Homework 9

Theory Question

Since the absolute conic is the same for both stereo images, as it is described only by the intrinsic parameters and not the extrinsics, then we can relate the image of the absolute conic. Take an epipolar line that is tangent to the image of the absolute conic in the left image. The point at which they "touch", lets call it x. Once we project this point to the plane at infinity, then this point will also lie on the absolute conic, and now we project this point to the right image. This point will lie on the image of the absolute conic and on a right epipolar line, similarly to how it was for the left image. The plane generated from the left epipolar line backprojected into the absolute conic is tangent to the absolute conic and thus tangent to the image of the absolute conic on the right image. Thus we can make a correspondence between the left epipolar line and the right epipolar line, call it epipole prime. Then since this point lies on line l, the image of it on the right image will lie on line l prime.

Task 1: Projective Stereo Reconstruction

Image Rectification

We start off with the 8 point correspondence method to be able to compute an estimation for $\mathbf{F}$. This consists of representing the follow equation as a linear system:

$$\mathbf{x}'\mathbf{F}\mathbf{x} = \mathbf{0} \tag{1}$$

Where the fundamental matrix, $\mathbf{F}$ is given by:

$$\mathbf{F} = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix}$$

Expanding Eq. 1 and putting it into matrix representation for us to use with linear least squares results in:

$$\begin{pmatrix} x_1'x_1 & x_1'x_2 & x_1' & x_2'x_1 & x_2'x_2 & x_2' & x_1 & x_2 & 1 \end{pmatrix} \begin{pmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{pmatrix} = 0$$

Since we have 8 correspondences, we can stack the matrices we get from all of them to solve with linear least squares. However, there is one important step that needs to be done before

this. We need to normalize the data points before doing this. The way we go about it is we shift the centroid of these data points to $(0, 0)$, and then calculate the distance from the shifted points to the new origin, from that we get the scale, given by $\sqrt{2}/d$ where d is the average distance. We then get a normalization matrix:

$$\mathbf{T} = \begin{pmatrix} s & 0 & -sc_x \\ 0 & s & -sc_y \\ 0 & 0 & 1 \end{pmatrix}$$

Where s is the scale, c_x, c_y are the x and y coordinates of the centroid respectively. Our points are then transformed according to their matrices:

$$\begin{aligned} \mathbf{x}_{normalized} &= \mathbf{T}\mathbf{x} \\ \mathbf{x}'_{normalized} &= \mathbf{T}'\mathbf{x}' \end{aligned} \tag{2}$$

We finally solve for $\mathbf{F}$ with linear least squares. We normalize this by dividing it by its very last variable, f_{33}. Now we need to ensure that the rank of this matrix is 2, we do so by doing an SVD decomposition, and zero-ing out the smallest eigenvalue. We then put it back together, and multiply it by our normalization matrices in the following manner:

$$\mathbf{F}_{conditioned} = \mathbf{T}'^{\top}\mathbf{F}\mathbf{T} \tag{3}$$

With our fundamental matrix we can now get the epipoles, which are just the left and right side of the null space, which we can get by getting the last column of v and the last row of u from the SVD decomposition of $\mathbf{F}$. Since these vectors will be in homogeneous coordinates, we divide by the last value.
Now we can calculate our $\mathbf{P}'$. We do this by setting $\mathbf{P} = [\mathbf{I}_{3\times3}|\mathbf{0}]$, getting the skew symmetric matrix of e', then:

$$\mathbf{P}' = [\mathbf{e}']_{\times}\mathbf{F} \tag{4}$$

We then set the last column of $\mathbf{P}'$ to be $\mathbf{e}'$. Now with $\mathbf{P}'$ we can proceed to use triangulation to arrive at the world coordinates for our 8 points. The world coordinates will be given by:

$$\begin{aligned} \mathbf{x} &= \mathbf{P}\mathbf{X} \\ \mathbf{x}' &= \mathbf{P}'\mathbf{X} \end{aligned} \tag{5}$$

We can express both of these in the form of a homogeneous system of equations $\mathbf{A}\mathbf{X} = \mathbf{0}$. Where $\mathbf{A}$ will be given by:

$$\mathbf{A} = \begin{pmatrix} x\mathbf{P}^{3\top} - \mathbf{P}^{1\top} \\ y\mathbf{P}^{3\top} - \mathbf{P}^{2\top} \\ x'\mathbf{P}'^{3\top} - \mathbf{P}'^{1\top} \\ y'\mathbf{P}'^{3\top} - \mathbf{P}'^{2\top} \end{pmatrix} \tag{6}$$

