

ECE661: Homework 9

Submission By: Joseph Wang
wang3450@purdue.edu

Prepared for Dr. Avinash Kak
GTA: Fangda Li
Purdue University
November 27, 2022

Contents

1	Projective Stereo Reconstruction	3
1.1	Methodology	3
1.2	Results	5
2	Loop and Zhang Algorithm	8
2.1	Methodology	8
2.2	Results	8
2.3	Discussion	9
3	Dense Stereo Matching	10
3.1	Methodology	10
3.2	Results	10
3.3	Discussion	11
4	Appendix	12
4.1	Image Rectification Helper Functions	12
4.2	Image Rectification Driver	16
4.3	Dense Stereo Matching Source	22

List of Figures

1	Input Image Set	5
2	Manually Selected Point Correspondences	5
3	Rectified Images	5
4	Correspondences on Rectified Images	6
5	View 1 of 3D Point Cloud	6
6	View 2 of 3D Point Cloud	6
7	3D point cloud with correspondences	7
8	Loop and Zhang Rectified Left Image	8
9	Loop and Zhang Rectified Right Image	9
10	Disparity Map $M = 10$ (left), Error Mask (right), Accuracy: 0.802141536460979 . . .	10
11	Disparity Map $M = 15$, (left) Error Mask (Right), Accuracy: 0.8366689016667682 . .	11
12	Disparity Map $M = 20$, (left) Error Mask (right), Accuracy: 0.8548739934033633 . .	11

1 Projective Stereo Reconstruction

1.1 Methodology

This section describes the necessary steps to perform projective stereo reconstruction from two stereo images using epipolar geometry. Those steps are as follows:

1. Establish at least 8 point correspondences between the left and right stereo images.
2. Perform the normalized 8-point method to compute an initial estimate of the Fundamental Matrix F:

- Transform the image coordinates to: $\hat{x}_i = Tx_i$ and $\hat{x}'_i = T'x'_i$

$$T = \begin{bmatrix} s & 0 & -s\bar{x} \\ 0 & s & -s\bar{y} \\ 0 & 0 & 1 \end{bmatrix}$$

$\bar{x}$ and $\bar{y}$: mean of x and y coordinates respectively

D : list of distances from each point to $(\bar{x}, \bar{y})$

$\bar{D}$: mean of D $s = \frac{\sqrt{2}}{\bar{D}}$

- Find $\hat{F}$ from $A\hat{F} = 0$ $A \in \mathbb{R}^{8 \times 9}$, $\hat{F} \in \mathbb{R}^9$
- Constraint Enforcement: Ensure that the rank($\hat{F}$) = 2 $\hat{F} = UDV^T$, $D = \text{diag}(r, s, t)$
 $\hat{F}' = U \text{diag}(r, s, 0) V^T$
- Denormalization: $F = T'^T \hat{F}' T$

3. Estimate the Left and Right Epipoles

$$F\vec{e} = 0$$

$$\vec{e}'^T F = 0$$

$\therefore \vec{e}$ is the right null vector and $\vec{e}'$ is the left null vector of F

4. Obtain the Initial Estimate of the Projection Matrices in Canonical Form:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}; P' = \begin{bmatrix} [e']_x & F & [e']_y \end{bmatrix}; [e']_x = \begin{bmatrix} 0 & -e'_3 & e'_2 \\ e'_3 & 0 & -e'_1 \\ -e'_2 & e'_1 & 0 \end{bmatrix}$$

5. Perform non linear Levenberg Marquardt optimization to refine the estimate of P'

- cost function minimizes: $\sum_i ||x_i - \hat{x}_i||^2 + ||x'_i - \hat{x}'_i||^2$ where $\hat{x}$ are the world points reprojected into the respective image frames
- $\hat{x}$ and $\hat{x}'$ are derived from the solution of the following system:

$$A\vec{X} = \vec{0} \rightarrow \begin{bmatrix} xP_3^T - P_1^T \\ yP_3^T - P_2^T \\ x'P_3'^T - P_1'^T \\ y'P_3'^T - P_2'^T \end{bmatrix} \begin{bmatrix} X_x \\ X_y \\ X_z \\ X_w \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A \in \mathbb{R}^{4 \times 4}; X \in \mathbb{R}^4$$

6. Rectify the Right Image:

- Compute the Translation Matrix which sends the image center to origin

$$T_1 = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Compute the Rotation Matrix which rotates the translated right epipole to $[f \ 0 \ 1]^T$

$$R = \begin{bmatrix} \cos t & -\sin t & 0 \\ \sin t & \cos t & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Compute G Matrix which sends the epipole to ∞

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix}$$

- Compute the Translational Matrix which makes the upper left corner the origin

$$T_2 = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

- The homography that rectifies the right image is: $H' = T_2 G R T_1$

7. Rectify the Left Image

- Find $M = P' P^\dagger$
Note: $P^\dagger$ is the right pseudoinverse of P
- Set $H_0 = H' M$
- Transform points in both the left and right images i.e (x, x') using H_0 and H' such that:
 $\hat{x} = H_0 x$ and $\hat{x}' = H' x'$
- Then we use the transformed coordinates to set up a distance min problem to address the issue of scaling the left image to match up to the right image as follows:

$$H_a = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

a, b, and c are determine by the min problem: $\min_{a,b,c} \sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}'_i)^2$

which simplifies to $\min_x ||Ax - b||^2$

$$A = \begin{bmatrix} \hat{x}_1 & \hat{y}_1 & 1 \\ \hat{x}_2 & \hat{y}_2 & 1 \\ \dots & \dots & \dots \end{bmatrix}$$

$$b = \begin{bmatrix} \hat{x}'_1 \\ \hat{x}'_2 \\ \dots \end{bmatrix}$$

- The homography that rectifies the left image is: $H = H_a H_0$

8. Obtain correspondences between the two rectified images by searching through the rows. Use these correspondences to perform one last optimization of F

