ECE661 Fall 2024: Homework 9

By – Aayush Bhadani

abhadani@purdue.edu

The objective of this assignment involves:

- Projective Stereo Reconstruction
- Dense Stereo Matching
- Depth Map and Automatic Extraction of Dense Correspondences

Theory Question:

For a given pixel $\vec{x}$ in the left camera there exists a point in 3D which we can obtain with backprojection as $P^+\vec{x}$ where $P^+ = P^T(PP^T)^{-1}$. The image of this world point in the right camera will be $P'P^+\vec{x}$. If this point lies on epipolar line $\vec{l}'$, we can get the equation of $\vec{l}'$ as,

$\vec{l}' = e' \times P'P^+\vec{x} = [e']_X P'P^+\vec{x} \equiv F\vec{x}$ where $F = [e']_X P'P^+$.

From fundamental relationship between the pixels $\vec{x}$ and $\vec{x}'$ in the two cameras for the same scene point X we have ${\vec{x}'}^T F\vec{x} = 0$, and replacing $F\vec{x}$ with $\vec{l}'$ in the fundamental relationship we get

${\vec{x}'}^T\vec{l}' = 0$. This means $\vec{x}'$ is on line $\vec{l}'$ which supports our assumption.

Thus, given a pixel $\vec{x}$ in the left camera, its corresponding pixel $\vec{x}'$ in the right camera will lie on the epipolar line $\vec{l}'$.

Task 1: Projective Stereo Reconstruction

1. **Image Rectification**

   The goal here is to find homographies H and H' for the stereo images.
   The steps are as follows:

   - Manually pick 8 point correspondences between left and right camera images.
   - Using 8 point algorithm, calculate linear estimate of fundamental matrix F_{3X3}.

     $$\vec{x}F\vec{x}' = 0$$

     For a single point correspondence between $\vec{x} = (x, y, 1)$ and $\vec{x}' = (x', y', 1)$

     We get a row in matrix $A \epsilon \mathbb{R}^{8X9}$, which is $[xx'\ x'y\ x'\ xy'\ yy'\ y'\ x\ y\ 1]$

     With 8 such correspondences, we can construct A_{8X9} matrix and then calculate the unknown vector f_{9X1} using linear least squares based on $Af = 0$. With the estimate of vector f_{9X1} we can construct our fundamental matrix F.

     We will however have to condition this fundamental matrix F to be rank 2, which we achieve by setting smallest eigen value to 0.

- Estimate left and right epipoles (e, e')

  e is the null vector of F where as e' is the null vector of F^T. We can obtain these with SVD.

- Get initial estimates of the cameras $P = [I|0]$ and $P' = [[e']_X F|e']$

  Here $[e']_X = \begin{bmatrix} 0 & -e_3' & e_2' \\ e_3' & 0 & -e_1' \\ -e_2' & e_1' & 0 \end{bmatrix}$

  Using P, P' and manually annotated correspondences, we can calculate the initial estimate for corresponding world points using triangulation.

  For a given $\vec{x}, \vec{x}'$ we can derive the corresponding world point $\vec{X}$ by solving:

  $$A\vec{X} = \vec{0}$$

  i.e $$\begin{bmatrix} xP_3^T - P_1^T \\ yP_3^T - P_2^T \\ x'P'^T_3 - P'^T_1 \\ y'P'^T_3 - P'^T_2 \end{bmatrix}_{4X4} \overrightarrow{X_{4X1}} = \vec{0}$$

- Refine the initial estimate of right camera matrix P' using non-linear least squares.

  Cost function uses geometric error which will involve triangulating image points back to their world coordinates , followed by projections back to the image planes. We use $3N + 12$ parameters for this optimization which consists of 12 elements of P' matrix and N world points ($X_{i \in N}$) having 3 dim each.

- Obtain refined fundamental matrix F using refined projection matrix P'using $F = [e']_X P' P^+$; We get e' from last column of refined P' since $P' = [[e']_X F|e']$ Also get refined epipoles from refined F.

- We now compute homographies for image rectification.

  For right image we compute $H' = T_2 G R T$

  $$T = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}, (x_0, y_0) \text{ is image center location}$$

  $$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and obeys } R \begin{bmatrix} (e_x' - x_0) \\ (e_y' - y_0) \\ 1 \end{bmatrix} = \begin{bmatrix} f \\ 0 \\ 1 \end{bmatrix} \text{ and gives } \theta, f.$$

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

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

For left image we compute $H = H_A H_0$

$H_0 = H'M$ where $M = P'P^+$

$H_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, where we solve for a, b, c with the objective of

Minimizing $\sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}'_i)^2$ where $\hat{x}_i, \hat{y}_i$ projection in left camera image plane and $\hat{x}'_i$ is in the right one. With N points we form a matrix equation from this minimization objective and solve for a,b,c.

- Finally, we apply these homographies to our stereo pair to rectify the images.

2. <u>Interest Point Detection</u>

- Use Canny detector to extract edge features from the rectified views which we use as interest points.
- Find best matching point in the right image corresponding to every interest point in the left image. We use SSD metric to select best candidate for each pair of correspondences.

$$SSD = \sum_i \cdot \sum_j |f_1(i,j) - f_2(i,j)|^2$$

Since we are working with rectified views, our search space for finding a good match can be reduced significantly. We just need to check around the same row in the corresponding image.

- Convert the above matching coordinates of rectified view to original image coordinates using inverse of rectification homographies.

3. <u>Projective Reconstruction</u>

- Use large number of correspondences obtained from previous step and refined cameras to obtain world coordinates ($\vec{X}$) using triangulation.
- To improve scene reconstruction, perform non-linear optimization to refine the estimates for world coordinates. For this optimization, we will have 3N parameters corresponding to N points. We use linear least squares solution from above step as initial solution for the refinement. The cost function will be based on reprojection error like the one done earlier.

4. <u>Projective Reconstruction</u>

- Finally, make 3D plot of the reconstructed points. Also draw pointer lines between corresponding pixels in the two images and with reconstructed 3D points.
- Useful for plotting - proj3d.proj_transform, ConnectionPatch (matplotlib's)

RESULTS

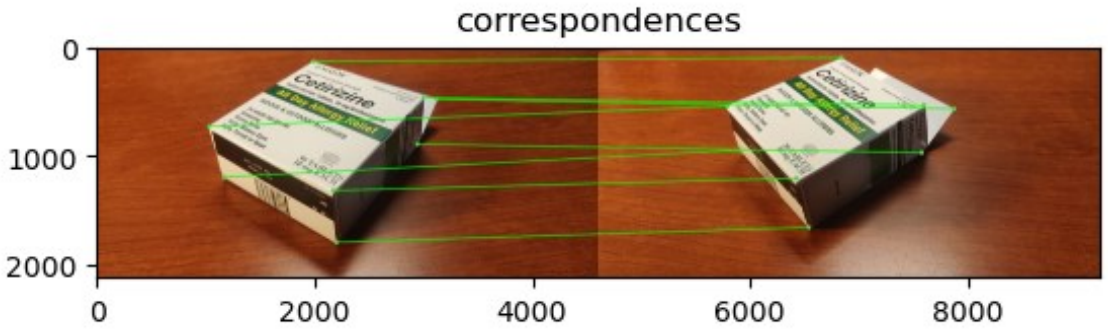
Manually selected points:

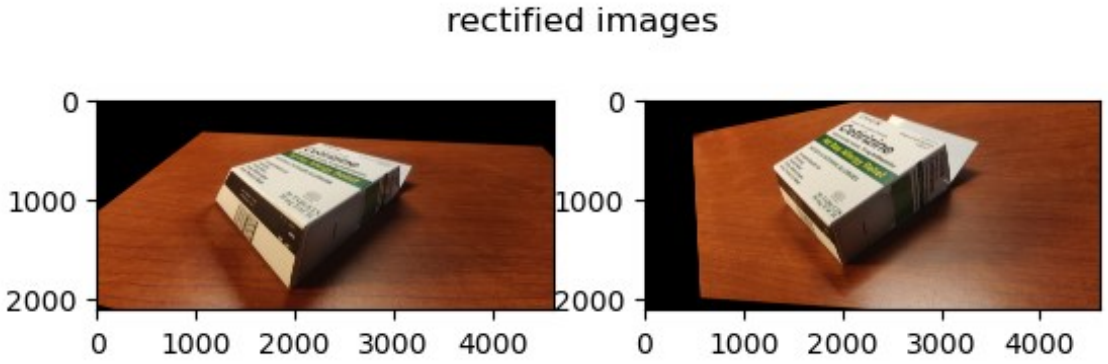
image1(left):
[(1996,141),(1034,736),(2177,1321),(3002,473),(1171,1199),(2206,1794),(2948,897),(3128,487)]

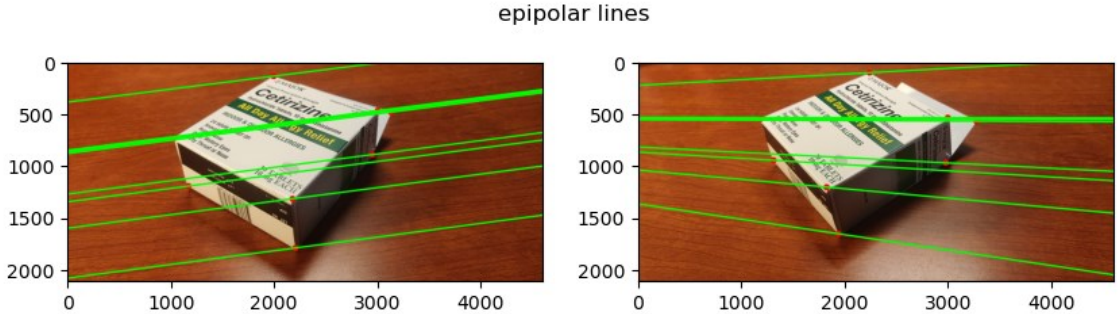
image2(right):
[(2235,107),(1186,551),(1820,1209),(2992,541),(1308,951),(1937,1658),(2972,975),(3270,575)]

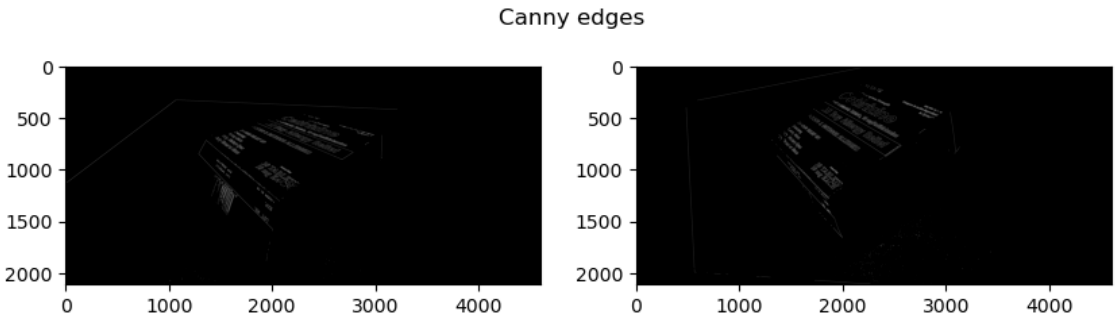
image1 (left image)

image2 (right image)