Solving this linear least squares problem will result in the first guess for the world coordinates for our correspondences. Now, we can use Levenberg Marquadt to optimize both $\mathbf{P}'$ and the world points to reduce the error that we can get from Eq. 5, since we have the image

coordinates. We also make sure that since we are refining 36 variables, 12 from $\mathbf{P}'$ and 24 from the 8 world coordinates, our error array needs to length 36.
With our now optimized $\mathbf{P}'$ we can now calculate our right homography. The process for this is to first get a translation matrix that will shift our image center to the origin, and also one that shifts the origin to the image center. These will have the following shape:

$$\mathbf{T}_1 = \begin{pmatrix} 1 & 0 & -\frac{w}{2} \\ 0 & 1 & -\frac{h}{w} \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{T}_2 = \begin{pmatrix} 1 & 0 & \frac{w}{2} \\ 0 & 1 & \frac{h}{w} \\ 0 & 0 & 1 \end{pmatrix}$$

And we also have a rotation matrix given by:

$$\mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Where θ is given by:

$$\arctan -\frac{e_2 - \frac{h}{2}}{e_1 - \frac{w}{2}}$$

Where h and w are the image height and width, respectively. There is one more transformation we need to apply, one that will shift the epipole to $(1, 0, 0)$. This is given by

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{pmatrix} \tag{7}$$

Where, from the previous transformations, we can find f.

$$f = (e_1 - \frac{w}{2})\cos\theta - (e_2 - \frac{h}{2})\sin\theta$$

Finally, our right homography is given by:

$$\mathbf{H}_{right} = \mathbf{T}_2\mathbf{G}\mathbf{R}\mathbf{T}_1 \tag{8}$$

Now with our right homography we can calculate the left one. First we get the $\mathbf{M}$ matrix, given by $\mathbf{M} = \mathbf{P}'\mathbf{P}^{-1}$. Now, we can set our $\mathbf{H}_0 = \mathbf{H}_{right}\mathbf{M}$. Now we have to build yet another linear least squares to minimize the distance between our transformed points, in the rectified images, by knowing that what is missing from our left homography is a transformation given by:

$$H_1 = \begin{pmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

With our linear least square system defined by

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = x' \tag{9}$$

For each of our 8 points, which we can stack together. After solving this system, we set $\mathbf{H}_{left} = H_1 H_0$. Now all that is left is rectify our images using the homographies found.
Our image rectification process starts by transforming the corners of our left and right image first, and choosing the minimum and maximum x and y values from these corners to know what the new size of our rectified images and our x and y offset will be. We use the same offset for both and also the same size.

Interest Point Detection

We first apply Canny on the non rectified images, then rectify them using the previously computed $\mathbf{H}_{left}$ and $\mathbf{H}_{right}$. This is to not get the edges are the new borders are the rectification. We then extract all the positions where there is an edge in the left image. Using each of these positions, we search the same row in the right image, comparing both windows with SSD. The SSD we employ is on all 3 channels, which we calculate the SSD per channel and then average them out. We then take the top match, verifying that the column is within certain distance or we reject that correspondence. We then use project these correspondences into our original images by using the inverse of the rectifying homographies.

Projective Reconstruction

We take these correspondences and use triangulation again, and proceed to optimize using LM, we then graph these 3D points.

3D Visual Inspection

We follow the instructions from Homework 8 to get the Camera center and the camera axis directions, we plot it together with the points.

Task 2: The Loop and Zhang Algorithm

The main idea behind the Loop and Zhang algorithm is that we don't have to use calibrated cameras, but only the fundamental matrix, which arises from the geometric relationship of the images, is needed, also it is fully automatic, so it requires no input aside from the images. Another fundamental idea to this algorithm is that it separates the rectifying homography $\mathbf{H}$ into a projective, affine and a similarity transformation. The affine part is also called the shearing transformation.

Task 3: Dense Stereo Matching

Census Transform

The idea behind dense stereo matching algorithm using the Census Transform is that, as we can see in the images, due to the difference in camera position, one is a "shifted" version of the other. If we can find a descriptor for each of the pixels we can then match a pixel position from one image to a pixel position from the other image. This is where we make use of Census Transform, which will take in a window of size $M \times M$ with a pixel at the center, and compute a bitvector composed of 0s or 1s, depending on whether the pixel in the window is greater than the middle one or not. We now start looking for a possible match, since we know (in the case of using the left image and searching a match in the right) that the right image is the left one shifted to the right), we consider this shift as well. We get all the bitvectors from the same position, up to the position - d_{Max}, which is the maximum displacement we consider, and compare all these bitvectors to our initial descriptor through the use of a logical XOR operation, which we then sum all the 1s in this. The corresponding match will be given by the position that resulted in the lowest sum. We now consider this position to be the displacement at the given pixel position. This displacement map can also be used for depth estimation as objects that are closer to the camera will seem to shift more than objects farther away, so if a value in the displacement map is higher, that is because the object is closer to the camera.