9. Triangulate the correspondences found in step 8 to create a 3D point cloud of the scene.

1.2 Results

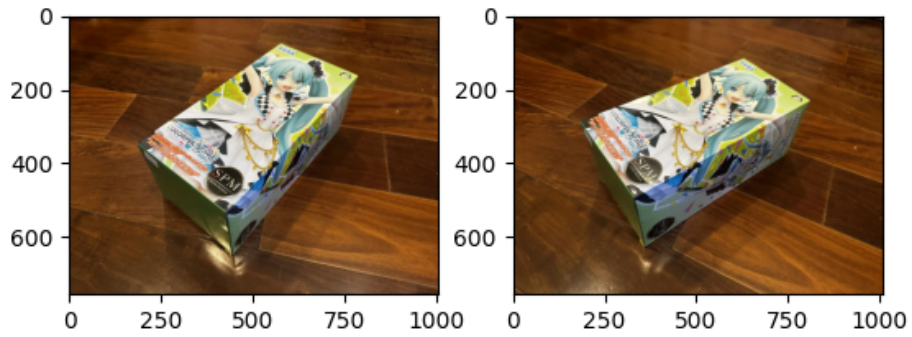


Figure 1: Input Image Set

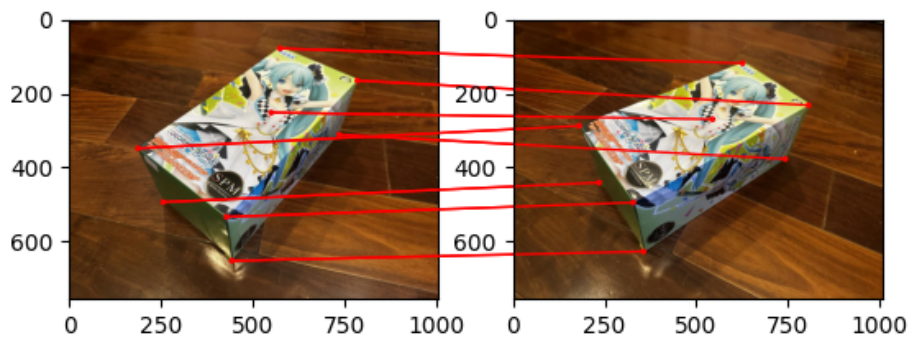


Figure 2: Manually Selected Point Correspondences

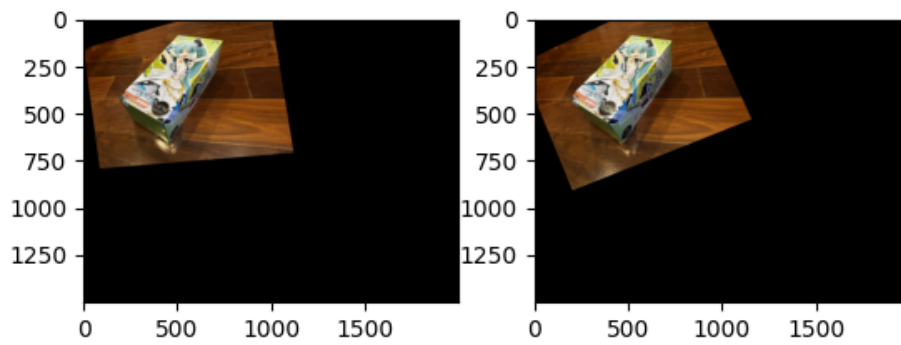


Figure 3: Rectified Images

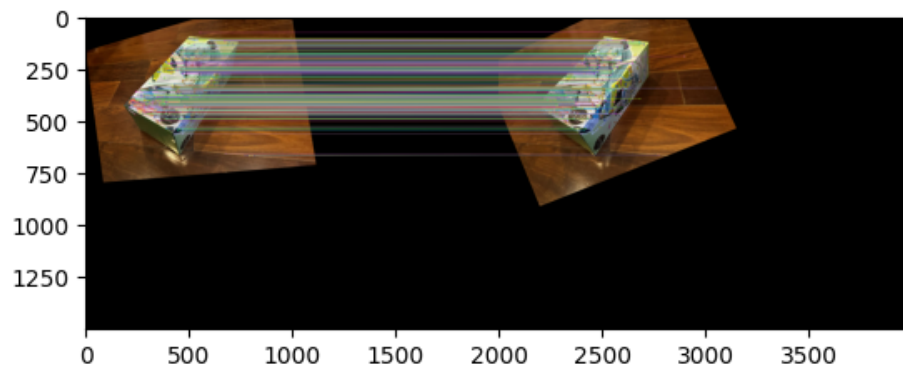


Figure 4: Correspondences on Rectified Images

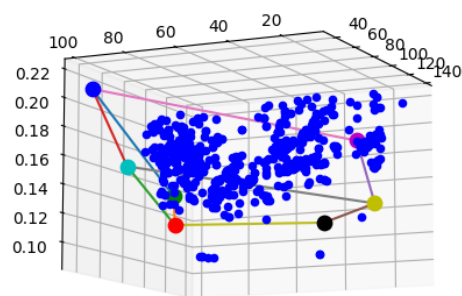


Figure 5: View 1 of 3D Point Cloud

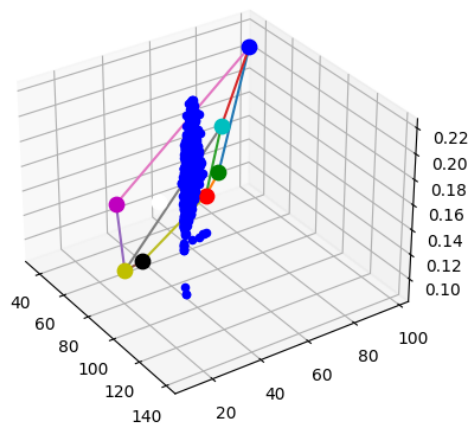


Figure 6: View 2 of 3D Point Cloud

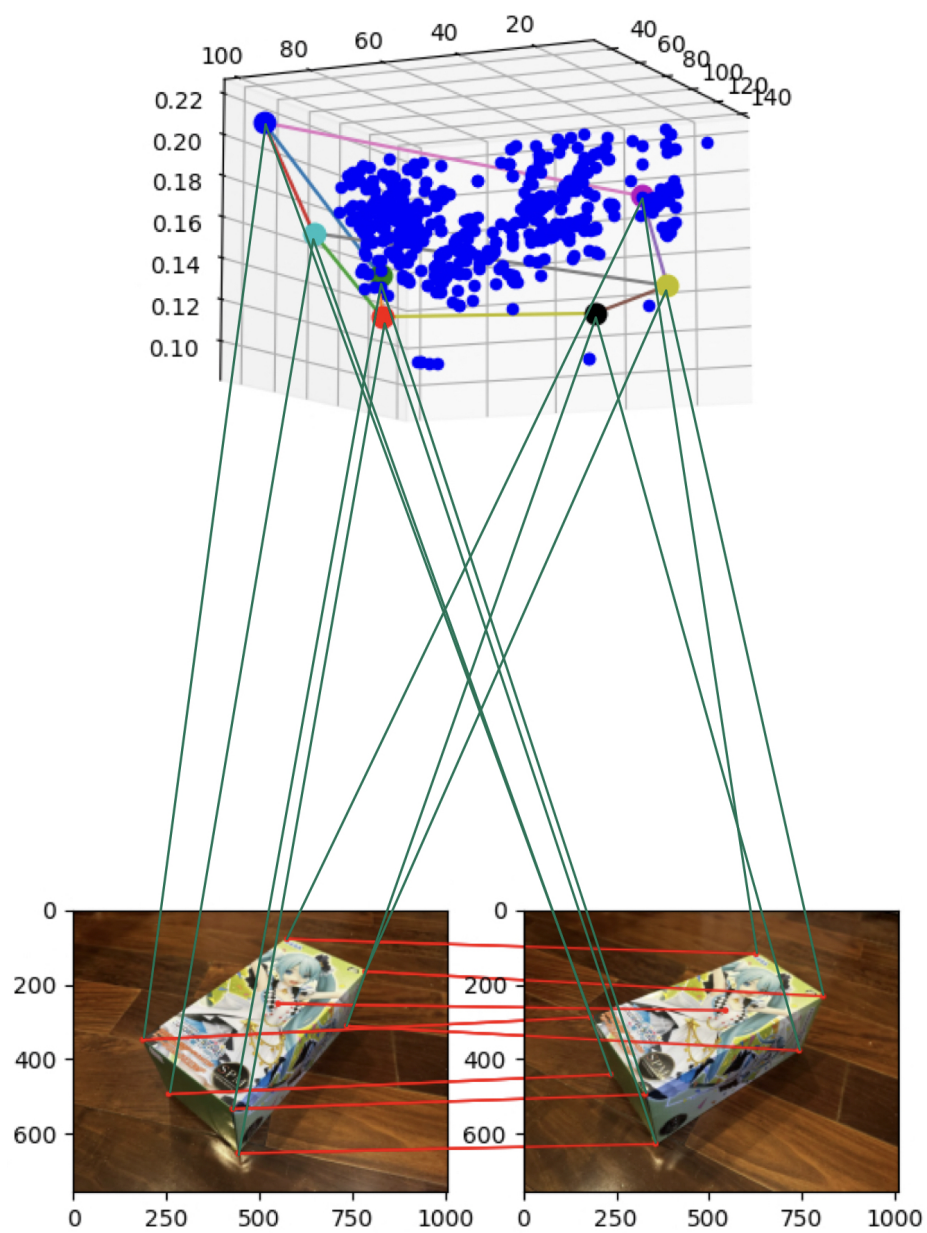


Figure 7: 3D point cloud with correspondences

2 Loop and Zhang Algorithm

2.1 Methodology

The Loop and Zhang Algorithm decomposes the rectifying homographies H and H' into the following form:

$$\begin{aligned} H &= H_{sh} H_{sim} H_p \\ H' &= H'_{sh} H'_{sim} H'_p \end{aligned}$$

H and H' are purely projective homographies whose purpose is to send the epipoles e and e' to infinity in the respective image planes. H_{sim} and H'_{sim} are similarity homographies tasked with rotating the epipoles, that are at infinity after the application of the projective homographies, so they are on the world-X axis. H_{sh} and H'_{sh} are shear homographies. They are tasked with reducing as much as possible, the purely projective distortion by introducing additional degrees of freedom into the overall rectification transformation.

2.2 Results

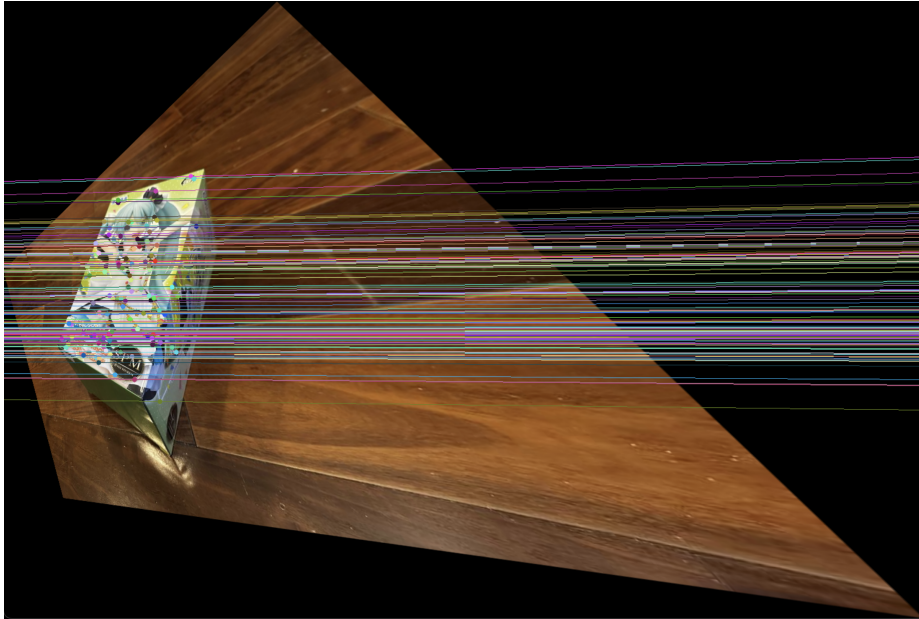


Figure 8: Loop and Zhang Rectified Left Image

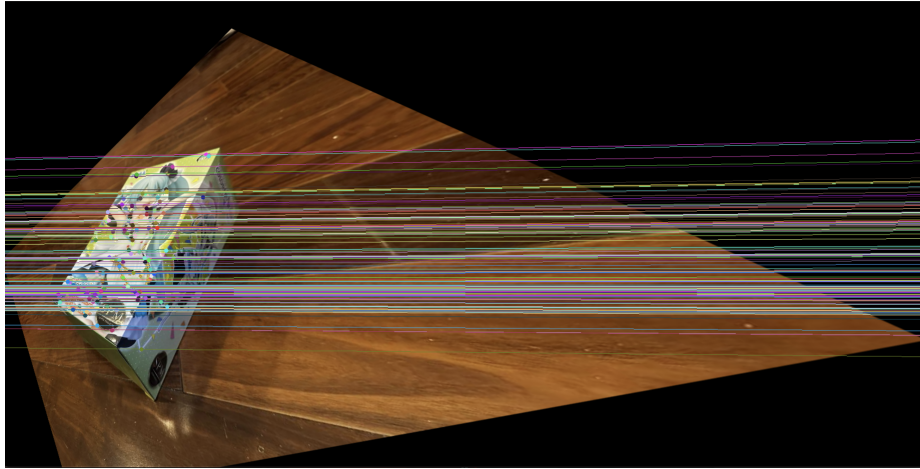


Figure 9: Loop and Zhang Rectified Right Image

2.3 Discussion