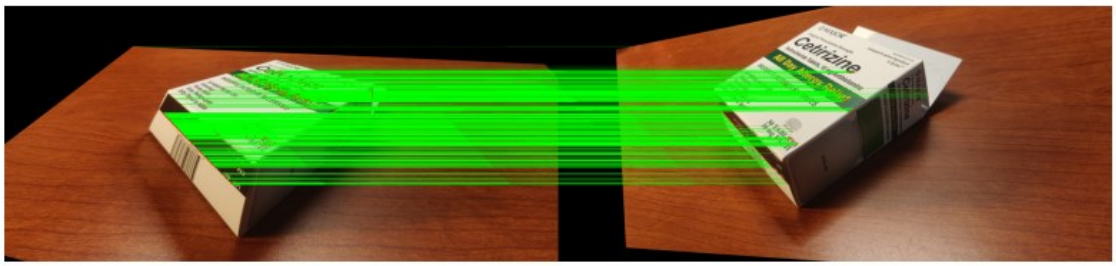




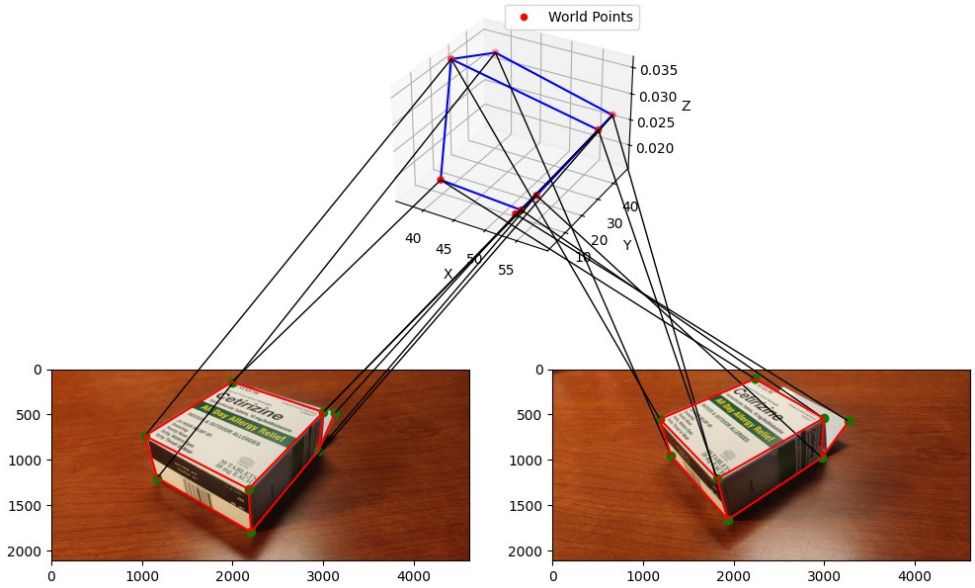


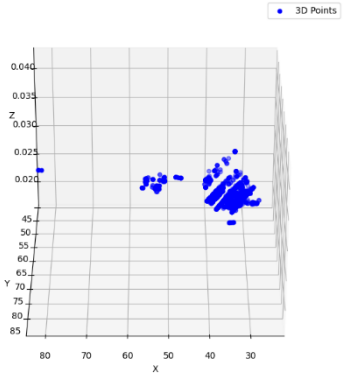


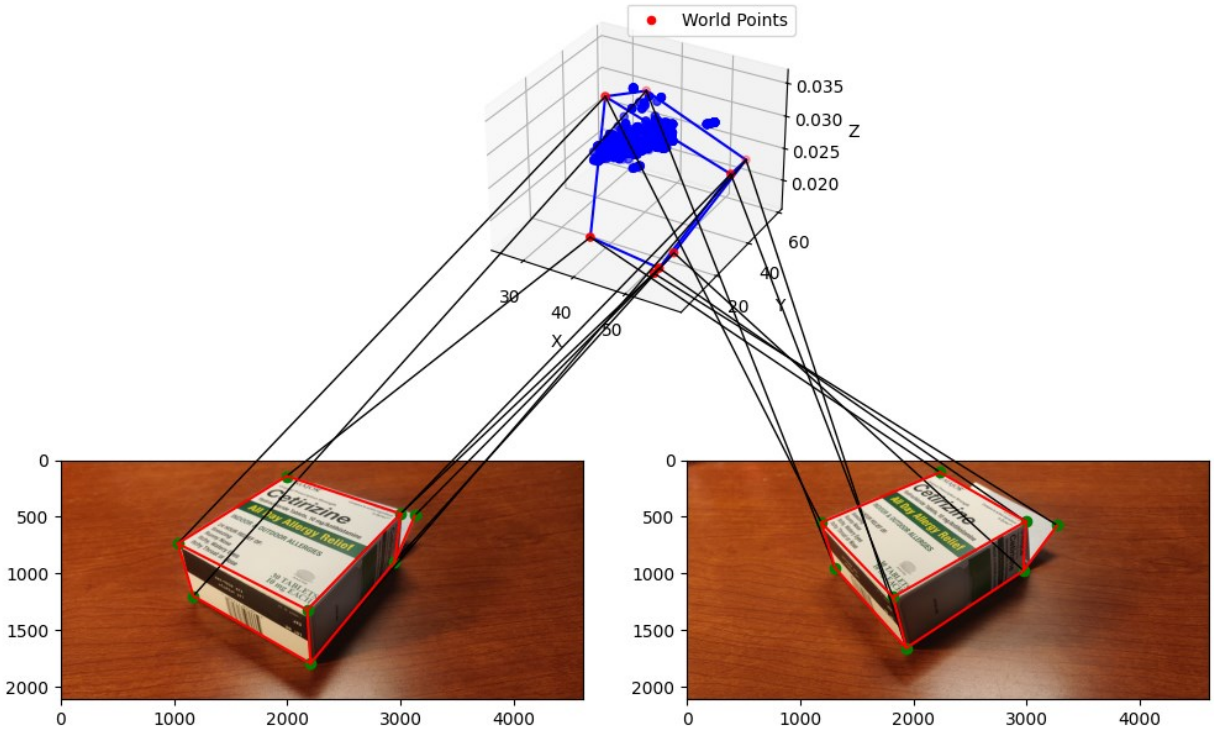
Large set of correspondences based on the Canny points on the rectified pair:

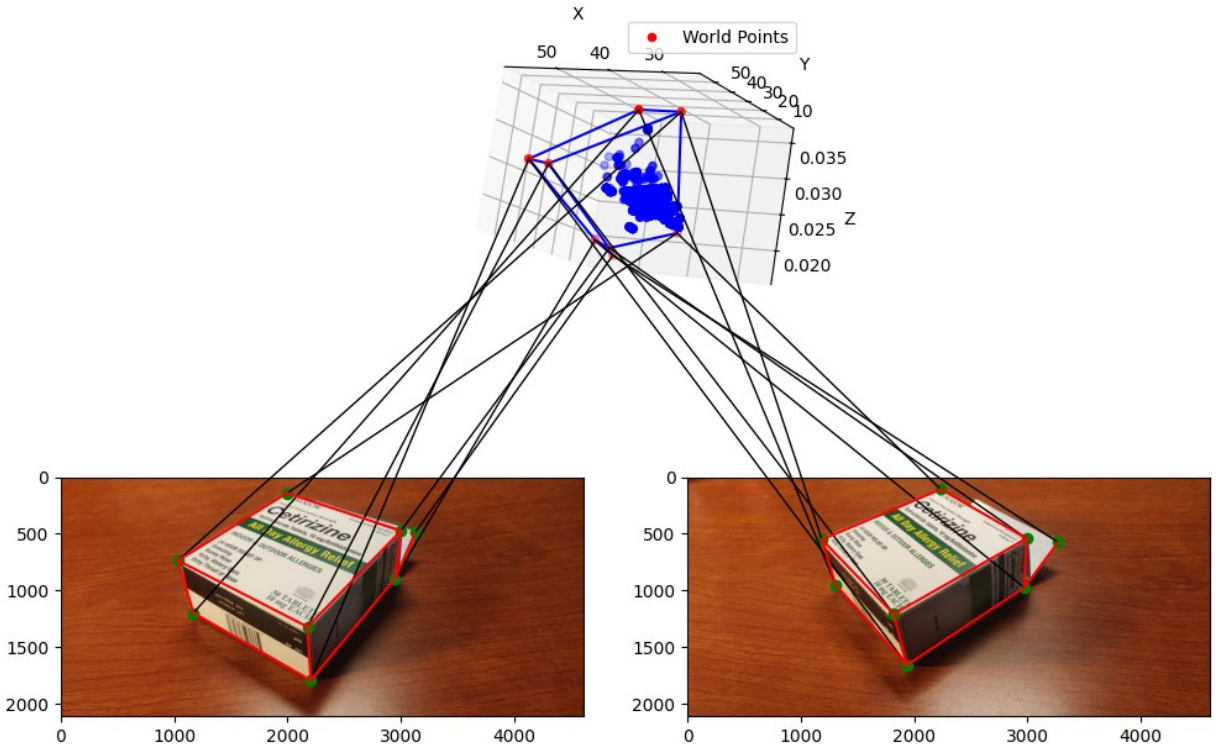


3D plot with selected correspondences:






World Points
0.035
0.030
0.025
0.020
Z
60
40
20
Y
30
40
50
X
0
500
1000
1500
2000
0
1000
2000
3000
4000
Cetirizine
All Day Allergy Relief


X
50
40
30
World Points
50
40
30
20
10
Y
0.035
0.030
0.025
0.020
Z
0
500
1000
1500
2000
0
1000
2000
3000
4000
Cetirizine
All Day Allergy Relief

Task 2: The Loop and Zhang Algorithm

Loop and Zhang algorithm used for image rectification effectively boils down to decomposing the rectifying homographies H, H' as follows:

$$H = H_{sh}H_{sim}H_p$$

$$H' = H'_{sh}H'_{sim}H'_p$$

H_p and H'_p are purely projective homographies that send the epipoles e and e' to infinity in the respective image planes. H_{sim} and H'_{sim} are similarity homographies which can only rotate, translate, and uniformly scale an image. Apart from the scale change, they cannot distort the image. Thus these two similarity homographies work to rotate the epipoles, which are at infinity post-projective homographies, so they are on the world-X axis. H_{sh} and H'_{sh} are shearing homographies with which we hope to reduce this purely projective distortion by introducing additional degrees of freedom into the overall rectification transformation.

RESULTS

Image1 (left)

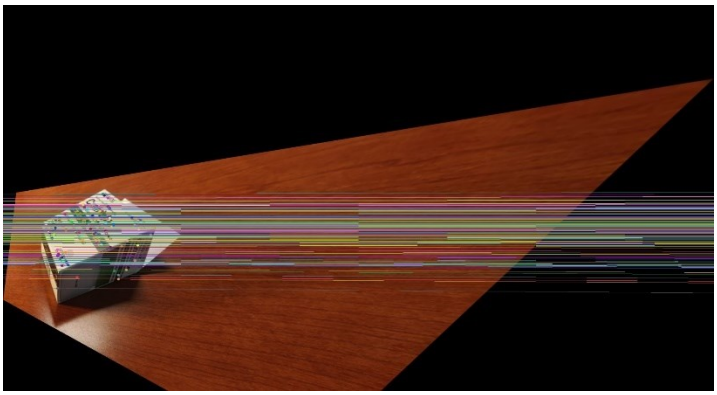
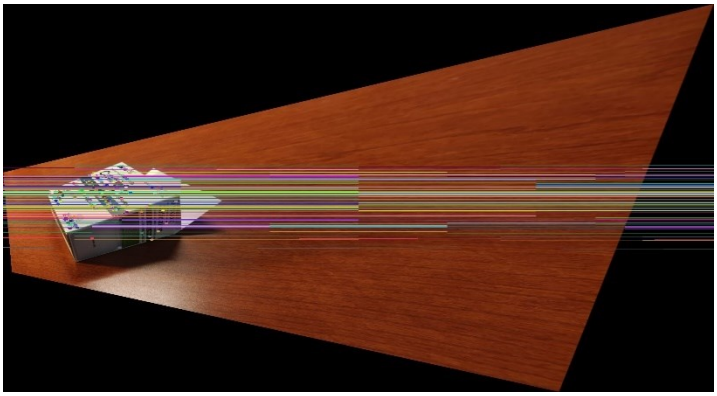


Image2 (right)

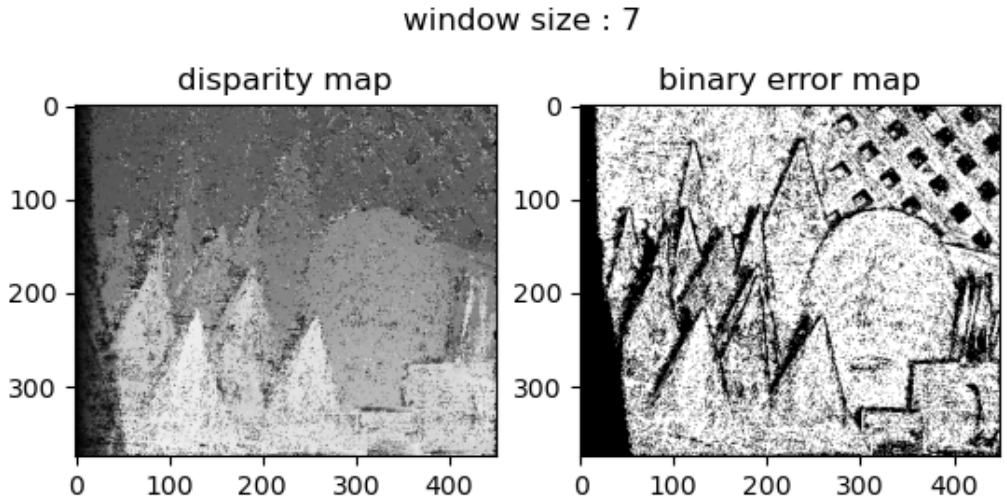


While both methods effectively rectified the images, Loop and Zhang's method seems to produce somewhat better quality of rectification results especially when the difference in the left and right camera images is more. This can be attributed to the fact that Loop and Zhang's algorithm also tries to correct for distortion, ensuring that rectified images are free of purely projective distortion. The former method can be sensitive to errors in estimating the fundamental matrix based on initially selected points, which may lead to poor rectification or misalignment of epipolar lines. On the other hand Loop and Zhang's method has better robustness to noise in point correspondences.

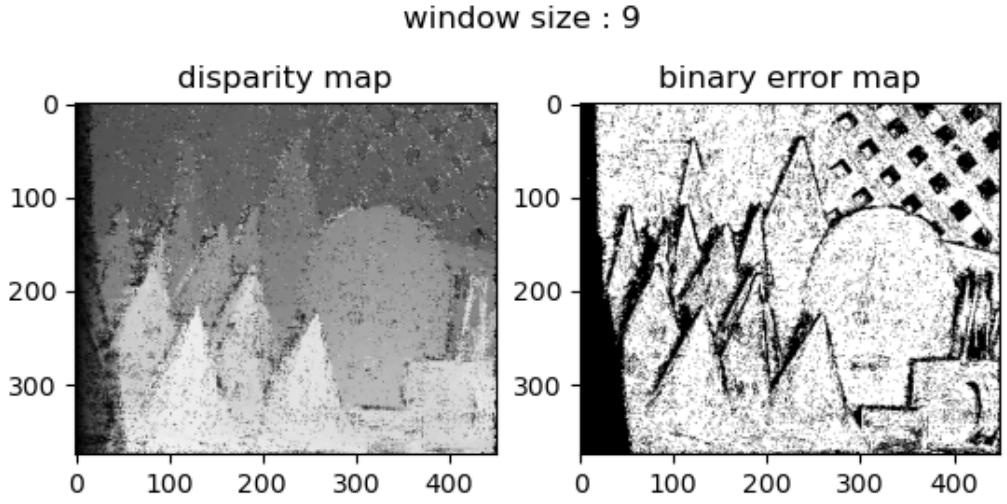
Task 3: Dense Stereo Matching

Dense stereo matching using the Census Transform involves computing a disparity map by finding pixel correspondences between a rectified stereo image pair. For each pixel in the left image, the algorithm searches along the same row in the right image within a range of disparities. The Census Transform encodes the local intensity pattern around each pixel into a binary bitvector by comparing each pixel in an $M \times M$ window to the center pixel and assigning a 1 if the pixel is brighter and 0 otherwise. For each potential match, the similarity is measured by performing a bitwise XOR on the Census bitvectors of the left and right pixels and counting the number of 1s which forms the data cost. The disparity that minimizes the data cost is assigned to the pixel, thus producing the disparity map.

RESULTS



Accuracy: 0.6501133431403284



Accuracy: 0.70299304670019

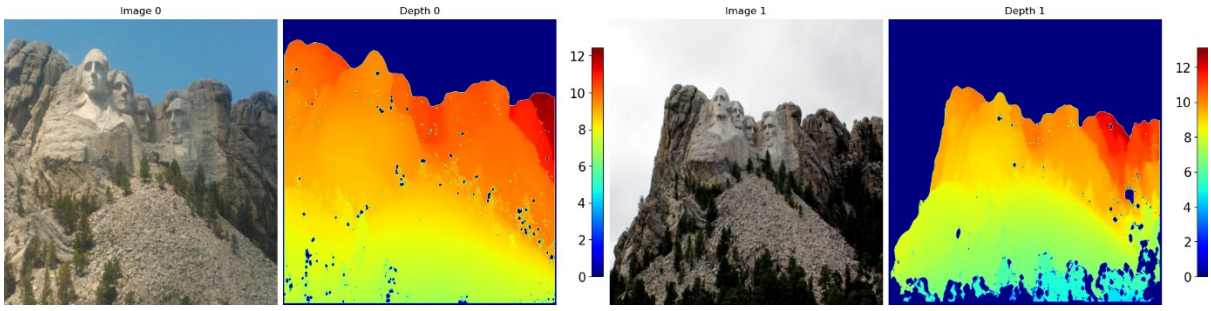
The choice of the local window size in Census Transform has a significant impact on the quality of the resulting map, as it directly affects how much context is used for computation. Increasing the window size seems to improve the accuracy, thus resulting in better output quality.

