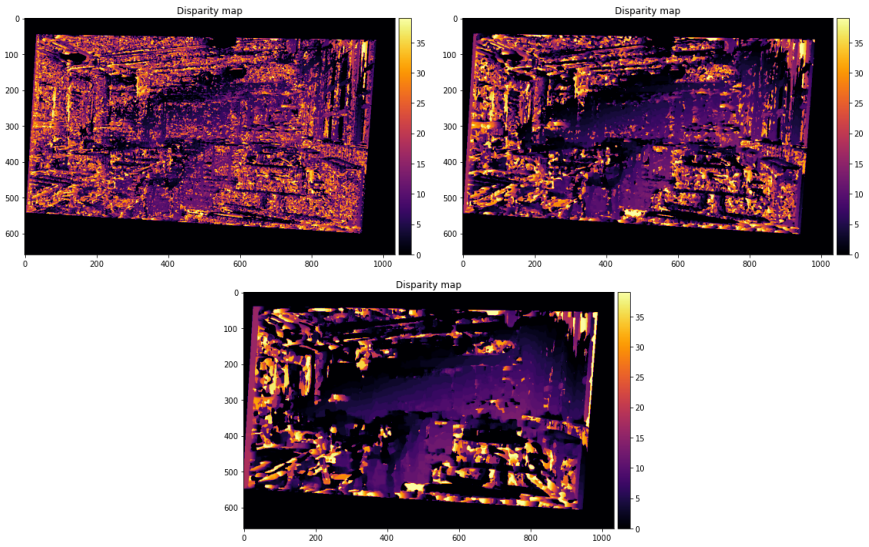

Figure 8: Max disparity is 40 and Window size is 3(Left), 7(Right), 15(Bottom)

Window Size	Max Disparity	Time (seconds)
3	3	135.42
3	10	393.38
3	40	1364.35
7	3	139.73
7	10	415.23
7	40	1385.66
15	3	138.38
15	10	413.61
15	40	1448.99

Table 1: Time to get result using processor: *Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30 GHz*, RAM: 8.00GB

If window size is small, detailed image can be obtained. And if window size is large, the approximation of image can be seen.

If max disparity is small, the time to get result can be short. But if max disparity is large, the detailed disparity map can be obtained.

Looking at the figures above, the wider the range of disparity is, the more detailed the depth of figure can be. And the larger the window size is, the less the noise is