Strictly speaking on the results listed above, the Loop and Zhang algorithm seemed to produce rectified images of similar quality to that of the algorithm we implemented in section 1. Looking at both sets of rectified images, we can see that all images seem to share a similar transformation. Further examination of the correspondence plots between rectified pairs indicates that both sets of rectified images have epipoles that lie at infinity. The main difference between the two algorithms becomes apparent when supplying stereo images that are taken far apart from another. In this scenario, the Loop and Zhang significantly outperformed the "vanilla" algorithm and was able to produce far better rectified image pairs. The vanilla algorithm on the other hand would produce incomprehensible images. From a high level of understanding, these results seem plausible since Loop and Zhang takes special actions to remediate the effects of the purley projective distortions.

3 Dense Stereo Matching

3.1 Methodology

This section describes the necessary steps to perform the census transform as a rudimentary method of dense stereo matching. Those steps are as follows:

1. For each pixel in the left image, denoted (x_i, y_i) , we assign a candidate correspondence pair in the right image denoted $(x'_i - d, y'_i)$.
2. For each candidate correspondence, we consider an $M \times M$ neighborhood centered around each pixel in the correspondence.
3. For each $M \times M$ neighborhood, we compute an $M \times M$ bitvector. At each position in this bitvector, a 1 is set if the pixel in the window location is strictly larger than the center pixel. Otherwise a 0 is set. Two $M \times M$ bitvectors are expected at each iteration, one for the left image and one for the right image.
4. The resulting bitvectors are then bitwise XORed together. The number of 1's in the resulting bitvector constitutes the data cost.
5. The minima of the data cost is then found and the corresponding disparity value is set in the disparity map at the respective position.