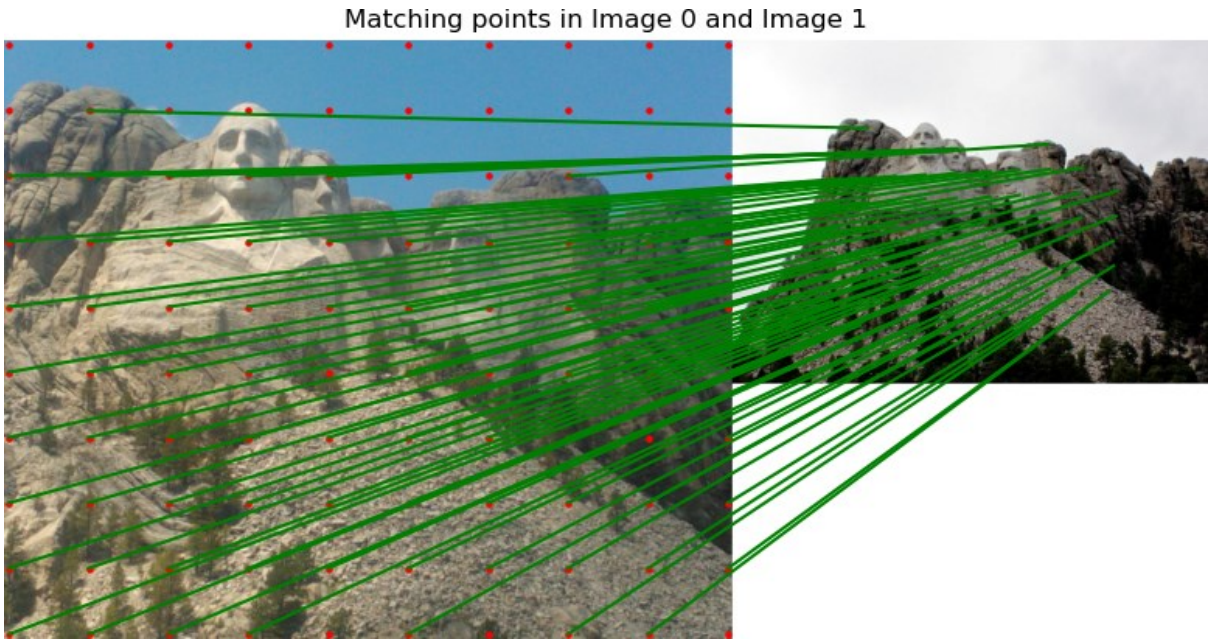
Task 3: Dense Stereo Matching

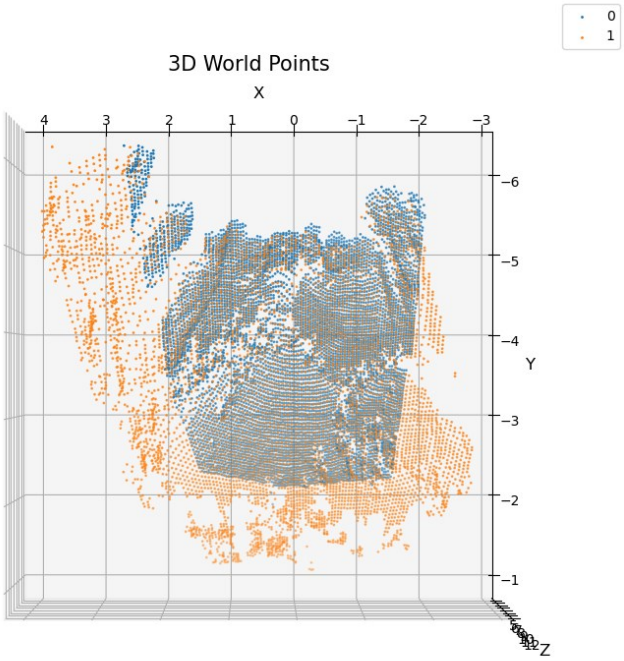
Automatic extraction of dense correspondences using depth maps involves matching pixels between two images by leveraging their depth information and camera parameters. A depth map provides the distance of each pixel from the camera, allowing 2D image points to be converted into 3D coordinates. These 3D points are projected into the second image to identify corresponding pixels, and a depth check ensures the validity of these matches by comparing the calculated depth with the depth map of the second image. This process generates dense and accurate correspondences, which are essential for tasks like stereo matching, 3D reconstruction, and image registration. It is particularly useful in training deep learning models, such as SuperPoint and LoFTR, by providing reliable ground truth for keypoint matching. By automating the correspondence extraction, this technique eliminates manual effort, improves scalability, and enhances the accuracy of various computer vision applications.

RESULTS

- Pair 0:






0
1
3D World Points
X
Y
Z

- Pair 1:

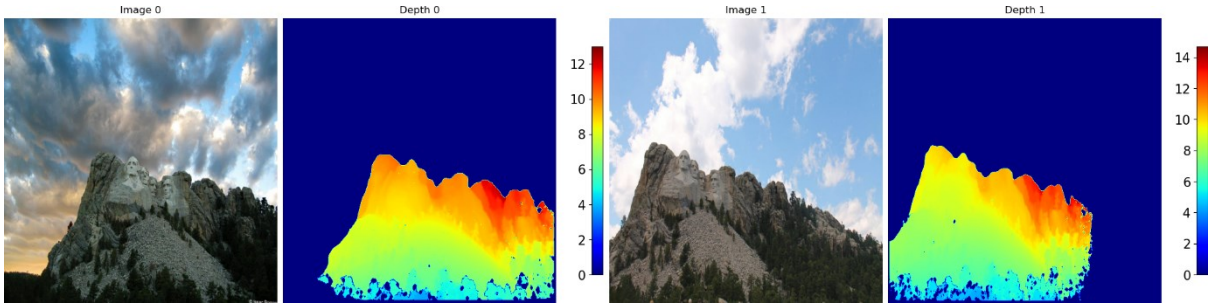
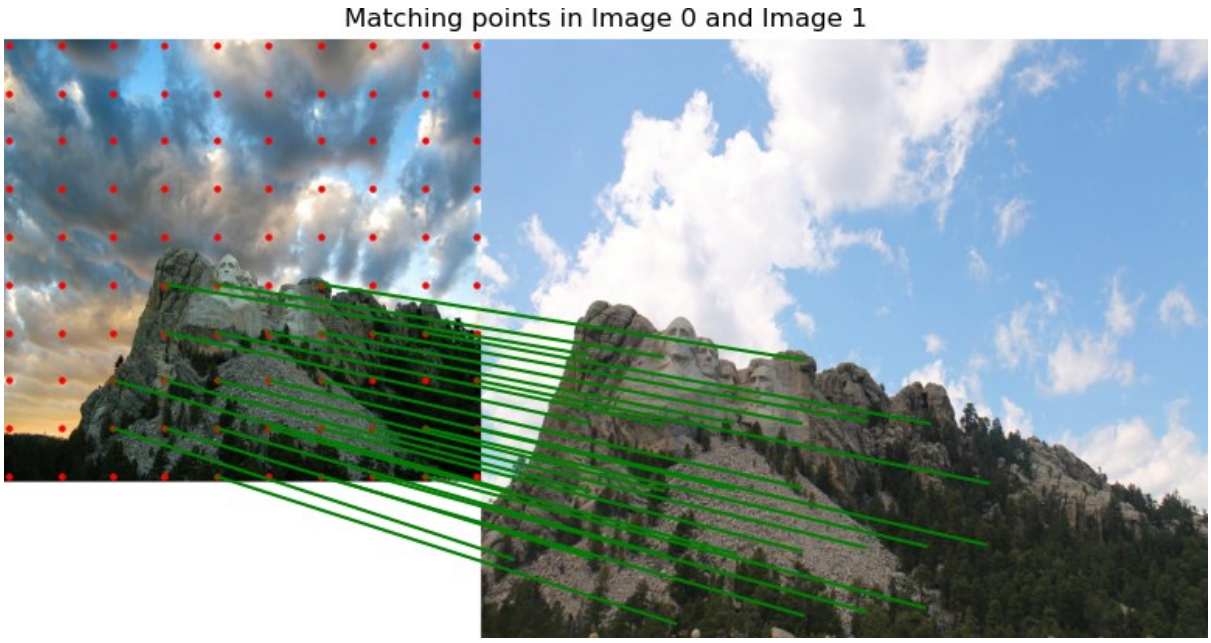
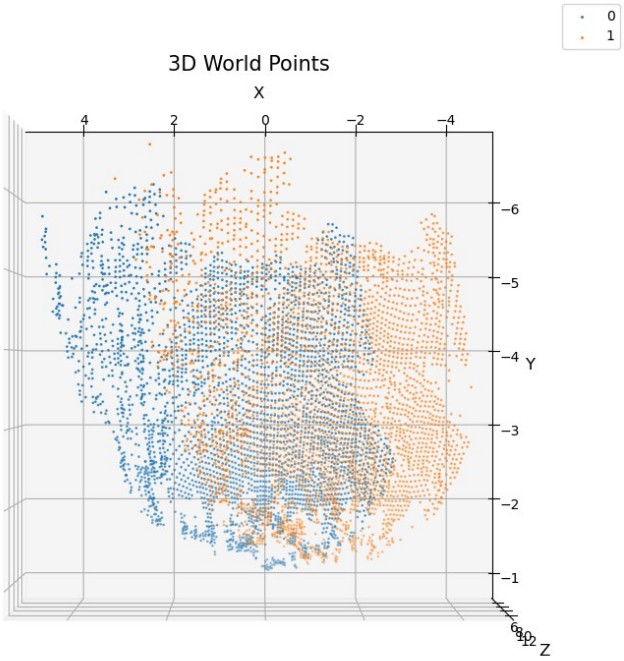


Image 0
Depth 0
Image 1
Depth 1


Matching points in Image 0 and Image 1


0
1
3D World Points
X
4
2
0
−2
−4
−6
−5
−4
Y
−3
−2
−1
Z

- Pair 2:

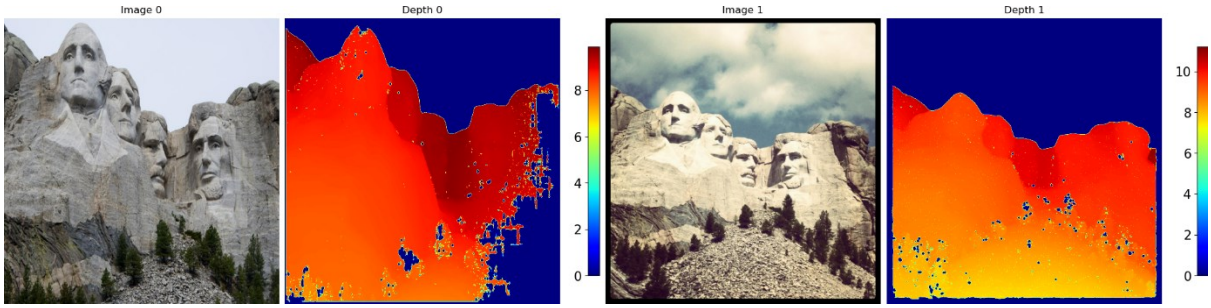
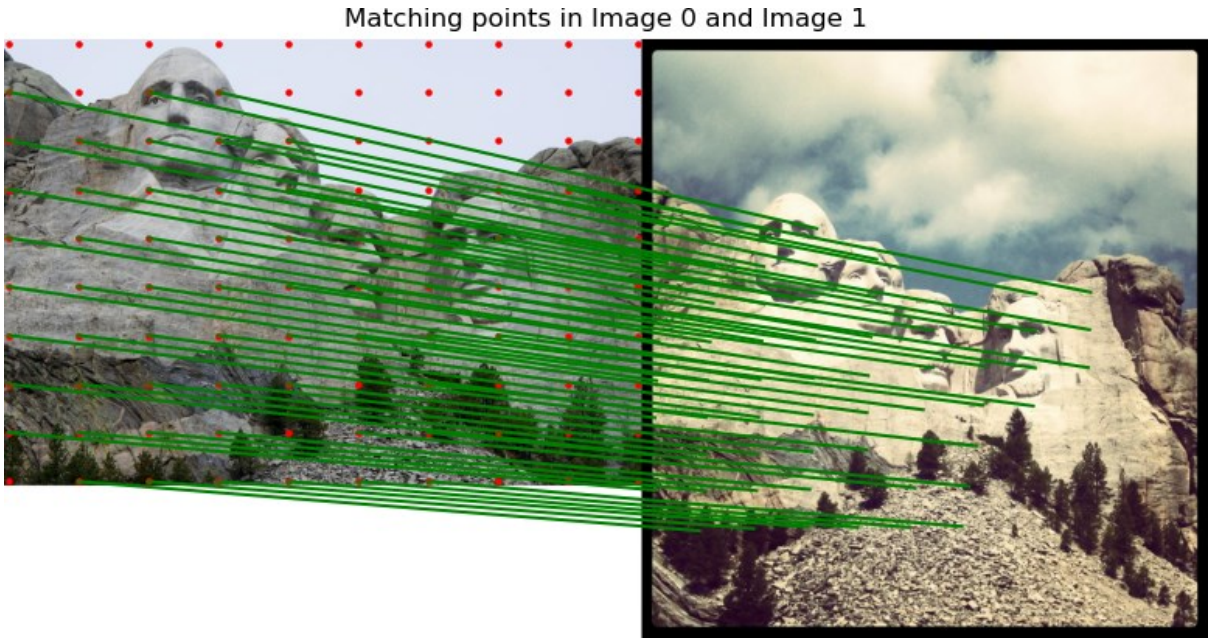
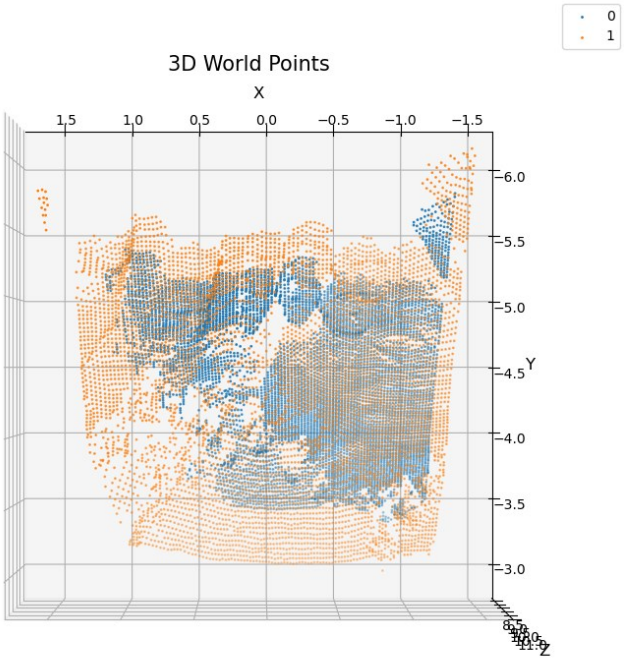