Task 4: Depth Map and Automatic Extraction of Dense Correspondences

The way dense correspondences work is by extracting a number of points from the image. Then, using the camera intrinsics, these points are projected into the camera coordinate system. It is in here where we make use of the depth maps, and multiply each of these points in the camera coordinate system by the depth value at the corresponding image points. Now that the depth is correct, we transform these points from the camera coordinates to the world coordinates using the camera extrinsics. Now that the points from image 1 are in the world coordinates, we can project these into image 2. The way we do it is to start from world coordinates, and transform into camera 2 coordinate system by making use of camera 2 extrinsics, and save the 2nd column/row of this (the z values). Then normalize these points by dividing by the last column. These normalized points are now transformed to be the points in image 2 using the camera 2 intrinsics. Now we can check that the depth values from image 2 at these points match the depth values we estimated, this is why we saved that column. If the difference is lower than a certain threshold, we can say that the points from image 1 and the projected points into image 2 are a match.
This is useful as we can make correspondences between images by just knowing the depth maps, and using this information we can make accurate 3D reconstructions of the scene with little complications, as we witnessed during the 2nd part of this task. And it is these reconstructions that are useful in areas like robotics.

Results: Task 1

Image Rectification

The stereo pair with the correspondences used is shown in 1. After finding the rectifying homographies, we used a similar approach to previous homeworks, that we computed the new size of the image after the transformation, and also the x and y offsets, and used both left and right at the same time to build our new rectified images. Once we rectified the image, the now notice that it works since the rows that the points are on are the same for both the left and the right images, there is only a small difference in the columns they are in (y is the same, x is different). The rectified pair with the correspondences is shown in Fig. 2.

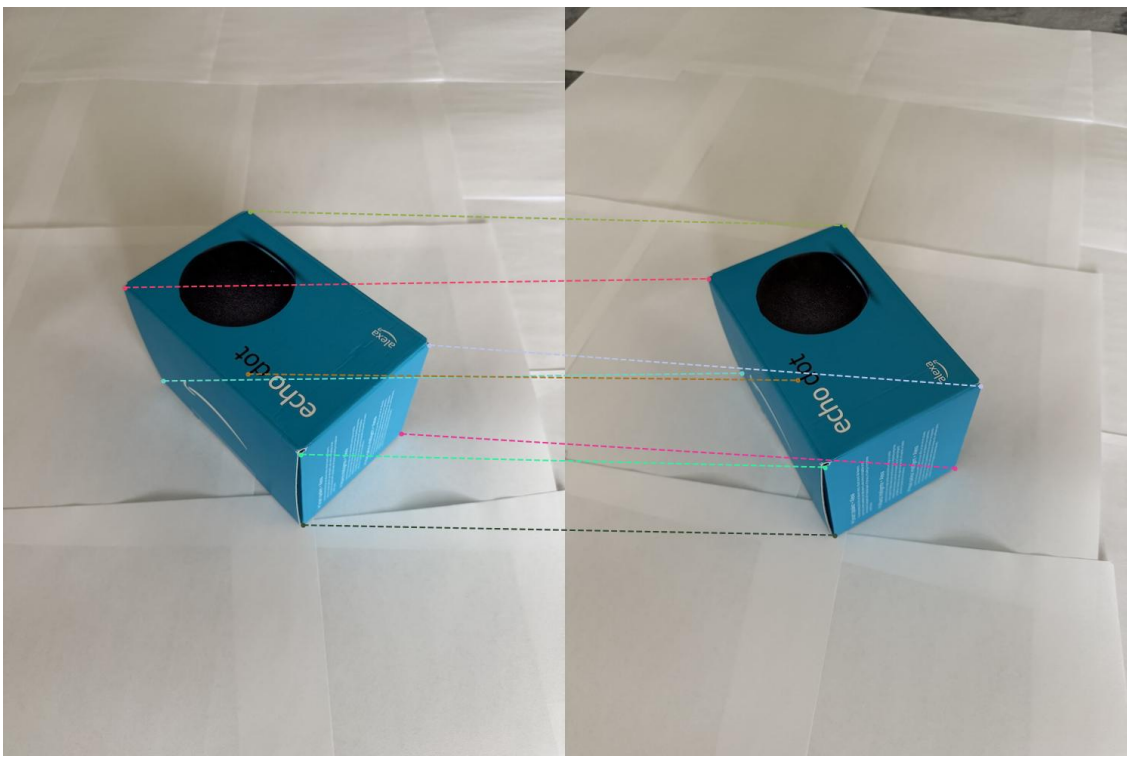


Figure 1: Stereo pair with correspondences.