3.2 Results

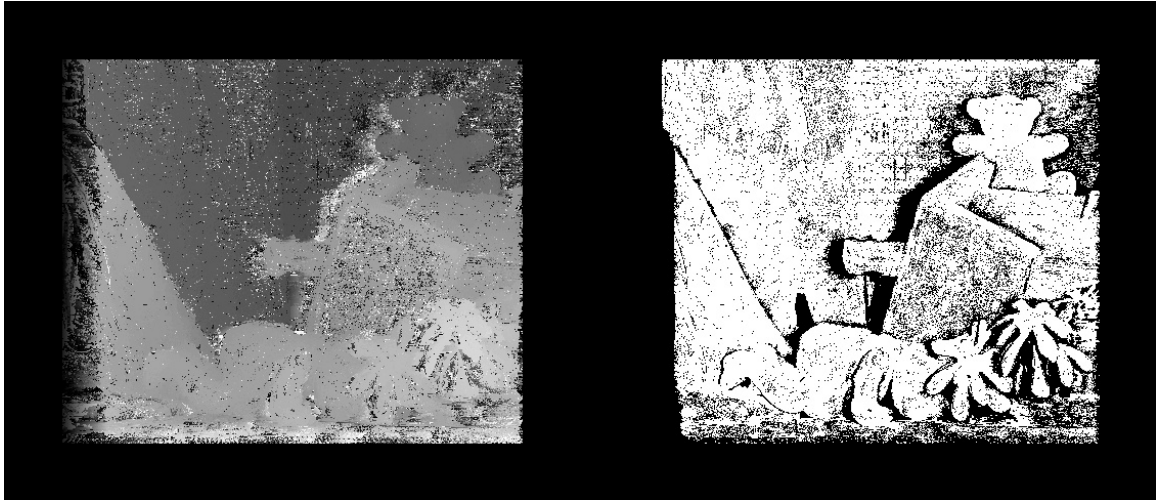


Figure 10: Disparity Map $M = 10$ (left), Error Mask (right), Accuracy: 0.802141536460979

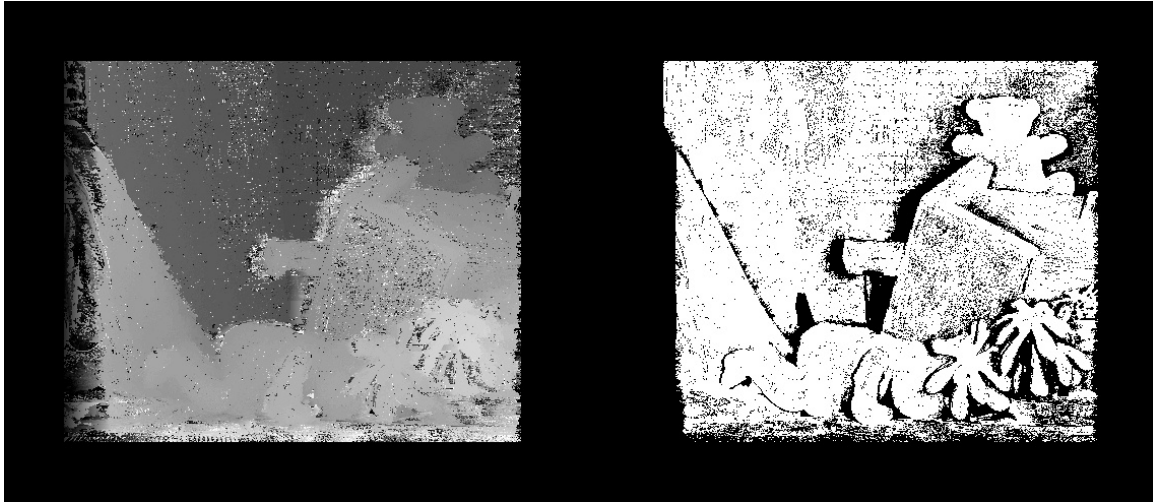


Figure 11: Disparity Map $M = 15$, (left) Error Mask (Right), Accuracy: 0.8366689016667682

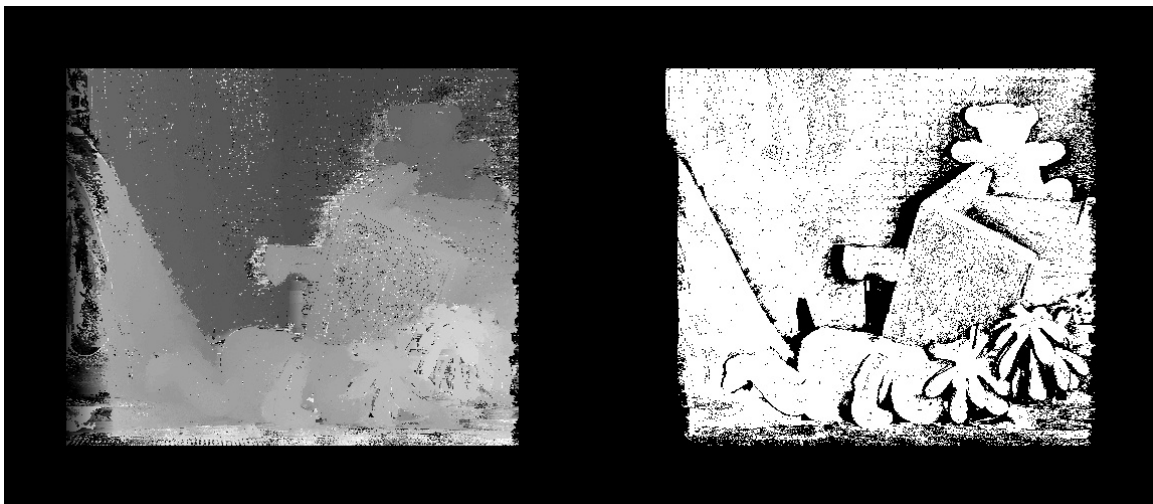


Figure 12: Disparity Map $M = 20$, (left) Error Mask (right), Accuracy: 0.8548739934033633

3.3 Discussion

In general, for relatively small window sizes, we found that the accuracy of the disparity map increased as the window size increased. However, there is a limit to this. Starting around window sizes of about 60 pixels, the accuracy begins to be inversely proportional to window size.