Image 0
Depth 0
Image 1
Depth 1
0
2
4
6
8
10


Matching points in Image 0 and Image 1


0
1
3D World Points
X
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
Y
Z

- Pair 3:

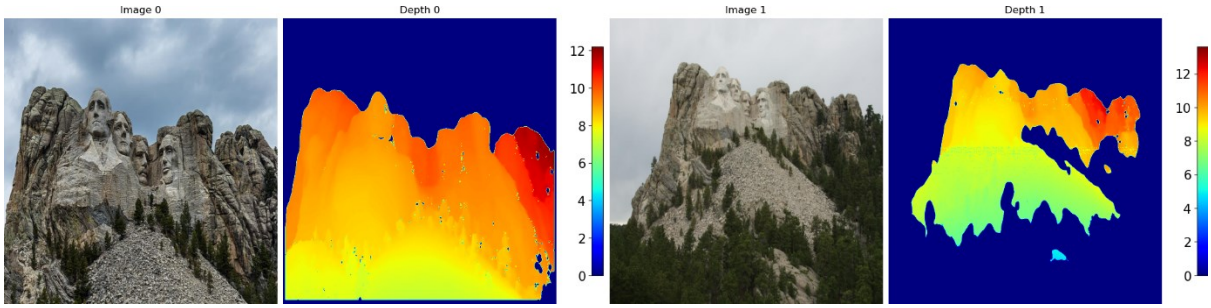
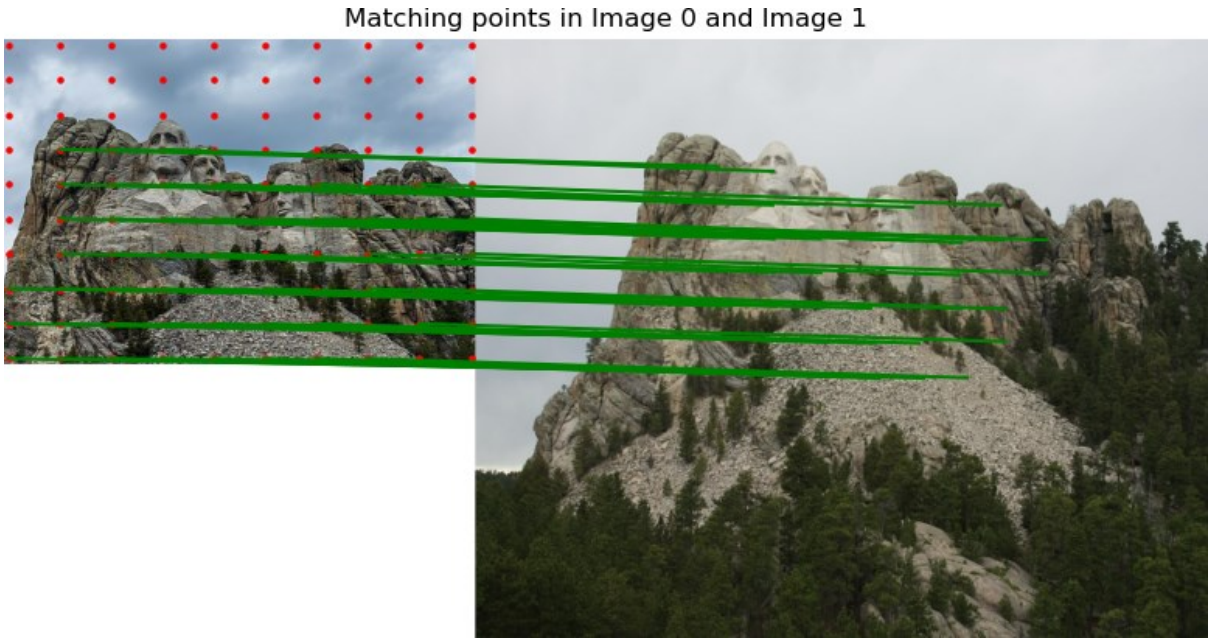
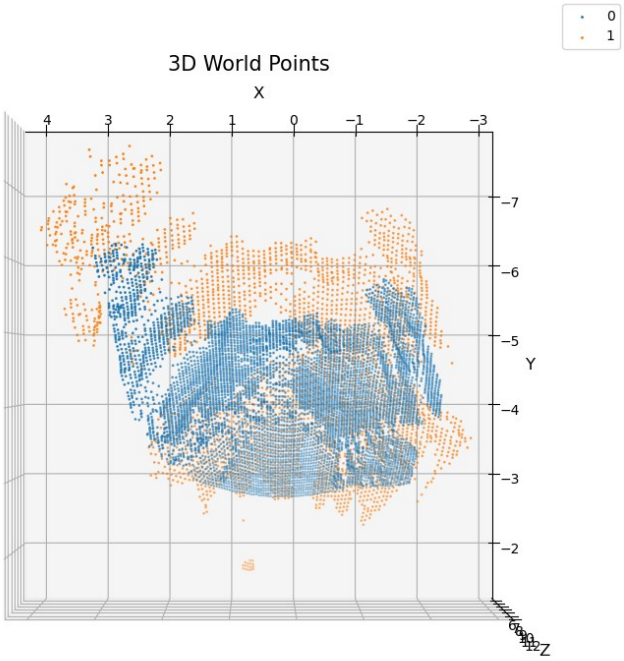


Image 0
Depth 0
Image 1
Depth 1
12
10
8
6
4
2
0


Matching points in Image 0 and Image 1


0
1
3D World Points
X
Y
Z

- Pair 4:

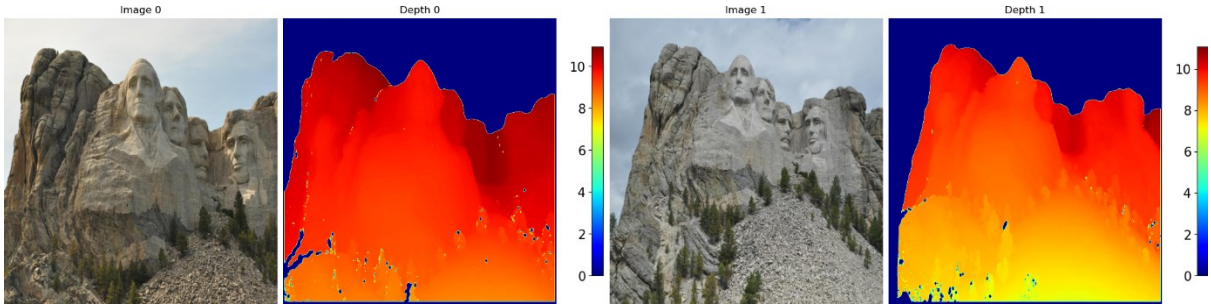
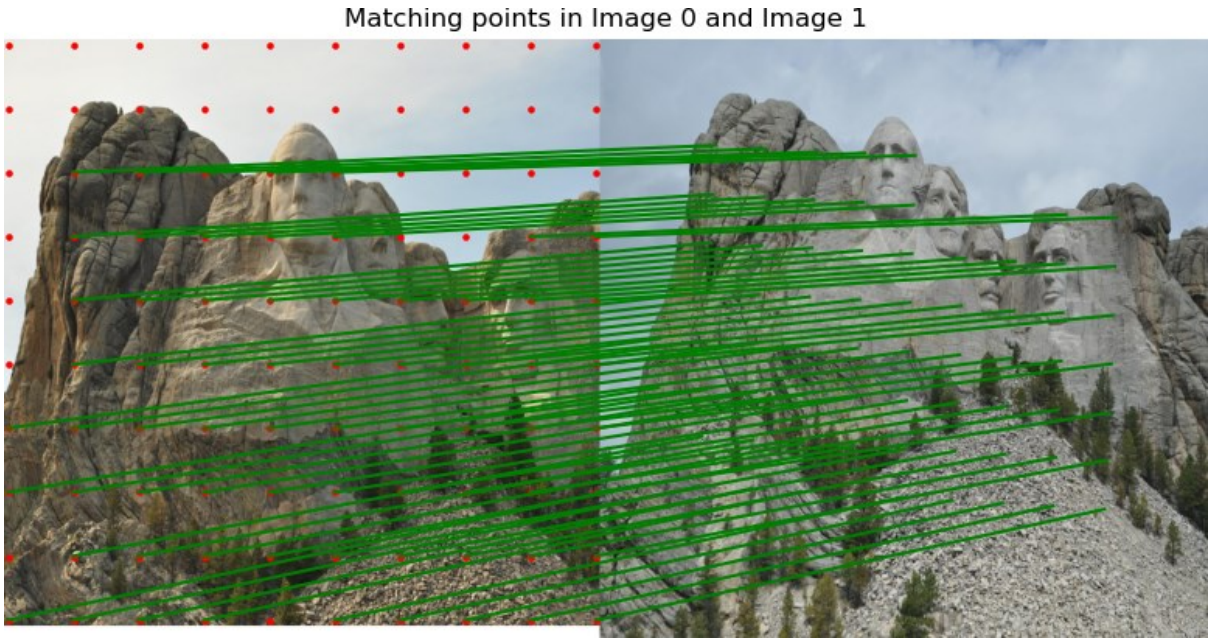
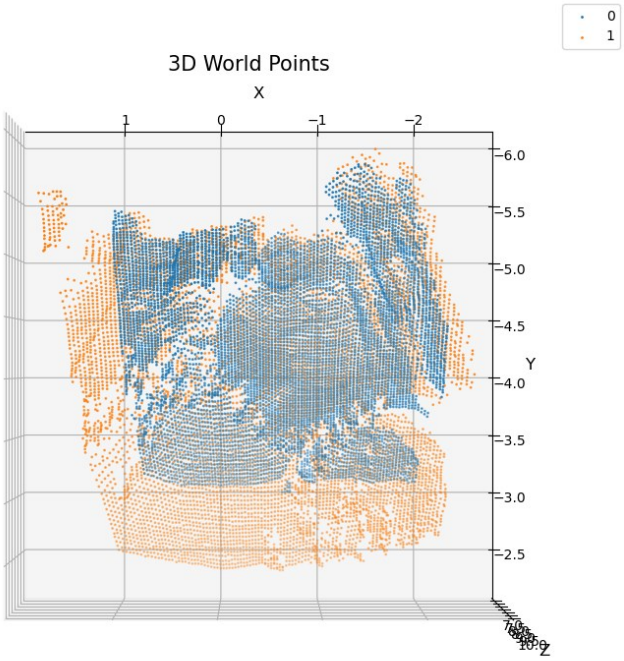


Image 0
Depth 0
Image 1
Depth 1


Matching points in Image 0 and Image 1


0
1
3D World Points
X
1
0
−1
−2
6.0
5.5
5.0
4.5
Y
4.0
3.5
3.0
2.5
Z

- Pair 5:

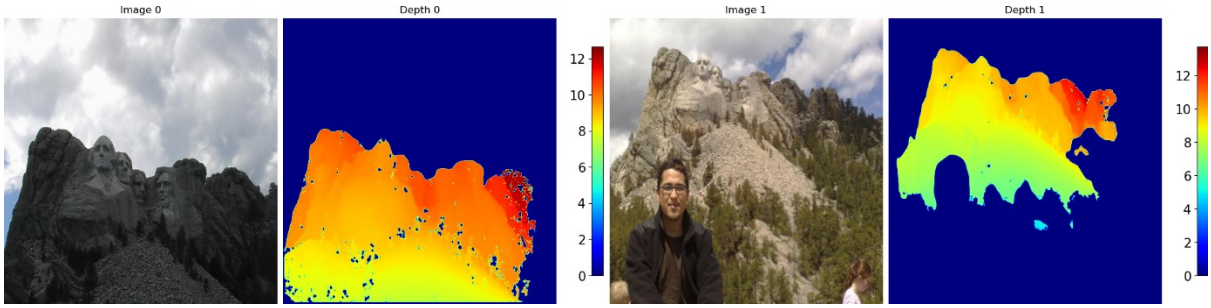
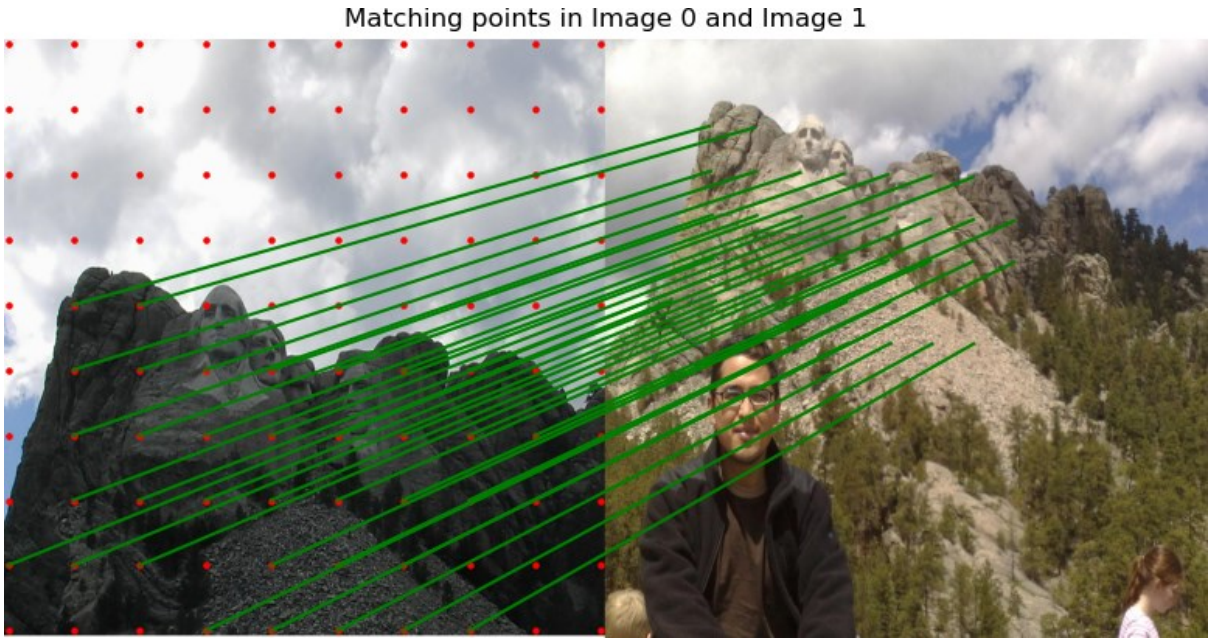
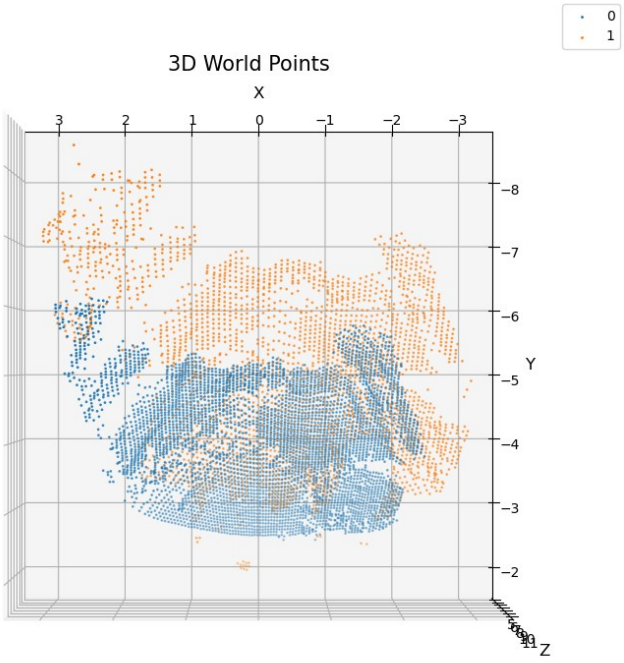


Image 0
Depth 0
Image 1
Depth 1
12
10
8
6
4
2
0


Matching points in Image 0 and Image 1


0
1
3D World Points
X
3
2
1
0
−1
−2
−3
−8
−7
−6
−5
−4
−3
−2
Y
Z

- Pair 6:

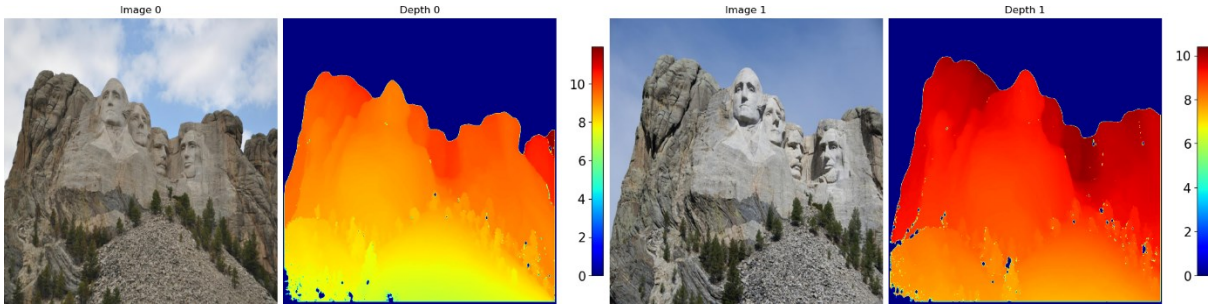
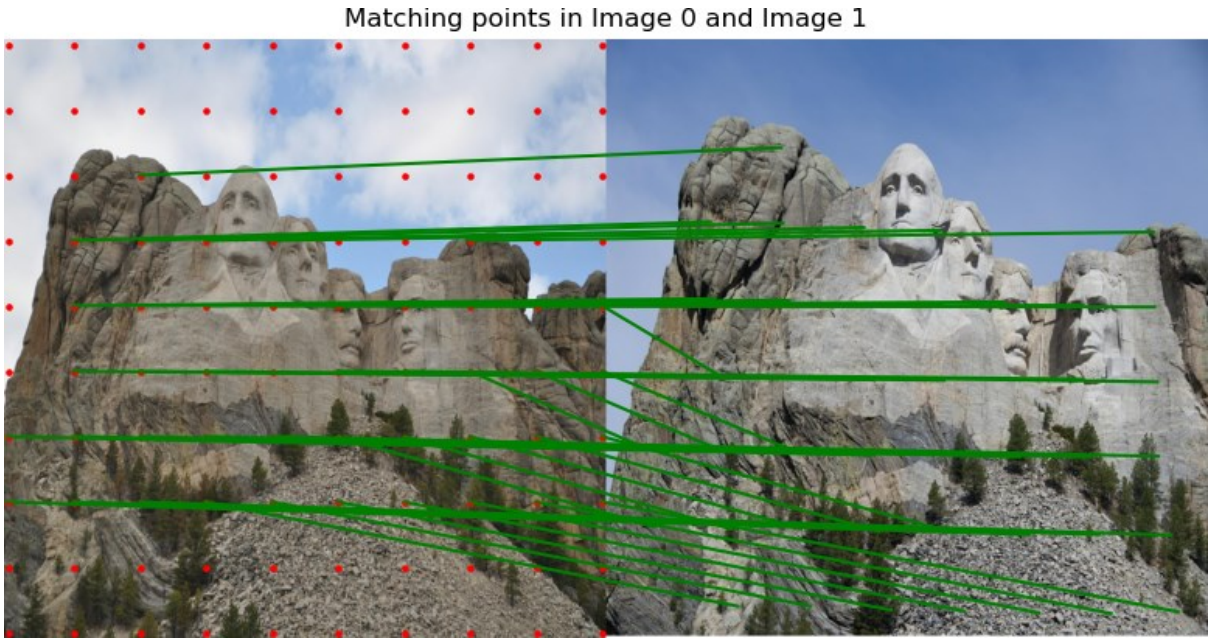
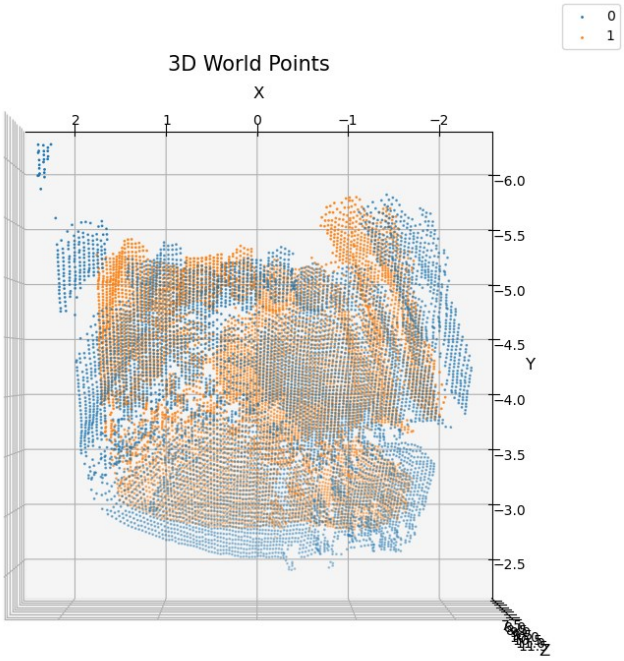


Image 0
Depth 0
Image 1
Depth 1
10
8
6
4
2
0


Matching points in Image 0 and Image 1


0
1
3D World Points
X
2
1
0
−1
−2
6.0
5.5
5.0
4.5
Y
4.0
3.5
3.0
2.5
Z

- Pair 7:

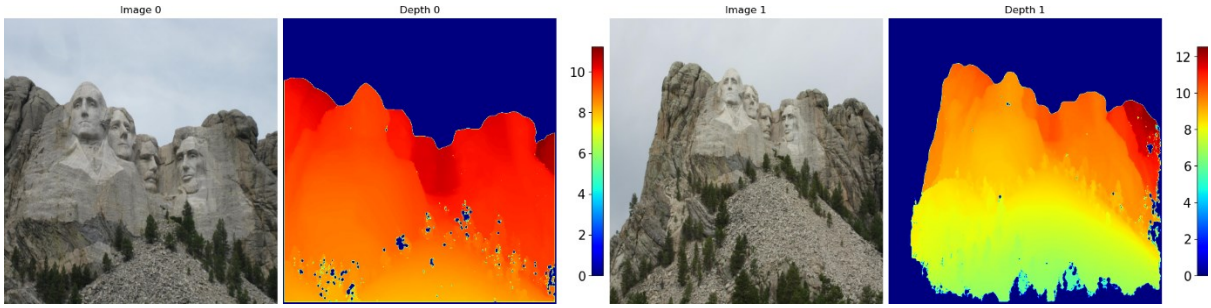
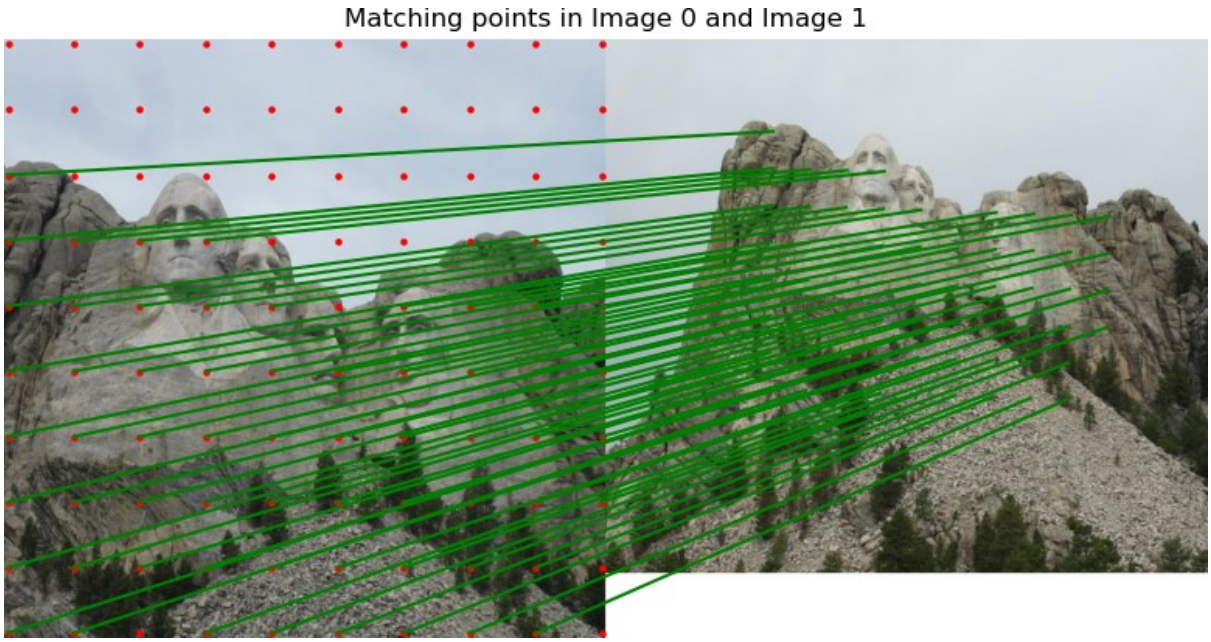
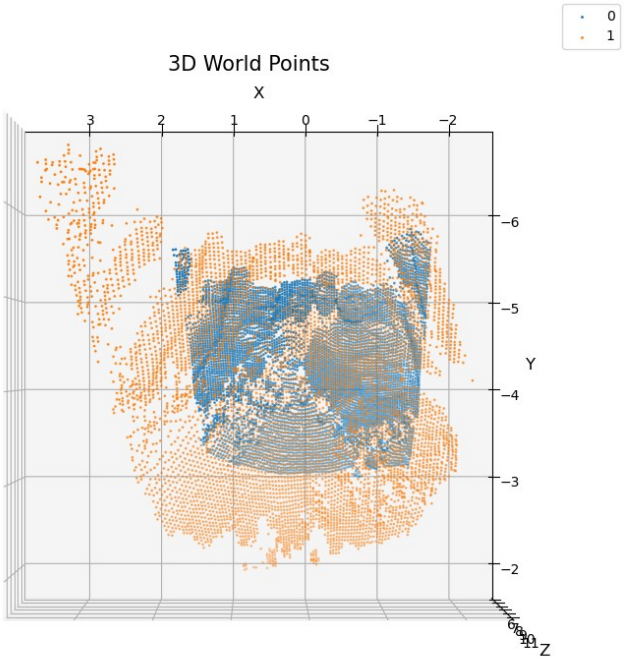
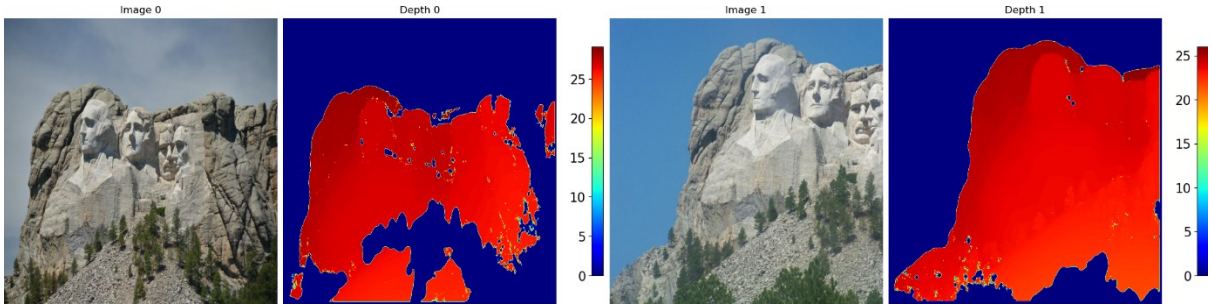