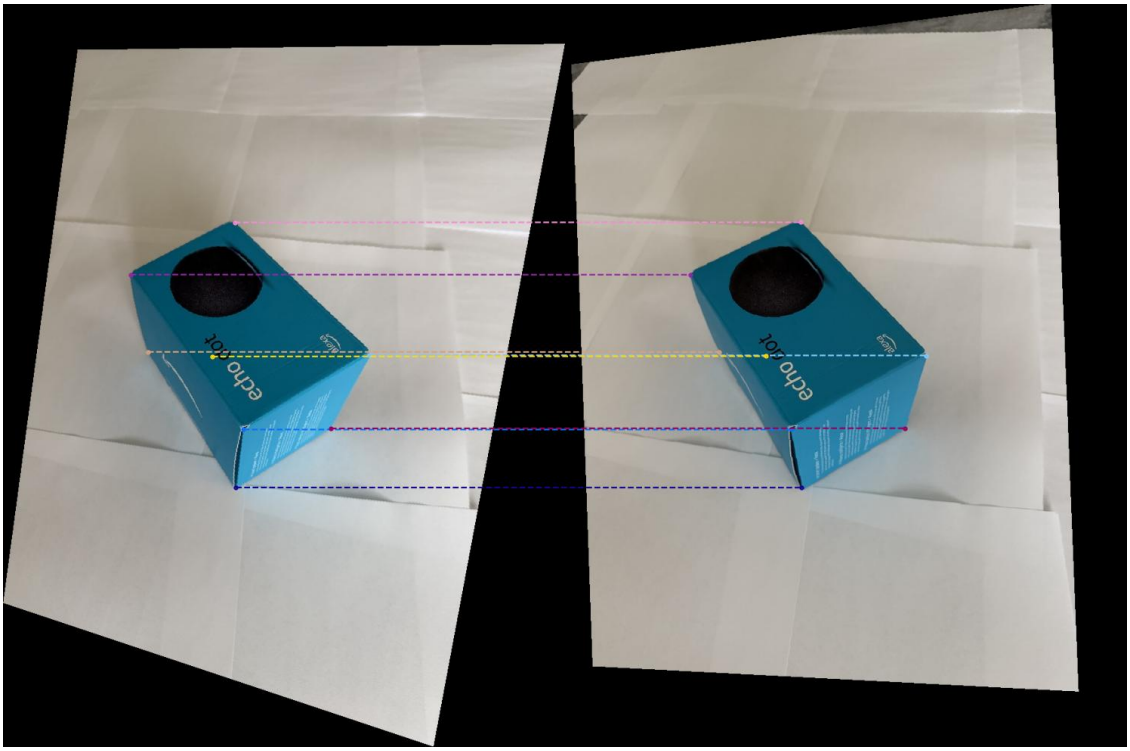


Figure 2: Rectified pair with correspondences.

Interest Point Detection

As described before, we got the Canny edges first, then rectified them, this is so we did not get the border of the image as part of the Canny edges. After this we follow the procedure described in the first task section. Our rectified Canny edges is shown in Fig. 3. We extract the edge pixel locations and use them in our rectified images and compute the matches using SSD for all 3 RGB channels which are then averaged out. The resulting matches are shown in Fig. 4. However these are of not much use for use for the next task, so we project the points back into the original, non-rectified images, which are shown in Fig.