4 Appendix

4.1 Image Rectification Helper Functions

```

1 import numpy as np
2 from copy import deepcopy
3 from tqdm import tqdm
4 '''getDistance(x_prime, computed_x_prime)
5 Input: two points
6 Output: euclidean distance
7 Purpose: compute the distance between two points'''
8 def getDistance(x_prime, computed_x_prime):
9     x1 = x_prime[0]
10    x2 = x_prime[1]
11    x_prime1 = computed_x_prime[0]
12    x_prime2 = computed_x_prime[1]
13    distance = np.sqrt((x1 - x_prime1) ** 2 + (x2 - x_prime2) ** 2)
14    return distance
15
16
17 '''mat_mul(T, x)
18 Input: T (3x3 ndarray), x (list 2)
19 Output: list [x1/x3, x2/x3]
20 Purpose: tx'''
21 def T_mul(T: np.ndarray, x: list):
22     temp = deepcopy(x)
23     temp.append(1)
24     temp = np.array(temp).reshape((3,1))
25     x_hat = T@temp
26     assert(x_hat.shape == (3,1))
27     return [float(x_hat[0] / x_hat[2]), float(x_hat[1] / x_hat[2])]
28
29
30 '''compute_F(x, x_prime)
31 Input: list of x and x_prime points
32 Output: F (3x3 ndarray)
33 Purpose: Given 8 point corr, compute F'''
34 def compute_F(x, x_prime):
35     '''computes mean(x) mean(y) mean(d) s and T for x'''
36     x_mean = np.mean([point[0] for point in x])
37     y_mean = np.mean([point[1] for point in x])
38     d_bar = np.mean([getDistance(point, [x_mean, y_mean]) for point in x])
39     s = np.sqrt(2) / d_bar
40     T = np.array([s, 0, -s*x_mean, 0, s, -s*y_mean, 0, 0, 1]).reshape((3,3))
41
42     '''computes mean(x) mean(y) mean(d) s and T for x_prime'''
43     x_prime_mean = np.mean([point[0] for point in x_prime])
44     y_prime_mean = np.mean([point[1] for point in x_prime])
45     d_prime_bar = np.mean([getDistance(point, [x_prime_mean, y_prime_mean]) for point
46                             in x_prime])
47     s_prime = np.sqrt(2) / d_prime_bar
48     T_prime = np.array([s_prime, 0, -s_prime*x_prime_mean, 0, s_prime, -s_prime*
49                         y_prime_mean, 0, 0, 1]).reshape((3,3))
50
51     '''compute x_hat = Tx and x_prime_hat = T_prime x_prime'''
52     x_hat = [T_mul(T, point) for point in x]
53     x_prime_hat = [T_mul(T_prime, point) for point in x_prime]
54     assert(len(x_hat) == len(x_prime_hat))
55     assert(len(x_hat) == 8)

```

```

55     '''compute F_hat using linear least squares'''
56     A = np.zeros((8,9))
57     for i in range(len(x)):
58         A[i] = [x_prime[i][0] * x[i][0],
59                 x_prime[i][0] * x[i][1],
60                 x_prime[i][0],
61                 x_prime[i][1] * x[i][0],
62                 x_prime[i][1] * x[i][1],
63                 x_prime[i][1],
64                 x[i][0],
65                 x[i][1],
66                 1]
67
68     -, -, vh = np.linalg.svd(A)
69     f_hat = np.reshape(vh[-1], (3,3))
70
71     '''condition f_hat into f_prime_hat'''
72     u, d, newVh = np.linalg.svd(f_hat)
73     d[2] = 0
74     d = np.diag(d)
75     f_prime_hat = u@d@newVh
76
77     '''denormalize f_prime_hat into f'''
78     f = T_prime.T@f_prime_hat@T
79     return f/f[-1,-1]
80
81
82     '''compute_e(F)
83     Input: F (3x3 ndarray)
84     Output: e, e'
85     Purpose: Given F, compute epipoles'''
86     def compute_e(F:np.ndarray):
87         u, -, vh = np.linalg.svd(F)
88         e = np.transpose(vh[-1,:])
89         e_prime = u[:, -1]
90         e = e / e[2]
91         e_prime = e_prime / e_prime[2]
92         return e, e_prime
93
94
95     '''compute_P(e_prime, F)
96     Input: e_prime (3,) ndarray
97           F (3,3) ndarray
98     Output: P, Prime (3,4) ndarray
99     Purpose: Estimate the Projection Matrices'''
100    def compute_P(e_prime, F):
101        P = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
102        e_prime_matrix = np.array([[0, -e_prime[2], e_prime[1]], [e_prime[2], 0, -e_prime[0]], [-e_prime[1], e_prime[0], 0]])
103        e_prime = np.reshape(e_prime, (3, 1))
104        P_prime = np.hstack((e_prime_matrix@F, e_prime))
105
106        assert(np.linalg.matrix_rank(P) == 3)
107        assert(np.linalg.matrix_rank(P_prime) == 3)
108
109        return P, P_prime
110
111
112    '''triangulate(P, P_prime, x_points, x_prime)
113    Input: P (3x4 ndarray)

```

```

114         P' (3x4 ndarray)
115         x_points (list of points in left image)
116         x_prime (list of points in right image)
117 Output: World point X in physical form (3 vector)
118 Purpose: Given P, P', find world point of (x,x') '''
119 def triangulate(P, P_prime, x_points, x_prime):
120     A = np.zeros((4,4))
121     ''' get the rows of p and p' '''
122     p1 = P[0,:]
123     p2 = P[1,:]
124     p3 = P[2,:]
125     p_1_prime = P_prime[0,:]
126     p_2_prime = P_prime[1,:]
127     p_3_prime = P_prime[2,:]
128
129     '''populate the A matrix'''
130     A[0] = (x_points[0] * p3) - p1
131     A[1] = (x_points[1] * p3) - p2
132     A[2] = (x_prime[0] * p_3_prime) - p_1_prime
133     A[3] = (x_prime[1] * p_3_prime) - p_2_prime
134
135     '''X is smallest eigenvector of A^TA'''
136     -, -, vh = np.linalg.svd(A)
137     world_point = vh[-1]
138     world_point = world_point / world_point[-1]
139
140     return [float(world_point[0]), float(world_point[1]), float(world_point[2])]
141
142
143 '''world2image(p,x)
144 Input: p (3x4) projection matrix
145       x (length 3 iterable)
146 Output: len(x) = 2 , world point in physical
147 Purpose: Project world point into image space'''
148 def world2image(p, x):
149     temp = list(x)
150     temp.append(1)
151     temp = np.array(temp)
152     temp = np.reshape(temp, (4,1))
153     img_point = p@temp
154     return [float(img_point[0]/img_point[2]), float(img_point[1] / img_point[2])]
155
156
157 '''cost_function(params, x_points, x_prime_points)
158 Input: params: list of parameters
159       x_points: [[x1,y1], ... [x8,y8]]
160       x_prime_points: [[x'1,y'1], ... [x'8,y'8]]
161 Output: d^2-geom
162 Purpose: Cost Function for LM'''
163 def cost_function(params, x_points, x_prime_points):
164     '''unpacking parameter'''
165     P = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
166     P_prime = np.reshape(params[0:12], (3,4))
167     world_points = list()
168     for i in range(len(x_points)):
169         world_points.append(params[12+3*i:12+3*(i+1)])
170
171     x_hat = [world2image(P, i) for i in world_points]
172     x_prime_hat = [world2image(P_prime, i) for i in world_points]
173