Image 0
Depth 0
Image 1
Depth 1
10
8
6
4
2
0
12
10
8
6
4
2
0

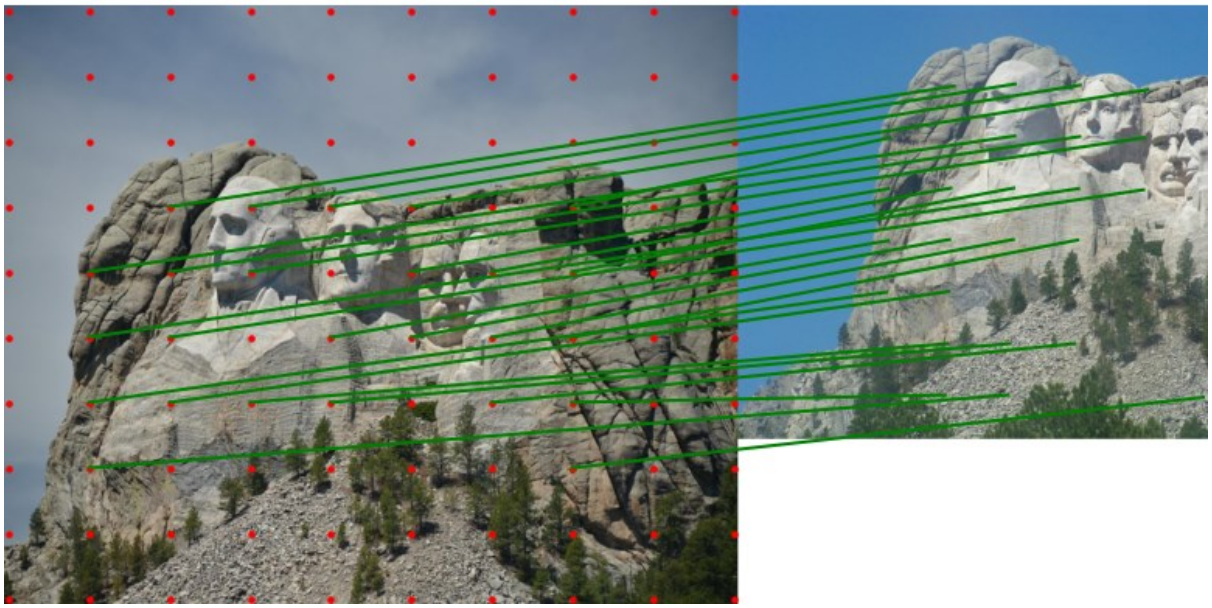

Matching points in Image 0 and Image 1

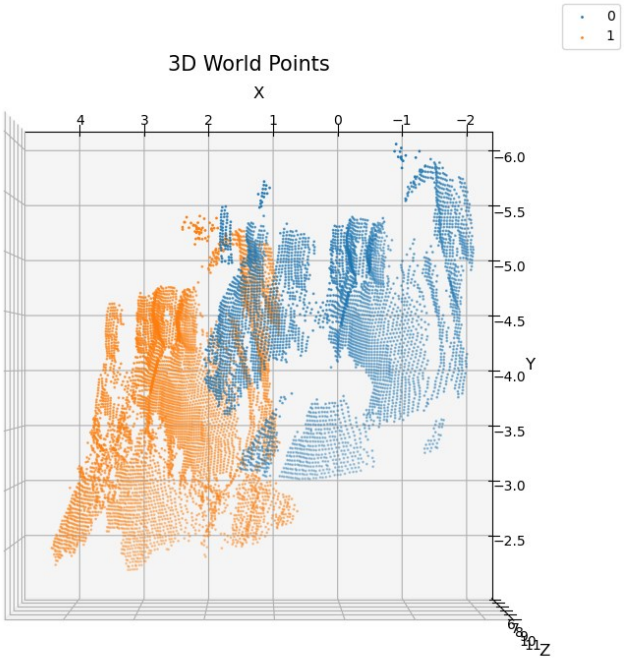


- Pair 8:



Matching points in Image 0 and Image 1




0
1
3D World Points
X
Y
Z

- Pair 9:

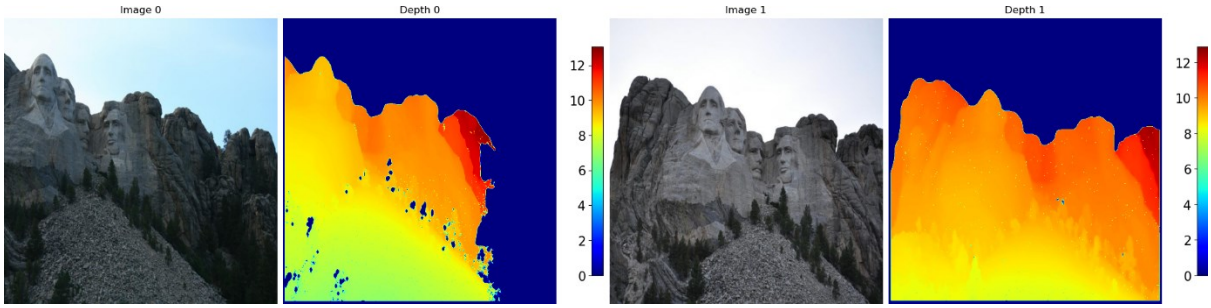
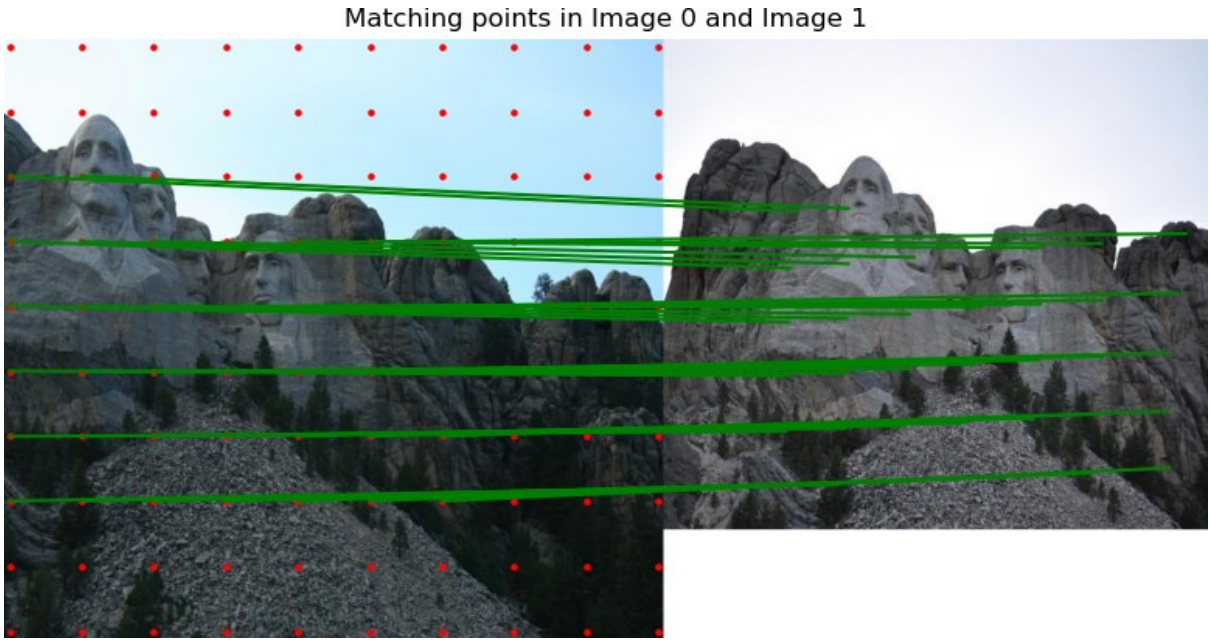
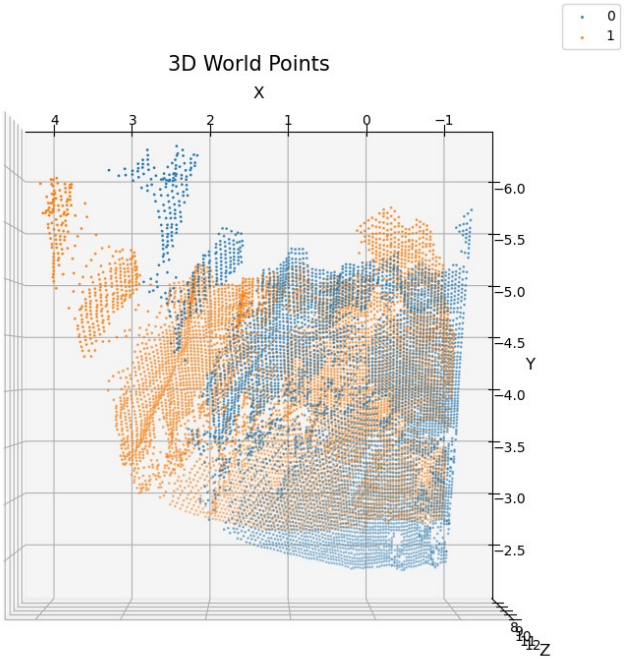


Image 0
Depth 0
Image 1
Depth 1


Matching points in Image 0 and Image 1



Acknowledgment:

- https://engineering.purdue.edu/kak/computervision/

```
# -*- coding: utf-8 -*-
"""HW9_CV_F.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1kGGFmpb0973791tWxde1hkowiCj0yZO0
"""

import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import random
import math
import os
import scipy.optimize as optim
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.patches import Rectangle
from mpl_toolkits.mplot3d import art3d,proj3d
from matplotlib.patches import Circle, ConnectionPatch
import matplotlib.lines as mplot_lines

"""Projective Stereo Reconstruction"""

#box
image1 = cv.imread('/content/box1.jpg')
image2 = cv.imread('/content/box2.jpg')

img1 = np.array([(1996,141),(1034,736),(2177,1321),(3002,473),(1171,1199),(2206,1794),(2948,897),(3128,487)])
img2 = np.array([(2235,107),(1186,551),(1820,1209),(2992,541),(1308,951),(1937,1658),(2972,975),(3270,575)])

#transforming pixel coordinates
def transform(image_d,H):
    img_dim = np.array([[0,0],[image_d.shape[1],0],[image_d.shape[1],image_d.shape[0]],[0,image_d.shape[0]]])
    rng_dim = np.zeros(img_dim.shape,dtype=int)
    for i in range(len(img_dim)):
        x = np.append(img_dim[i],[1])
        x_d = H@x
        rng_dim[i][0] = int(x_d[0]/x_d[2])
        rng_dim[i][1] = int(x_d[1]/x_d[2])

    y_lim,x_lim = max(rng_dim[:,1])-min(rng_dim[:,1]),max(rng_dim[:,0])-min(rng_dim[:,0])
    image_r = np.zeros((y_lim,x_lim,3),dtype=int)

    for j in range(y_lim):
        for i in range(x_lim):
            x = np.array([i+min(rng_dim[:,0]), j+min(rng_dim[:,1]), 1])
            x_d = np.linalg.inv(H)@x
            xi = int(x_d[0]/x_d[2])
            xj = int(x_d[1]/x_d[2])
            if 0<= xj <image_d.shape[0] and 0<= xi < image_d.shape[1]:
                image_r[j,i] = image_d[xj,xi]
    return image_r

#Image Rectification

def estimate_F(pts1,pts2):
    assert pts1.shape==pts2.shape
    A = []
    for i in range(len(pts1)):
        A.append([pts2[i][0]*pts1[i][0], pts2[i][0]*pts1[i][1], pts2[i][0], pts2[i][1]*pts1[i][0], pts2[i][1]*pts1[i][1], pts2[i][1], pts1[i][0], pts1[i][1], 1
    A = np.array(A)
    U,D,V = np.linalg.svd((A.T)@A)
    F = V[-1].reshape((3,3))
    # condition F
    U,D,V = np.linalg.svd(F)
    D[-1] = 0
    F = U@np.diag(D)@V
    return F/F[-1,-1]


def estimate_epipoles(F):
    U,D,V = np.linalg.svd(F)
    e = V[-1]   #
    e = e/e[-1]
    U,D,V = np.linalg.svd(F.T)
    e_d = V[-1] #
    e_d = e_d/e_d[-1]
    return e,e_d


def est_proj_mat(F,e_d):
    P = np.column_stack((np.eye(3),np.zeros(3)))
    e_d_Xmat = np.array([[0,-e_d[2],e_d[1]],[e_d[2],0,-e_d[0]],[-e_d[1],e_d[0],0]])
    P_d = np.column_stack((e_d_Xmat@F,e_d))
    return P,P_d


def world_pts_w_triangulation(x,x_d,P,P_d):
    world_pts = []
    #triangulate
    for i in range(len(x)):
        A = np.zeros((4,4))
        A[0,:] = x[i][0]*P[2,:]-P[0,:]
        A[1,:] = x[i][1]*P[2,:]-P[1,:]
        A[2,:] = x_d[i][0]*P_d[2,:]-P_d[0,:]
        A[3,:] = x_d[i][1]*P_d[2,:]-P_d[1,:]
        U,D,V = np.linalg.svd((A.T)@A)
        world_pt = V[-1]
        world_pt /= world_pt[-1]
        world_pts.append(world_pt)
    world_pts = np.array(world_pts)
    world_pts = world_pts[:,:-1]
    return world_pts
```

```
def cost_refine_P_d(params,x,x_d,P):
    P_d = params[:12].reshape((3,4))
    world_pts = params[12:].reshape((-1,3))
    world_pts_HC = np.hstack((world_pts,np.ones((len(world_pts),1))))
    x_h = (P@(world_pts_HC.T)).T
    x_h = (x_h/x_h[:,-1].reshape(-1,1))[:,:-1]
    x_d_h = (P_d@(world_pts_HC.T)).T
    x_d_h = (x_d_h/x_d_h[:,-1].reshape(-1,1))[:,:-1]
    diff_x_h = (x - x_h)
    diff_x_d_h = (x_d - x_d_h)
    cost = np.array(list(diff_x_h.flatten())+list(diff_x_d_h.flatten()))
    return cost


def refine_P_d_and_F(x,x_d,P,P_d):
    world_pts = world_pts_w_triangulation(x,x_d,P,P_d)
    params = np.array(list(P_d.flatten()) + list(world_pts.flatten()))
    res_lsq = optim.least_squares(cost_refine_P_d,params,args=(x,x_d,P))
    params_r = res_lsq.x
    P_d_r = params_r[:12].reshape((3,4))
    e_d_r = P_d_r[:,-1]
    e_d_r_mat = np.array([[0,-e_d_r[2],e_d_r[1]],[e_d_r[2],0,-e_d_r[0]],[-e_d_r[1],e_d_r[0],0]])
    F_r = e_d_r_mat@P_d_r@np.linalg.pinv(P)
    F_r /= F_r[-1,-1]
    return F_r,P_d_r


def est_homographies(img,e_d,P,P_d,x,x_d):
    h,w = img.shape[:2]
    y0,x0 = h/2,w/2
    #right homography
    T  = np.array([[1,0,-x0],[0,1,-y0],[0,0,1]])
    T2 = np.array([[1,0,x0],[0,1,y0],[0,0,1]])
    theta = np.arctan(-(e_d[1]-y0)/(e_d[0]-x0))
    f = abs((e_d[0]-x0)*np.cos(theta)-(e_d[1]-y0)*np.sin(theta))
    R = np.array([[np.cos(theta),-np.sin(theta),0],[np.sin(theta),np.cos(theta),0],[0,0,1]])
    G = np.array([[1,0,0],[0,1,0],[-1/f,0,1]])
    H_d = T2@G@R@T
    H_d = H_d/H_d[-1,-1]
    #left homography
    M = P_d@np.linalg.pinv(P)
    H0 = H_d@M
    x = np.hstack((x,np.ones((len(x),1))))
    x_d = np.hstack((x_d,np.ones((len(x_d),1))))
    x_h = (H0@(x.T)).T
    x_h = (x_h/x_h[:,-1].reshape(-1,1))[:,:-1]
    x_d_h = (H_d@(x_d.T)).T
    x_d_h = (x_d_h/x_d_h[:,-1].reshape(-1,1))[:,:-1]
    A = np.ones((len(x_h),3))
    A[:,:-1] = x_h
    b = x_d_h[:,0]
    abc_vec = np.linalg.pinv(A)@b
    H_A = np.array([[abc_vec[0],abc_vec[1],abc_vec[2]],[0,1,0],[0,0,1]])
    H = H_A@H0
    H = H/H[-1,-1]
    return H,H_d


def plot_rectified_images(image1,image2,H,H_d):
    img1_rec = cv.warpPerspective(image1,H,(image1.shape[1],image1.shape[0]))
    img2_rec = cv.warpPerspective(image2,H_d,(image2.shape[1],image2.shape[0]))
    # img1_rec = transform(image1,H)
    # img2_rec = transform(image2,H_d)
    fig,axes = plt.subplots(1,2)
    fig.suptitle('rectified images',y=0.7)
    axes[0].imshow(img1_rec[:,:,::-1])
    axes[1].imshow(img2_rec[:,:,::-1])
    return img1_rec,img2_rec


def rectify_images(plot_pts=False):

    if plot_pts:
        combined_img = np.hstack((image1,image2))
        for pt1, pt2 in zip(img1,img2):
            pt2_shifted = (pt2[0] + image1.shape[1], pt2[1])
            cv.circle(combined_img,pt1, 20, (100, 200, 100), -1)
            cv.circle(combined_img,pt2_shifted, 20, (100, 200, 100), -1)
            cv.line(combined_img, pt1, pt2_shifted, (0, 255, 0), 10)
        plt.imshow(combined_img[:,:,::-1])
        plt.title('correspondences')
        plt.show()

    F = estimate_F(img1,img2)
    e,e_d = estimate_epipoles(F)
    P,P_d = est_proj_mat(F,e_d)
    F,P_d = refine_P_d_and_F(img1,img2,P,P_d)
    e,e_d = estimate_epipoles(F)
    H,H_d = est_homographies(image2,e_d,P,P_d,img1,img2)

    img1_rec,img2_rec = plot_rectified_images(image1,image2,H,H_d)
    # print(H_d)
    # print(H)
    return combined_img,img1_rec,img2_rec,[H,H_d,e,e_d,P,P_d,F]

combined_img,img1_rec,img2_rec,ParamList = rectify_images(plot_pts=True)

def draw_epipolarLines(image1,image2,img1_pts,img2_pts,F):
    epolarLines1 = (F@(np.hstack((img1_pts,np.ones((len(img1_pts),1))))).T).T
    epolarLines2 = (F.T@(np.hstack((img2_pts,np.ones((len(img2_pts),1))))).T).T
    image1_copy = np.copy(image1)
    image2_copy = np.copy(image2)
    for i in range(len(img1_pts)):
        cv.circle(image1_copy,(img1_pts[i][0],img1_pts[i][1]),radius=30,thickness=-1,color=(0,0,255))
        cv.circle(image2_copy,(img2_pts[i][0],img2_pts[i][1]),radius=30,thickness=-1,color=(0,0,255))
    for i in range(len(epolarLines1)):
```

```
        cv.line(image1_copy, (0, int(-epolarLines2[i][2]/epolarLines2[i][1])), (image1.shape[1]-1,int(-(epolarLines2[i][2]+epolarLines2[i][0]*(image1.shape[1]-
        cv.line(image2_copy, (0, int(-epolarLines1[i][2]/epolarLines1[i][1])), (image2.shape[1]-1,int(-(epolarLines1[i][2]+epolarLines1[i][0]*(image2.shape[1]-
    fig,axes = plt.subplots(1,2, figsize=(10,8))
    axes[0].imshow(image1_copy[:,:,::-1])
    axes[1].imshow(image2_copy[:,:,::-1])
    fig.suptitle('epipolar lines',y=0.65)
    plt.show()
    return image1_copy,image2_copy
ep_img1,ep_img2 = draw_epipolarLines(image1,image2,img1,img2,ParamList[-1])

# Canny detector
thres_l,thres_h = 300,400

img1_gray = cv.cvtColor(img1_rec,cv.COLOR_BGR2GRAY)
img2_gray = cv.cvtColor(img2_rec,cv.COLOR_BGR2GRAY)
edges_img1 = cv.Canny(img1_gray,thres_l,thres_h)
edges_img2 = cv.Canny(img2_gray,thres_l,thres_h)
fig,axes = plt.subplots(1,2, figsize=(10,8))
axes[0].imshow(edges_img1,cmap='gray')
axes[1].imshow(edges_img2,cmap='gray')
fig.suptitle('Canny edges',y=0.65)
plt.show()

points_img1 = np.argwhere(edges_img1>0)
points_img2 = np.argwhere(edges_img2>0)

def SSD(image1, image2, img1_pts, img2_pts, sig):
    img1_op = np.copy(image1)
    image1 = cv.cvtColor(image1, cv.COLOR_BGR2GRAY).astype(np.float32)/255.0
    img2_op = np.copy(image2)
    image2 = cv.cvtColor(image2, cv.COLOR_BGR2GRAY).astype(np.float32)/255.0
    paired_img = np.concatenate((img1_op, img2_op), axis=1)

    M = int(5*sig)+1
    N = M//2
    delta_bound = 3

    shortest_dist_array = []
    for c1 in img1_pts:
        dist_array = []

        # Filter points in img2 based on proximity to c1
        filter_mask = np.abs(img2_pts[:,0] - c1[0]) <= delta_bound
        filtered_img2_pts = img2_pts[filter_mask]
        # Skip if no valid points in img2
        if filtered_img2_pts.size == 0:
            shortest_dist_array.append(None)
            continue

        for c2 in filtered_img2_pts:
            # Extract patches
            f1_w = image1[max(0, c1[0] - N):c1[0] + N + 1, max(0, c1[1] - N):c1[1] + N + 1]
            f2_w = image2[max(0, c2[0] - N):c2[0] + N + 1, max(0, c2[1] - N):c2[1] + N + 1]
            # Check if patches are valid
            if f1_w.shape == f2_w.shape and f1_w.size > 0:
                dist = np.sum((f1_w - f2_w) ** 2)
                dist_array.append([c2, dist])

        # Skip if no valid distances found
        if not dist_array:
            shortest_dist_array.append(None)
            continue
        # Find the closest match based on SSD
        shortest_dist = sorted(dist_array, key=lambda x: x[1])[0]
        shortest_dist_array.append(shortest_dist)

    filtered_idx = [i for i,x in enumerate(shortest_dist_array) if x!=None]
    shortest_dist_array = [x for x in shortest_dist_array if x!=None]
    img1_pts = img1_pts[filtered_idx]
    # print(img1_pts.shape,len(shortest_dist_array))
    # SSD threshold
    ssd_threshold = np.percentile([x[1] for x in shortest_dist_array], 10)
    final_img1_pts,final_img2_pts = [],[]
    # Visualize matches
    for p1, p2 in zip(img1_pts, shortest_dist_array):
        if p2[1] < ssd_threshold:
            cv.circle(paired_img,(p1[1],p1[0]),radius=3,thickness=-1,color=(0,0,255))
            cv.circle(paired_img,(p2[0][1]+image1.shape[1],p2[0][0]),radius=3,thickness=-1,color=(0,0,255))
            cv.line(paired_img,(p1[1],p1[0]),(p2[0][1]+image1.shape[1],p2[0][0]),(0,255,0),1)
            final_img1_pts.append(p1)
            final_img2_pts.append(p2[0])
    plt.figure(figsize=(10,8))
    plt.imshow(paired_img[:, :, ::-1])
    plt.axis('off')
    plt.show()
    assert img1_pts.shape[0]==len(shortest_dist_array)
    return np.array(final_img1_pts),np.array(final_img2_pts)

final_img1_pts,final_img2_pts = SSD(img1_rec,img2_rec,points_img1,points_img2,sig=1.6)

# plot_rectified_images(ep_img1,ep_img2,ParamList[0],ParamList[1])

def inv_rectification(pts1,pts2,H,H_d):
    pts1 = np.hstack((pts1,np.ones((len(pts1),1))))
    pts2 = np.hstack((pts2,np.ones((len(pts2),1))))
    x = (np.linalg.inv(H)@pts1.T).T
    x = (x/x[:,-1].reshape(-1,1))[:,:-1]
    x_d = (np.linalg.inv(H_d)@pts2.T).T
    x_d = (x_d/x_d[:,-1].reshape(-1,1))[:,:-1]
    return x,x_d

inv_rect_x, inv_rect_x_d = inv_rectification(final_img1_pts,final_img2_pts,ParamList[0],ParamList[1])

reconstruct_world_pts = world_pts_w_triangulation(inv_rect_x,inv_rect_x_d,ParamList[4],ParamList[5])

selected_img_world_pts = world_pts_w_triangulation(img1,img2,ParamList[4],ParamList[5])
```

```
# refine world points
def refine_recont_world_pts(x,x_d,P,P_d):
    world_pts = world_pts_w_triangulation(x,x_d,P,P_d)  #reconstructed world pts

    def cost_refine_recont_world_pts(params):
        world_pts = params.reshape((-1,3))
        world_pts_HC = np.hstack((world_pts,np.ones((len(world_pts),1))))
        x_h = (P@(world_pts_HC.T)).T
        x_h = (x_h/x_h[:,-1].reshape(-1,1))[:,:-1]
        x_d_h = (P_d@(world_pts_HC.T)).T
        x_d_h = (x_d_h/x_d_h[:,-1].reshape(-1,1))[:,:-1]
        diff_x_h = (x - x_h)
        diff_x_d_h = (x_d - x_d_h)
        cost = np.array(list(diff_x_h.flatten())+list(diff_x_d_h.flatten()))
        return cost

    params = world_pts.flatten()    #np.array(list(diff_x_h.flatten())+list(diff_x_d_h.flatten()))
    res_lsq = optim.least_squares(cost_refine_recont_world_pts,params)
    params_r = res_lsq.x
    world_pts_r = params_r.reshape((-1,3))
    return world_pts_r

reconstruct_world_pts_refined = refine_recont_world_pts(inv_rect_x,inv_rect_x_d,ParamList[4],ParamList[5])



# Create 3D plot
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.view_init(elev=30, azim=90, )