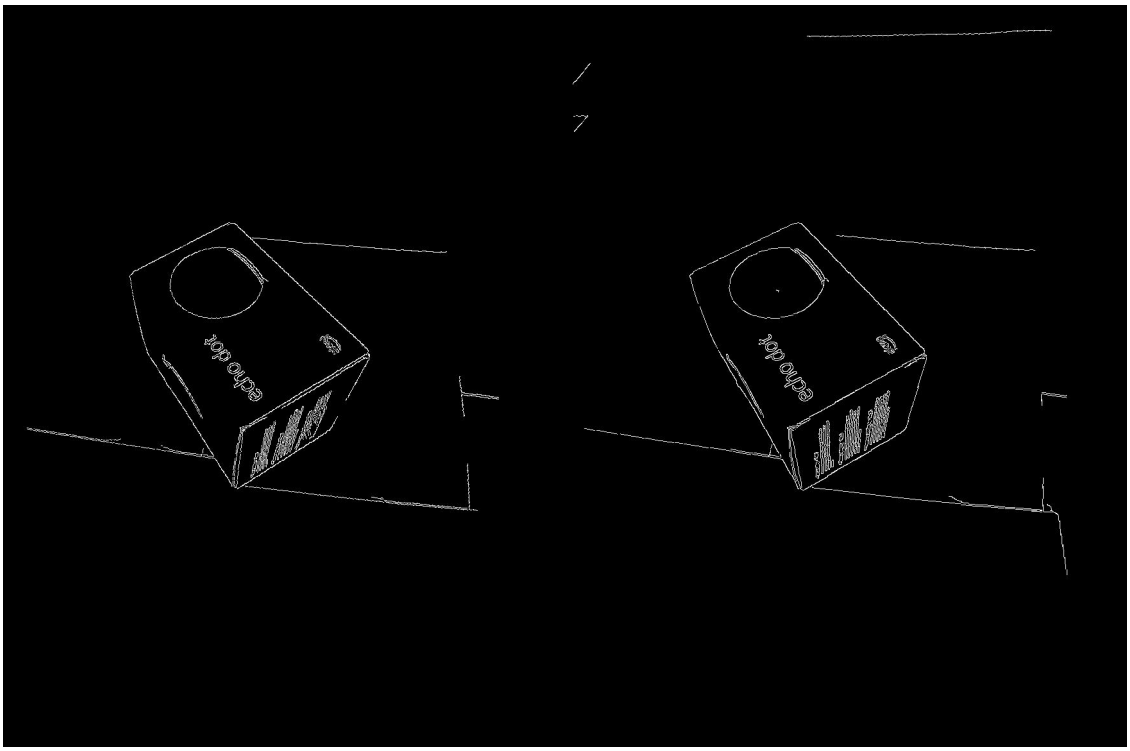


Figure 3: Canny edges both both left and right images.

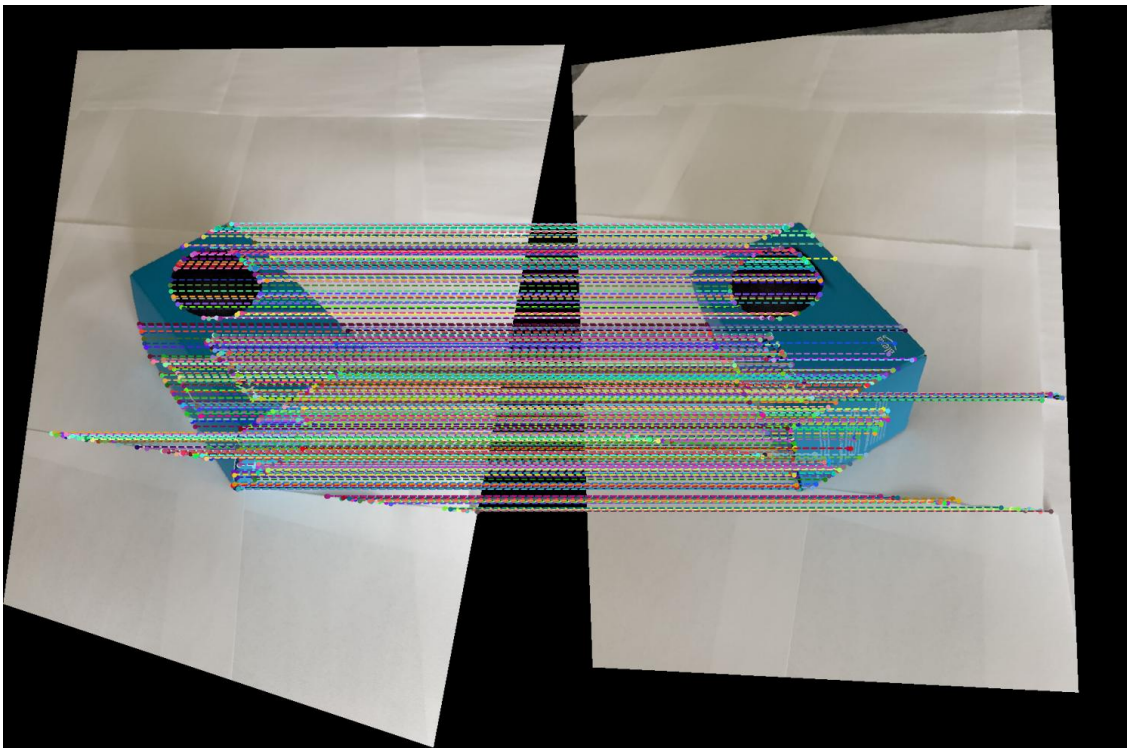


Figure 4: Large number of correspondences obtained from Canny edges.