```

```

174     diff_x_hat = np.subtract(x_points, x_hat)
175     diff_x_prime_hat = np.subtract(x_prime_points, x_prime_hat)
176
177     cost = np.hstack((diff_x_hat[:,0], diff_x_hat[:,1], diff_x_prime_hat[:,0],
178                       diff_x_prime_hat[:,1]))
179
180     return cost
181
182 '''def get_H_prime(img, e_prime)
183 Input: img (ndarray): right image
184       e_prime (3, ndarray): refined right epipole
185 Output: H_prime (3x3 ndarray)
186 Purpose: Compute H that rectifies right image'''
187 def get_H_prime(img, e_prime):
188     h, w, _ = img.shape
189
190     '''Compute T1'''
191     T1= np.array([[1.0, 0.0, -(w/2)],
192                  [0.0, 1.0, -(h/2)],
193                  [0.0, 0.0, 1.0]])
194
195     '''Compute T2'''
196     T2= np.array([[1.0, 0.0, (w/2)],
197                  [0.0, 1.0, (h/2)],
198                  [0.0, 0.0, 1.0]])
199
200     '''Compute R, f'''
201     x0=w/2
202     y0=h/2
203     e_prime = e_prime/e_prime[-1]
204     ex = e_prime[0]
205     ey = e_prime[1]
206     t = np.arctan(-(ey-y0)/(ex-x0)) # theta
207     R=np.array([[np.cos(t), -1*np.sin(t), 0],
208                [np.sin(t), np.cos(t), 0],
209                [0, 0, 1]])
210     f=np.abs((ex-x0)*np.cos(t)-(ey-y0)*np.sin(t))
211
212     '''Compute G'''
213     G= np.array([[1, 0, 0],
214                 [0, 1, 0],
215                 [-1/f, 0, 1]])
216
217     H_prime = T2@G@R@T1
218
219     return H_prime, f
220
221
222 '''def get_H(P, P_prime, H_prime, x_points, x_prime_points)
223 Input: P: [3x4] matrix
224       P_prime: refined p'
225       H_prime: right rectifying H
226       x_points: pts in left image
227       x_prime_points: pts in right image
228 Output: H
229 Purpose: Compute H to rectify left image'''
230 def get_H(P, P_prime, H_prime, x_points, x_prime_points):
231     '''compute pseudo inverse of P'''
232     P_pinv = P.T@(np.linalg.inv(P@P.T))

```

```

233     '''compute H_0'''
234     M = P_prime @ P_pinv
235     H_0 = H_prime @ M
236     '''Transform Points'''
237     x_hat = np.array([T.mul(H_0, i) for i in x_points])
238     x_prime_hat = np.array([T.mul(H_prime, i) for i in x_prime_points])
239
240     '''Solve For H_a'''
241     A = np.ones((x_hat.shape[0], 3))
242     A[:, :-1] = x_hat
243     b = x_prime_hat[:, 0]
244     # solve Ax=b = 0 using left Psuedo inverse
245     a = (np.linalg.inv((A.T @ A) @ A.T) @ b
246     Ha = np.eye(3)
247     Ha[0, :] = a
248     print(f'Ha: \n{Ha}')
249     H = Ha @ H_0
250     return H, H_0
251

```

image_rectification_helper.py

4.2 Image Rectification Driver

```

1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # # Image Rectification
5  # The goal of image rectification is to find the Homographies H and H' for the
6  # stereo images such that in the transformed image pair, the same world point
7  # appears on the same row.
8
9  # ### Import Statements
10
11  # In [1]:
12
13  import cv2
14  import numpy as np
15  from scipy.optimize import least_squares
16  from image_rectification_helper import *
17
18  # ### Load Images
19
20  # In [2]:
21
22  left_file_path = "/Users/wang3450/Desktop/ECE661/HW09/input_images/left.jpg"
23  right_file_path = "/Users/wang3450/Desktop/ECE661/HW09/input_images/right.jpg"
24  left_img = cv2.imread(left_file_path, 1)
25  right_img = cv2.imread(right_file_path, 1)
26  assert(left_img.shape == right_img.shape)
27
28
29
30  # ### Point Correspondences
31  # Purpose: Establish Point 8 point correspondences between left and right image
32  # * x_points (list): list of points in left image

```

```

33 # * x_points_prime (list): list of points in right image
34 # * Both lists look like this:
35 # [[x1,y1], [x2,y2], [x3,y3], ..., [x8,y8]]
36
37 # In [3]:
38
39
40 point_corr_image = np.hstack((left_img, right_img))
41 width_adj = right_img.shape[1]
42 x_points = [[722, 347], [690, 466], [820, 338], [785, 453], [284, 487], [280, 463],
43             [477, 191], [622, 185]]
44 x_prime_points = [[618, 428], [599, 531], [750, 433], [717, 537], [265, 457],
45                  [274, 430], [505, 182], [674, 184]]
46 rainbow = [(211, 0, 148), (130, 0, 75), (255, 0, 0), (0, 255, 0), (0, 255, 255), (0,
47             127, 255), (0, 0, 255), (0, 0, 0)]
48
49 for i in range(len(x_points)):
50     left_point = tuple(x_points[i])
51     right_point = (x_prime_points[i][0] + width_adj, x_prime_points[i][1])
52     cv2.circle(point_corr_image, left_point, 4, rainbow[i], -1)
53     cv2.circle(point_corr_image, right_point, 4, rainbow[i], -1)
54     cv2.line(point_corr_image, left_point, right_point, rainbow[i], 1)
55
56 # cv2.imwrite("point_correspondences.jpg", point_corr_image)
57
58 ##### Normalized 8-point Algorithm for F
59 # Purpose: Given 8 image point correspondences (x, x'), determine the fundamental
60 #           matrix F
61 #
62 # Algorithm:
63 # 1. Normalization: Transform the image coordinates to:  $\hat{x}_i = Tx_i$  and  $\hat{x}'_i = T'x'_i$ 
64 #
65 #
66 #  $T = \begin{bmatrix} s & 0 & -s\bar{x} \\ 0 & s & -s\bar{y} \\ 0 & 0 & 1 \end{bmatrix}$ 
67 #  $\bar{x}$  and  $\bar{y}$  : mean of x and y coordinates respectively
68 #  $D$  : list of distances from each point to  $(\bar{x}, \bar{y})$ 
69 #  $\bar{D}$ : mean of D
70 #  $s = \frac{\sqrt{2}}{\bar{D}}$ 
71 #
72 # 2. Find  $\hat{F}$  from  $A \hat{F} = 0$ 
73 #
74 #  $A \in \mathbb{R}^{8 \times 9}$ ,  $\hat{F} \in \mathbb{R}^9$ 
75 #
76 # 3. Constraint Enforcement: Ensure that the rank( $\hat{F}$ ) = 2
77 #
78 #  $\hat{F} = UDV^T$ ,  $D = \text{diag}(r, s, t)$ 
79 #  $\hat{F}' = U \text{diag}(r, s, 0) V'^T$ 
80 #
81 # 4. Denormalization:  $F = T'^T \hat{F}' T$ 
82
83 # In [4]:
84
85
86 F = compute_F(x_points, x_prime_points)
87 assert(np.linalg.matrix_rank(F) == 2)