# Plot reconstructed points
ax.scatter(reconstruct_world_pts_refined[:,0], reconstruct_world_pts_refined[:,1], reconstruct_world_pts_refined[:,2], color='b', label='3D Points')
# ax.scatter(selected_img_world_pts[:,0], selected_img_world_pts[:,1], selected_img_world_pts[:,2],color='r')

# Labels and final touches
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.legend()
plt.show()

def plot_3D_points(img1,img2,pts_3d=None,recont_world_pts=[]):

  pts_3d = pts_3d
  pts_2d_img1 = img1
  pts_2d_img2 = img2

  fig = plt.figure(figsize=(12, 8))

  ax = fig.add_subplot(2,3,2, projection='3d')
  # ax.view_init(elev=-9, azim=120, roll=-9)
  pt_lines_idx = [(0,1),(0,3),(3,2),(1,2),(1,4),(4,5),(2,5),(5,6),(3,6),(6,7)]

  image1_cp = np.copy(image1)
  image2_cp = np.copy(image2)
  axes1 = fig.add_subplot(223)
  # axes1.set_aspect('equal')
  axes1.imshow(image1_cp[:,:,::-1])
  axes2 = fig.add_subplot(224)
  # axes1.set_aspect('equal')
  axes2.imshow(image2_cp[:,:,::-1])

  if len(recont_world_pts) > 0 :
    pts_3d_xMean,pts_3d_yMean,pts_3d_zMean = np.mean(pts_3d,axis=0)
    x_max,y_max,z_max = 40,55,0.2
    # mask = (-x_max<recont_world_pts[:,0]-pts_3d_xMean)&(recont_world_pts[:,0]-pts_3d_xMean<x_max) & (-y_max<recont_world_pts[:,1]-pts_3d_yMean)&(recont_world
    mask = (recont_world_pts[:,0]<x_max) & (recont_world_pts[:,1]<y_max) & (recont_world_pts[:,2]<z_max)
    recont_world_pts = recont_world_pts[mask]
    ax.scatter(recont_world_pts[:,0], recont_world_pts[:,1], recont_world_pts[:,2], color='b')
  ax.scatter(pts_3d[:,0], pts_3d[:,1], pts_3d[:,2], color='r', label='World Points')
  axes1.scatter(pts_2d_img1[:,0], pts_2d_img1[:,1], color='g')
  axes2.scatter(pts_2d_img2[:,0], pts_2d_img2[:,1], color='g')

  for i,j in pt_lines_idx:
    ax.plot(xs=[pts_3d[i,0],pts_3d[j,0]],ys=[pts_3d[i,1],pts_3d[j,1]],zs=[pts_3d[i,2],pts_3d[j,2]], color='b')
    axes1.plot([pts_2d_img1[i,0],pts_2d_img1[j,0]],[pts_2d_img1[i,1],pts_2d_img1[j,1]],color='r')
    axes2.plot([pts_2d_img2[i,0],pts_2d_img2[j,0]],[pts_2d_img2[i,1],pts_2d_img2[j,1]],color='r')

  for i in range(len(pts_2d_img1)):
    xs,ys,_ = proj3d.proj_transform(pts_3d[i][0],pts_3d[i][1],pts_3d[i][2],ax.get_proj())
    conn = ConnectionPatch((pts_2d_img1[i][0],pts_2d_img1[i][1]),(xs,ys),coordsA=axes1.transData,coordsB=ax.transData)
    axes1.add_artist(conn)
    conn = ConnectionPatch((pts_2d_img2[i][0],pts_2d_img2[i][1]),(xs,ys),coordsA=axes2.transData,coordsB=ax.transData)
    axes2.add_artist(conn)

  ax.set_xlabel("X")
  ax.set_ylabel("Y")
  ax.set_zlabel("Z")
  ax.legend()
  plt.show()

plot_3D_points(img1,img2,selected_img_world_pts,[])
# plot_3D_points(img1,img2,selected_img_world_pts,reconstruct_world_pts_refined)



"""Dense Stereo Matching"""

def census_transform(img1,img2,M,d_max):
    census_map = np.zeros_like(img1)
    half_M = M//2
    img1_pad = np.pad(img1,half_M)
    img2_pad = np.pad(img2,half_M)

    for i in range(img1.shape[0]):
```

```
        for j in range(img1.shape[1]):
            # img1_window = img1_pad[i-half_M:i+half_M+1, j-half_M:j+half_M+1]
            img1_window = img1_pad[i:i+M, j:j+M]
            img1_window_vec = (img1_window > img1[i,j]).astype(np.uint8).flatten()
            cost_list = []
            if j-d_max<0:
                d_max_eff = j
            else: d_max_eff=d_max
            for d in range(d_max_eff+1):
                # img2_window = img2_pad[i-half_M:i+half_M+1, j-d-half_M:j-d+half_M+1]
                img2_window = img2_pad[i:i+M, j-d:j-d+M]
                img2_window_vec = (img2_window > img2[i,j-d]).astype(np.uint8).flatten()
                XOR_vec = img1_window_vec ^ img2_window_vec
                cost_list.append(sum(XOR_vec))
            d_pick = np.argmin(cost_list)
            census_map[i,j] = d_pick

    return census_map.astype(np.uint8)

# Load images and ground truth disparity maps
left_img = cv.imread('/content/im2.png', cv.IMREAD_GRAYSCALE)
right_img = cv.imread('/content/im6.png', cv.IMREAD_GRAYSCALE)
left_ground_truth = cv.imread('/content/disp2.png', cv.IMREAD_GRAYSCALE)
left_ground_truth = (left_ground_truth.astype(np.float32) / 4).astype(np.uint8)
right_ground_truth = cv.imread('/content/disp6.png', cv.IMREAD_GRAYSCALE)
right_ground_truth = (right_ground_truth.astype(np.float32) / 4).astype(np.uint8)

d_max = np.max(left_ground_truth)

window_sizes = [3, 7, 9, 13, 15]
disparity_maps_left = []
disparity_maps_right = []
error_maps_left = []
error_maps_right = []

for window in window_sizes:
    d_map = census_transform(left_img,right_img,window,d_max)
    disparity_maps_left.append(d_map)

delta = 2
for i,disparity_map in enumerate(disparity_maps_left):
    err = np.abs(disparity_map - left_ground_truth)
    binary_err_mask = np.where(err<=delta,255,0)
    N = np.sum(disparity_map>0)
    accuracy = np.sum(err[disparity_map>0]<delta)/N
    print(f'Accuracy: {accuracy} for window_size: {window_sizes[i]}')
    fig,axes = plt.subplots(1,2)
    plt.suptitle(f'window size : {window_sizes[i]}',y=0.8)
    axes[0].imshow(disparity_map,cmap='gray')
    axes[0].set_title('disparity map')
    axes[1].imshow(binary_err_mask,cmap='gray')
    axes[1].set_title('binary error map')
    plt.show()
```

```
# %% [markdown]
# Depth Map and Automatic Extraction of Dense Cor
# respondences

# %%
import numpy as np
import pickle as pkl
import matplotlib.pyplot as plt
import h5py # for reading depth maps

DEPTH_THR = 0.1

# %%
def plot_image_and_depth(img0, depth0, img1, depth1, plot_name):
    # Enable constrained layout for uniform subplot sizes
    fig, ax = plt.subplots(1, 4, figsize=(20, 5), constrained_layout=True)

    # Image 0
    ax[0].imshow(img0, aspect='auto')
    ax[0].set_title('Image 0')
    ax[0].axis('off')

    # Depth 0
    im1 = ax[1].imshow(depth0, cmap='jet', aspect='auto')
    ax[1].set_title('Depth 0')
    ax[1].axis('off')
    cbar1 = fig.colorbar(im1, ax=ax[1], shrink=0.8, aspect=20)
    cbar1.ax.yaxis.set_ticks_position('left')
    cbar1.ax.yaxis.set_label_position('left')
    cbar1.ax.tick_params(labelsize=15)

    # Image 1
    ax[2].imshow(img1, aspect='auto')
    ax[2].set_title('Image 1')
    ax[2].axis('off')

    # Depth 1
    im2 = ax[3].imshow(depth1, cmap='jet', aspect='auto')
    ax[3].set_title('Depth 1')
    ax[3].axis('off')
    cbar2 = fig.colorbar(im2, ax=ax[3], shrink=0.8, aspect=20)
    cbar2.ax.yaxis.set_ticks_position('left')
    cbar2.ax.yaxis.set_label_position('left')
    cbar2.ax.tick_params(labelsize=15)

    plt.savefig(plot_name, bbox_inches='tight', pad_inches=0)
    plt.close()


# %%
scene_info = pkl.load(open('./data/scene_info/1589_subset.pkl', 'rb'))

for i_pair in range(len(scene_info)):
    # i_pair = 7

    # print(scene_info[i_pair].keys())
    # ['image0','image1','depth0', 'depth1', 'K0', 'K1', 'T0', 'T1', 'overlap_score']
    # print(scene_info[i_pair]['image0']) # path to image0
    # print(scene_info[i_pair]['image1']) # path to image1
    # print(scene_info[i_pair]['depth0'])  # path to depth0
    # print(scene_info[i_pair]['depth1'])  # path to depth1
    # print(scene_info[i_pair]['K0'])  # intrinsic matrix of camera 0 [3,3]
    # print(scene_info[i_pair]['K1'])  # intrinsic matrix of camera 1 [3,3]
    # print(scene_info[i_pair]['T0'])  # pose matrix of camera 0 [4,4]
    # print(scene_info[i_pair]['T1'])  # pose matrix of camera 1 [4,4]
    # print('--------------------')

    # read images
    img0 = plt.imread(scene_info[i_pair]['image0'])
    img1 = plt.imread(scene_info[i_pair]['image1'])

    # read depth
    with h5py.File(scene_info[i_pair]['depth0'], 'r') as f:
        depth0 = f['depth'][:]
    with h5py.File(scene_info[i_pair]['depth1'], 'r') as f:
        depth1 = f['depth'][:]

    # check shapes
    h0, w0 = img0.shape[:-1]
```

```
h1, w1 = img1.shape[:-1]
assert img0.shape[:-1] ==  depth0.shape, f"depth and image shapes do not match: {img0}, {depth0}"
assert img1.shape[:-1] ==  depth1.shape, f"depth and image shapes do not match: {img1}, {depth1}"

# plot image and depth
plot_name = f'./pics/image_and_depth_pair_{i_pair}.png'
plot_image_and_depth(img0, depth0, img1, depth1, plot_name)


#(1) make meshgrid of points in image 0
x = np.linspace(10, img0.shape[1]-10, 10) # ignore a border of 10 pxls
"""
<Student code>
# meshgrid of x and y coordinates
xx, yy = ...
"""
y = np.linspace(10,img0.shape[0]-10, 10)
xx, yy = np.meshgrid(x,y)

# make homogeneous coordinates for points0 #[3, N]
"""
<Student code>
points0 = ...
"""
points0 = np.vstack((xx.flatten(),yy.flatten(),np.ones(len(xx.flatten())))).astype(int)

#(2) get depth values at points0
"""
<Student code>
depth_values0 = ...
"""
depth_values0 = depth0[points0[1],points0[0]]
# remove points with depth 0 (invalid points)
"""
<Student code>
valid_points = ...
# mask points0 and depth_values0
"""
valid_points = depth_values0 > 0
points0 = points0[:,valid_points]
depth_values0 = depth_values0[valid_points]

# (3) Find the 3D coordinates of these points in camera 0 frame
K0 = scene_info[i_pair]['K0'] # [3,3]
T0 = scene_info[i_pair]['T0'] # [4,4]
"""
<Student code>
# inverse of K0
K0_inv = ...
# convert points0 to camera coordinates
xyz_cam0 = ...
# normalize xyz_cam0 to set z = 1 (sanity check)
xyz_cam0 = ...
# get the point at depth
xyz_cam0 = ...
# make homogeneous coordinates [4,N]
xyz_cam0_hc = ...
# convert to world frame [4,N]
xyz_world_hc = ...
"""
K0_inv = np.linalg.inv(K0)
xyz_cam0 = K0_inv@points0
xyz_cam0 = depth_values0*xyz_cam0
xyz_cam0_hc = np.vstack((xyz_cam0,np.ones(xyz_cam0.shape[1])))
xyz_world_hc = np.linalg.inv(T0)@xyz_cam0_hc


# (4) Transform these points to camera 1 frame
T1 = scene_info[i_pair]['T1']
"""
<Student code>
# transform points to camera 1 frame
xyz_cam1_hc = ...
# convert to camera 1 coordinates
xyz_cam1 = ...
# get z coordinates for depth check
estimated_depth_values1 = ...
"""
xyz_cam1_hc = T1@xyz_world_hc
```

```
xyz_cam1 = xyz_cam1_hc[:-1,:]
estimated_depth_values1 = xyz_cam1[-1,:]


# project to image 1
"""
<Student code>
points1 = ... # [3, N]
# normalize by dividing by last row
points1 = ... # [3, N]
# check if points1 are within image bounds
...
# get the depth values at these points using the depth map
true_depth_values1 = ...
"""
K1 = scene_info[i_pair]['K1']
points1 = K1@xyz_cam1
points1 /= points1[-1]
valid_idx = (0<=points1[0]) & (points1[0]<img1.shape[1]) & (0<=points1[1]) & (points1[1]<img1.shape[0])
points1 = (points1[:,valid_idx]).astype(int)
estimated_depth_values1 = estimated_depth_values1[valid_idx]
true_depth_values1 = depth1[points1[1],points1[0]]


# (5) plot matching points in image 0 and image 1 with depth check such that the depth values match
fig, ax = plt.subplots(1, 1, figsize=(10, 5))

# Horizontally stack the images
combined_img = np.ones((max(img0.shape[0], img1.shape[0]), img0.shape[1] + img1.shape[1], 3), dtype=np.uint8) * 255
combined_img[:img0.shape[0], :img0.shape[1]] = img0
combined_img[:img1.shape[0], img0.shape[1]:] = img1

ax.imshow(combined_img, aspect='auto')
ax.scatter(xx, yy, c='r', s=5)
ax.set_title('Matching points in Image 0 and Image 1')
ax.axis('off')

# draw lines between matching points
for i in range(points1.shape[1]):
    # if depth values match
    if np.abs(estimated_depth_values1[i] - true_depth_values1[i]) < DEPTH_THR and true_depth_values1[i] != 0:
        ax.plot([points0[0,i], points1[0, i] + img0.shape[1]], [points0[1,i], points1[1, i]], 'g')

plt.savefig(f'./pics1/depth_check_pair_{i_pair}.png', bbox_inches='tight', pad_inches=0)
plt.close()
print(f"Done with pair {i_pair}")

# (6) Plot all 3D points for the pair
"""
<Student code>
...
"""
x = np.linspace(10,img1.shape[1]-10, 100)
y = np.linspace(10,img1.shape[0]-10, 100)
xx, yy = np.meshgrid(x,y)
points1 = np.vstack((xx.flatten(),yy.flatten(),np.ones(len(xx.flatten())))).astype(int)
depth_values1 = depth1[points1[1],points1[0]]
valid_points = depth_values1 > 0
points1 = points1[:,valid_points]
depth_values1 = depth_values1[valid_points]
K1 = scene_info[i_pair]['K1'] # [3,3]
T1 = scene_info[i_pair]['T1'] # [4,4]
K1_inv = np.linalg.inv(K1)
xyz_cam1 = K1_inv@points1
xyz_cam1 = depth_values1*xyz_cam1
xyz_cam1_hc = np.vstack((xyz_cam1,np.ones(xyz_cam1.shape[1])))
xyz_world_hc_1 = np.linalg.inv(T0)@xyz_cam1_hc
xyz_world_hc_1 /= xyz_world_hc_1[-1]

xyz_world_hc /=xyz_world_hc[-1]
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xyz_world_hc[0,:], xyz_world_hc[1,:], xyz_world_hc[2,:],s=1,label='0')
            # c=xyz_world_hc[2,:], cmap='jet', s=1)
ax.scatter(xyz_world_hc_1[0,:], xyz_world_hc_1[1,:], xyz_world_hc_1[2,:],s=1,label='1')
ax.set_title("3D World Points", fontsize=15,y=0.97)
ax.legend()
ax.set_xlabel("X", fontsize=12)
ax.set_ylabel("Y", fontsize=12)
```

```
    ax.set_zlabel("Z", fontsize=12)
    ax.view_init(elev=90, azim=90)
    plt.savefig(f'./pics2/3D_World_points_{i_pair}.png', bbox_inches='tight', pad_inches=0)
    plt.close()



# %%
```