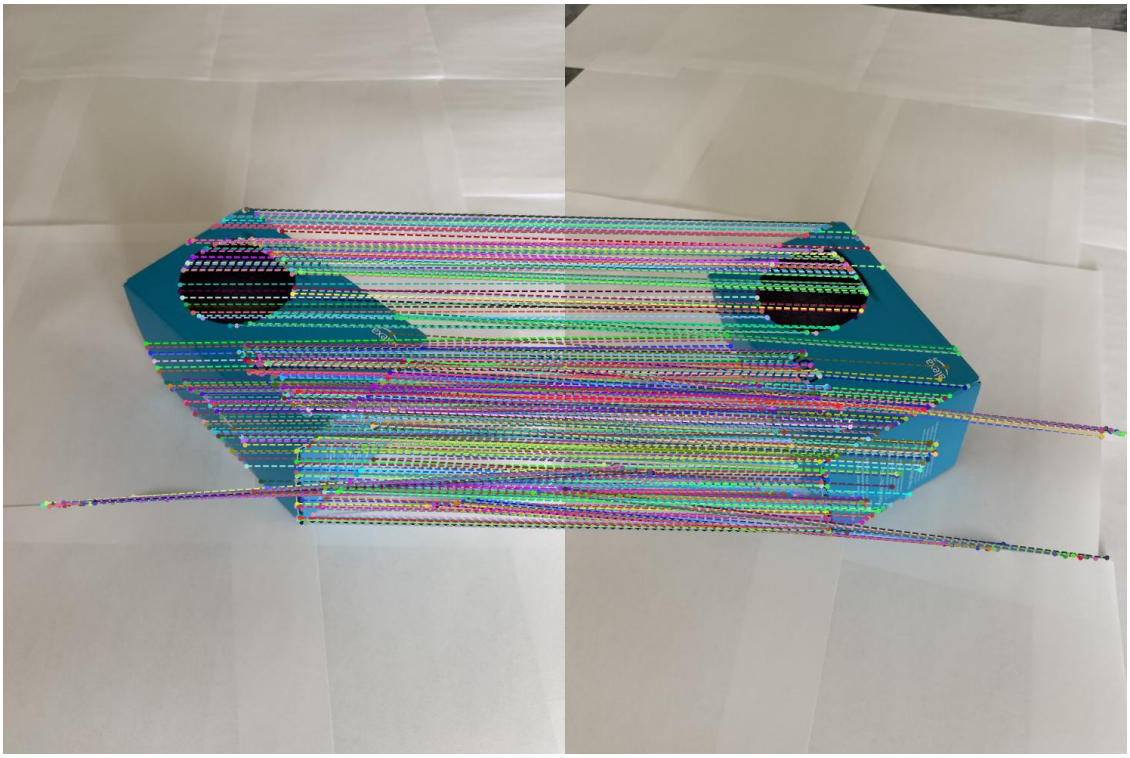


Figure 5: Large number of correspondences obtained from Canny edges in the non-rectified pair.

Projective Reconstruction & 3D Visual Inspection

We projected the points into 3D using the projection matrices and also triangulation, however, this 3D reconstruction is a bit hard to understand. Shown in Fig. 6 we have the cameras, the distance from the left and right has been a bit exaggerated to notice this difference. Now, in Fig. 7, we have first view together with the triangulated Canny correspondences and the original correspondences. We can see how the image looks to be stretched in a particular direction, with the points near "echo" and on the right top edge are further apart from all the other points. Looking at the YZ plane, in Fig. 8, we can see that this distortion also has stretched more in a particular direction, seeing as how the pink and red points are also far apart. Fig. 9 shows a zoomed in view and it seems they have been flipped, as the right points are in the left and viceversa.