```

```

88 print(f'Fundamental Matrix F: \n{F}')
89
90
91 ##### Estimate the Left and Right Epipoles
92 #  $F \vec{e} = 0$ 
93 # <br>
94 #  $\vec{e}^T F = 0$ 
95 #
96 #  $\therefore \vec{e}$  is the right null vector and  $\vec{e}'$  is the left null vector
   of  $F$ 
97 #
98
99 # In [5]:
100
101
102 e, e_prime = compute_e(F)
103 print(f'left epipole e: {e}')
104 print(f'right epipole e\': {e_prime}')
105
106
107 ##### Obtain the Initial Estimate of the Projection Matrices in Canonical Form
108 #  $P =$ 
109 #  $\begin{bmatrix}$ 
110 # 1 & 0 & 0 & 0 \\
111 # 0 & 1 & 0 & 0 \\
112 # 0 & 0 & 1 & 0 \\
113 #  $\end{bmatrix}$ 
114 # ;
115 #  $P' =$ 
116 #  $\begin{bmatrix}$ 
117 #  $\vec{e}'_x & F & \vec{e}'$  \\
118 #  $\end{bmatrix}$ 
119 # ;
120 #  $\vec{e}'_x =$ 
121 #  $\begin{bmatrix}$ 
122 # 0 &  $-\vec{e}'_3$  &  $\vec{e}'_2$  \\
123 #  $\vec{e}'_3$  & 0 &  $-\vec{e}'_1$  \\
124 #  $-\vec{e}'_2$  &  $\vec{e}'_1$  & 0 \\
125 #  $\end{bmatrix}$ 
126
127 # In [6]:
128
129
130 P, P_prime = compute_P(e_prime, F)
131 print(f'P: \n {P}\n')
132 print(f'P\': \n {P_prime}')
133
134
135 ##### Perform Triangulation to find World Points from (x,x') correspondences
136 #  $A \vec{X} = \vec{0} \rightarrow$ 
137 #  $\begin{bmatrix}$ 
138 #  $xP_3^T - P_1^T$  \\
139 #  $yP_3^T - P_2^T$  \\
140 #  $x'P'_3^T - P'_1^T$  \\
141 #  $y'P'_3^T - P'_2^T$  \\
142 #  $\end{bmatrix}$ 
143 #  $\begin{bmatrix}$ 
144 #  $X_x$  \\  $X_y$  \\  $X_z$  \\  $X_w$  \\
145 #  $\end{bmatrix} =$ 
146 #  $\begin{bmatrix}$ 

```

```

147 # 0 \ \ 0 \ \ 0 \ \ 0 \ \
148 # \end{bmatrix}$
149 #
150 # $A \in \mathbb{R}^{4 \times 4}$ ; $X \in \mathbb{R}^4$
151 #
152 # $X$ is given by the smallest eigenvector of $A^T A$
153 #
154 # In [7]:
155 #
156
157 world_points = [triangulate(P, P_prime, x_points[i], x_prime_points[i]) for i in
158                  range(len(x_points))]
159 for i, wp in enumerate(world_points):
160     print(f'World Point {i+1}: {wp}')
161
162 ### Refinement of P'
163 # * params contains p' and every 3-vector world point
164 # * cost function minimizes: $\sum_i ||x_i - \hat{x}_i||^2 + ||x'_i - \hat{x}'_i||$
165   where $\hat{x}$ are the world points reprojected into the respective image frames
166 #
167 # In [8]:
168 #
169 '''pack p' into params'''
170 params = np.zeros(12 + (3 * len(x_points)))
171 params[0:12] = np.reshape(P_prime, 12)
172 params[12:] = np.reshape(world_points, (3 * len(x_points)))
173 #
174 #
175 # In [9]:
176 #
177
178 optim = least_squares(cost_function, params, args=(x_points, x_prime_points), verbose
179               =1)
180 params_star = optim['x']
181 #
182 # In [10]:
183 #
184
185 P_prime_ref = params_star[0:12]
186 P_prime_ref = np.reshape(P_prime_ref, (3,4))
187
188 t_ref = P_prime_ref[:, -1]
189 t_ref_matrix = np.array([[0, -t_ref[2], t_ref[1]], [t_ref[2], 0, -t_ref[0]], [-t_ref
190               [1], t_ref[0], 0]])
191 F_ref = t_ref_matrix @ P_prime[:, 0:3]
192 F_ref = F_ref / F_ref[-1, -1]
193 e_prime_ref = compute_e(F_ref)
194 print(f'Refined F: \n{F_ref}')
195 print(f'P: \n{P}')
196 print(f'Refined P \n{P_prime_ref}')
197 print(f'Refined e \n{e_prime_ref}')
198 #
199 ### Rectifying the Right Image
200 # 1. Compute the Translation Matrix which sends the image center to origin
201 #
202 # $T_1 = \begin{bmatrix}
```

```

203 #      1 & 0 & -x_0\\
204 #      0 & 1 & -y_0\\
205 #      0 & 0 & 1\\
206 #      \end{bmatrix}$
207 #
208 #      $(x_0, y_0)$ is the center of the right image
209 # 2. Compute the Rotation Matrix which rotates the translated right epipole to $\begin{bmatrix} f & 0 & 1 \end{bmatrix}^T$
210 #
211 #      $R = \begin{bmatrix} \cos\{t\} & -\sin\{t\} & 0 \\ \sin\{t\} & \cos\{t\} & 0 \\ 0 & 0 & 1 \end{bmatrix}$
212 #
213 #      $R = \begin{bmatrix} \cos\{t\} & -\sin\{t\} & 0 \\ \sin\{t\} & \cos\{t\} & 0 \\ 0 & 0 & 1 \end{bmatrix}$
214 #
215 #      $R = \begin{bmatrix} \cos\{t\} & -\sin\{t\} & 0 \\ \sin\{t\} & \cos\{t\} & 0 \\ 0 & 0 & 1 \end{bmatrix}$
216 #
217 # 3. Compute G Matrix which sends the epipole to $\infty$
218 #
219 #      $G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix}$
220 #
221 #      $G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix}$
222 #
223 #      $G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{bmatrix}$
224 #
225 # 4. Compute the Translational Matrix which makes the upper left corner the origin
226 #
227 #      $T_2 = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$
228 #
229 #      $T_2 = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$
230 #
231 #      $T_2 = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$
232 #
233 #      $(x_0, y_0)$ is the center of the right image
234 #
235 # 5. The homography that rectifies the right image is: $H' = T_2 G R T$
236 #
237 # In [11]:
238 #
239 #
240 H_prime, f = get_H_prime(right_img, e_prime.ref)
241 H_prime = H_prime / H_prime[-1,-1]
242 print(f'H\': \n {H_prime}')
243 print(f'f: \n {f}')
244 #
245 #
246 # ### Rectifying the Left Image
247 # 1. Find $M = P'P^{\dagger}$
248 #      Note: $P^{\dagger}$ is the right pseudoinverse of P
249 #
250 # 2. Set $H_0 = H'M$
251 # 3. Transform points in both the left and right images i.e (x, x') using $H_0$ and $H'$ such that:
252 #      $\hat{x} = H_0 x$
253 #      $\hat{x}' = H' x'$
254 # 4. Then we use the transformed coordinates to set up a distance min problem to address the issue of scaling the left image to match up to the right image as follows:
255 #
256 #      $H_a = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \end{bmatrix}$
257 #
258 #      $H_a = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \end{bmatrix}$

```

```

259 #      0 & 0 & 1\\
260 #      \end{bmatrix}$
261 #
262 #      a, b, and c are determine by the min problem:
263 #      $\min_{\{a,b,c\}}\sum_i(a \hat{x}_i + b \hat{y}_i + c - \hat{x}'_i)^2$
264 #
265 #      which simplifies to $\min_x ||Ax-b||^2$
266 #
267 #      $A = \begin{bmatrix}
268 #      \hat{x}_1 & \hat{y}_1 & 1\\
269 #      \hat{x}_2 & \hat{y}_2 & 1\\
270 #      \dots & \dots & \dots
271 #      \end{bmatrix}$
272 #
273 #      $b = \begin{bmatrix}
274 #      \hat{x}'_1\\
275 #      \hat{x}'_2\\
276 #      \dots
277 #      \end{bmatrix}$
278 #
279 # 5. The homography that rectifies the left image is: $H = H_a H_0$
280
281 # In [12]:
282
283
284 H, H_0 = get_H(P, P_prime_ref, H_prime, x_points, x_prime_points)
285 H = H / H[-1][-1]
286 H_0 = H_0 / H_0[-1][-1]
287
288 print(f'H: \n{H} ')
289 print(f'H_0: \n{H_0} ')
290
291
292 # In [17]:
293
294
295 w, h = 2500, 2000
296 from matplotlib import pyplot as plt
297 def rectification_plot (image1,image2,H, Hp, w, h, save_img_path):
298     rectified1 = cv2.warpPerspective(image1, Hp, (w, h))
299     rectified2 = cv2.warpPerspective(image2, (Hp), (w, h))
300     plt.figure()
301     plt.subplot(1, 2, 1)
302     plt.imshow(cv2.cvtColor(rectified1, cv2.COLOR_BGR2RGB))
303     plt.subplot(1, 2, 2)
304     plt.imshow(cv2.cvtColor(rectified2, cv2.COLOR_BGR2RGB))
305     plt.savefig(save_img_path,dpi=1200)
306     return rectified1, rectified2
307
308 # H=np.array([[ -7.13781513e+00,  2.86260684e+01,  2.10576556e+02],
309 # [-1.28237139e+01,  5.16282910e+01,  3.74581045e+02],
310 # [-3.40561613e-02,  1.36831769e-01,  1.00000000e+00]])
311 # print(H)
312 rec1, rec2 = rectification_plot(left_img, right_img, H, H_prime, w, h, '/Users/
313     wang3450/Desktop/ECE661/test_hw09/test.jpg')
314
315 # In [14]:
316
317

```

```

318 # test = transformInputImage(left_img , H_prime)
319 # cv2.imshow('test', test)
320 # cv2.waitKey(0)
321 # cv2.destroyAllWindows()

```

image_rectification.py

4.3 Dense Stereo Matching Source

```

1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # In [14]:
5
6
7  import numpy as np
8  import cv2
9  from tqdm import tqdm
10
11
12 # In [15]:
13
14
15 def census(left_img , right_img , M, dMax) :
16     win_half = int(M / 2 )
17     border = dMax + win_half
18
19     '''pad the images to prevent data loss'''
20     imgL = np.pad(left_img , border)
21     imgR = np.pad(right_img , border)
22     width = imgL.shape[1]
23     height = imgL.shape[0]
24     dMap = np.zeros_like(imgL)
25
26     '''double for loop to raster scan image'''
27     for left_row in tqdm(range(border , height - border)):
28         for left_col in range(width - border - 1, border - 1, -1):
29             costVec = []
30             windowL = imgL[left_row - win_half : left_row + win_half + 1,
31                             left_col - win_half : left_col + win_half + 1]
32             binWinL = np.ravel((windowL > imgL [left_row , left_col]) * 1)
33
34             '''searching through disparity values to identify minima'''
35             for d in range(dMax + 1):
36                 rowR = left_row
37                 colR = left_col - d
38                 windowR = imgR[rowR - win_half : rowR + win_half + 1 ,
39                                 colR - win_half : colR + win_half + 1]
40                 binWinR = np.ravel((windowR > imgR[rowR , colR]) * 1)
41                 costVec.append(sum(binWinL ^ binWinR))
42             dMap[left_row , left_col] = np.argmin(costVec)
43     return dMap.astype(np.uint8)
44
45 # In [16]:
46
47 # Read in the input images

```

```
48 left_image = cv2.imread('/Users/wang3450/Desktop/ECE661/HW09/input_images/im2.png',0)
49 right_image = cv2.imread('/Users/wang3450/Desktop/ECE661/HW09/input_images/im6.png',0)
50
51 # Hyper Parameters
52 M = 20
53 dmax = 52
54
55 # apply census transform
56 dmap = census(left_image, right_image, M, dmax)
57 dmap_viewable = (dmap / np.max(dmap) * 255).astype(np.uint8)
58
59
60 # In [17]:
61
62
63 dmap_gt = cv2.imread('/Users/wang3450/Desktop/ECE661/HW09/input_images/disp2.png')
64 dmap_gt = cv2.cvtColor(dmap_gt, cv2.COLOR_BGR2GRAY)
65 dmap_gt = dmap_gt.astype(np.float32) / 4
66 dmap_gt = dmap_gt.astype(np.uint8)
67
68 # compute the error
69 dmap_gt = np.pad(dmap_gt, (52 + (int(M/2))))
70 error = abs(dmap.astype(np.int16) - dmap_gt.astype(np.int16)).astype(np.uint8)
71 print(f'Error Min/Max: {np.min(error)}, {np.max(error)} ')
72
73 c1 = (error <= 2) * 255
74 c1 = c1.astype(np.uint8)
75
76 validMask = cv2.imread('/Users/wang3450/Desktop/ECE661/HW09/input_images/occl.png',
77 0)
78 validMask = np.pad(validMask, (52 + (int(M/2))))
79 N = cv2.countNonZero(validMask)
80
81 validError = cv2.bitwise_and(validMask, c1)
82
83 d2Error = cv2.countNonZero(validError) / N
84
85 submit_image = np.hstack((dmap_viewable, validError))
86 cv2.imwrite('/Users/wang3450/Desktop/ECE661/HW09/results/task3_results/submit_image-'
87 + str(dmax) + "-" + str(M) + '.jpg', submit_image)
```

disparity.py