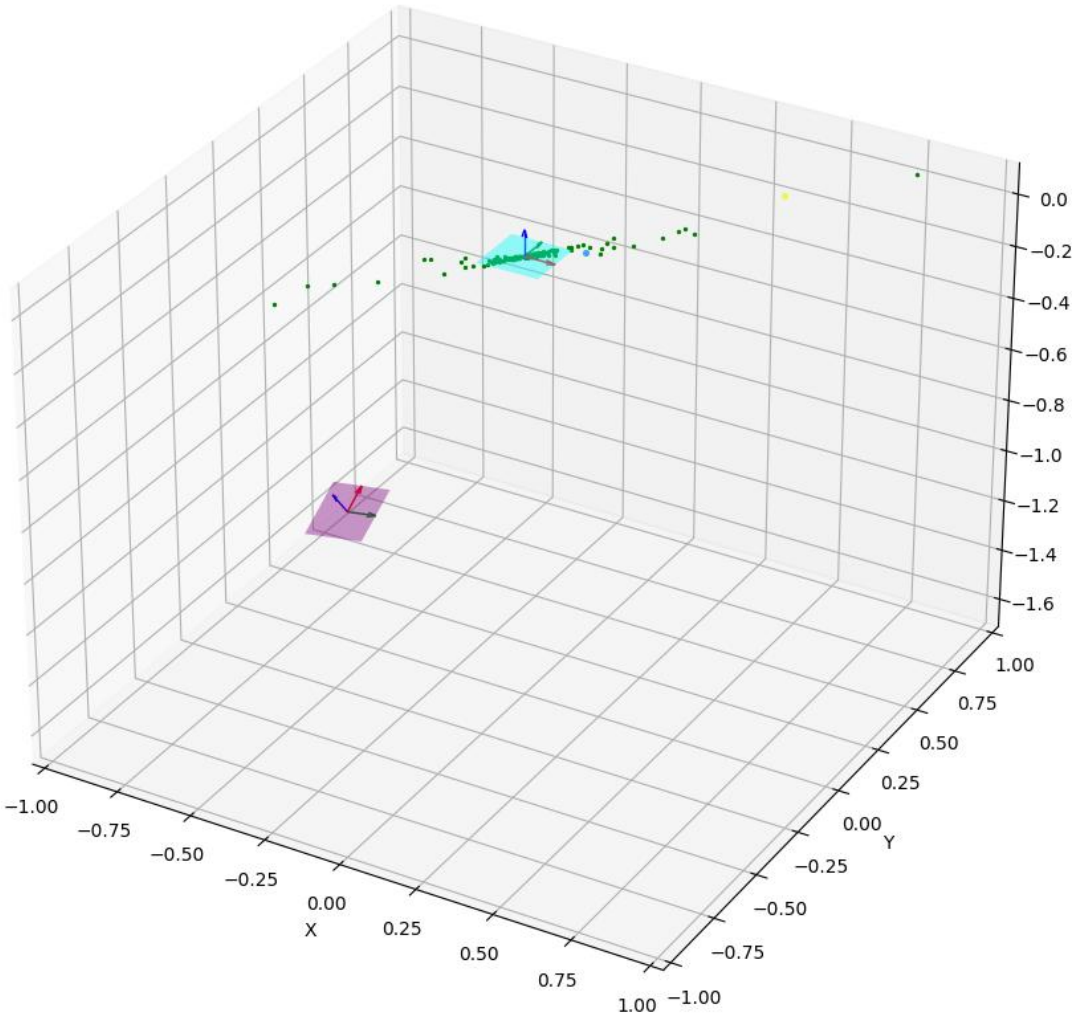


Figure 6: Some of the points with the cameras, cyan square is the left camera and magenta is for the right camera.

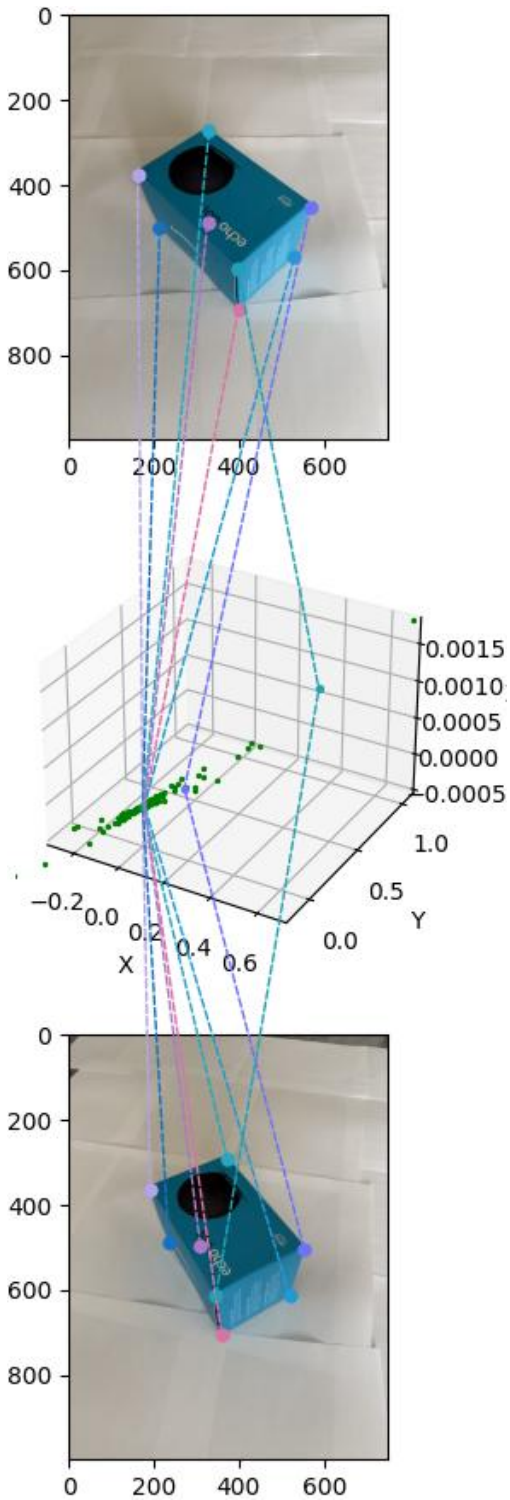


Figure 7: View 1 of the 3D reconstruction.

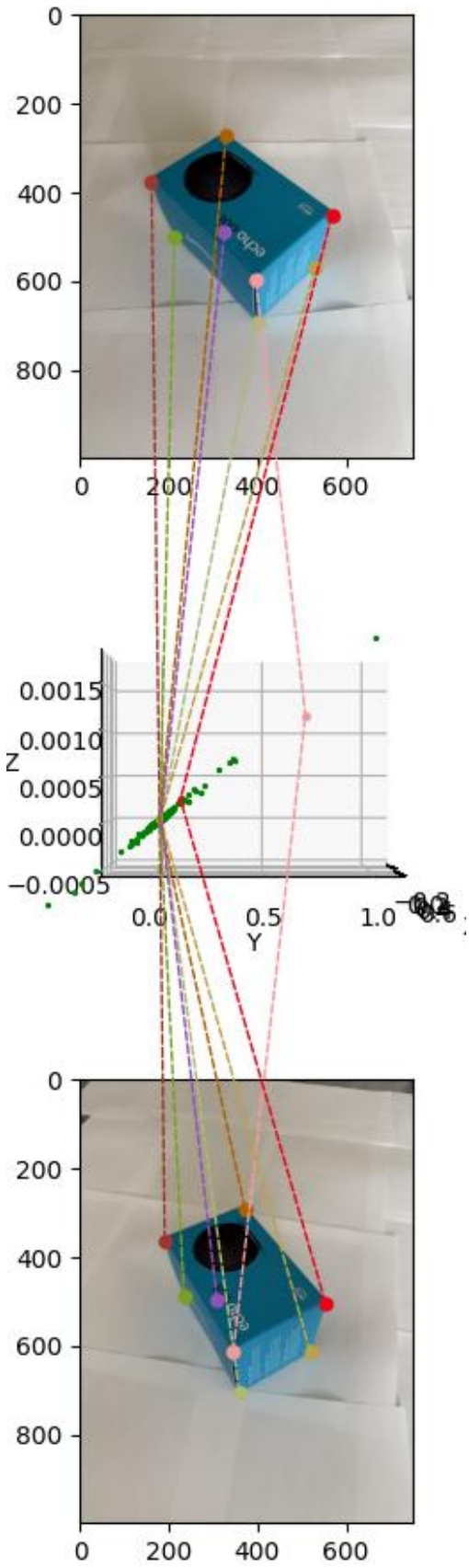


Figure 8: View 2 of the 3D reconstruction.

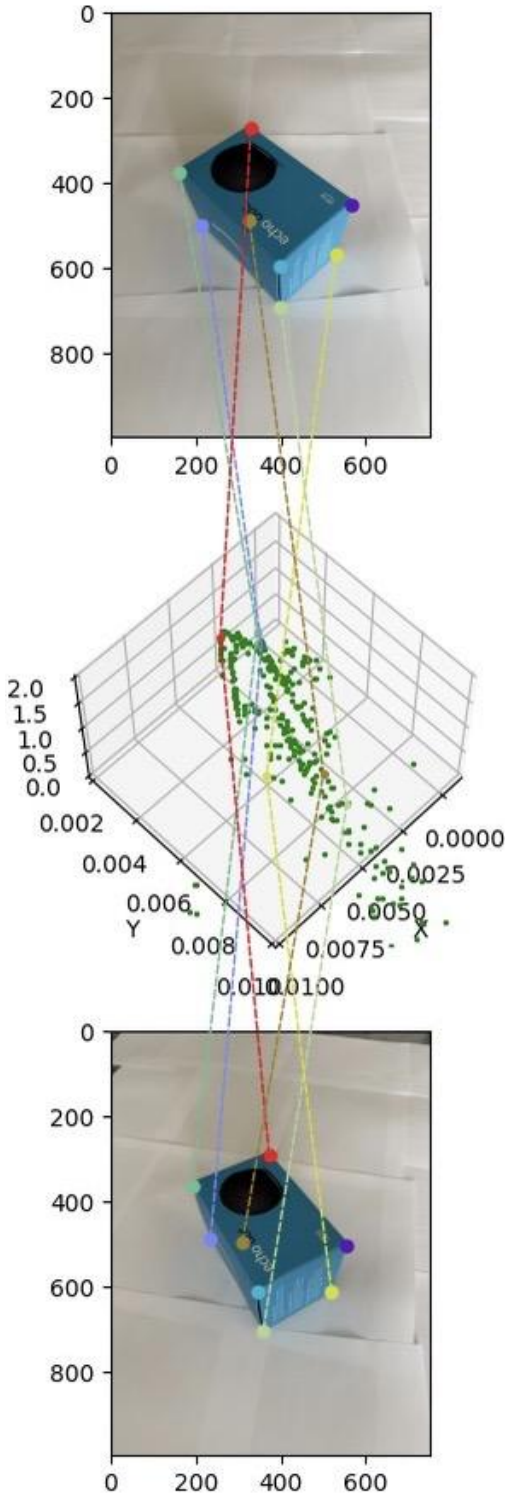


Figure 9: View 3 of the 3D reconstruction.

Results: Task 2

The performance of Loop & Zhang on the stereo pair was not what we expected. The transformations for both are very different, for the right images it looks like Loop & Zhang has calculated an almost "exaggerated" version of our own, as there are some similarities. It can be hard to tell but it does not look like the epipole is in a similar angle in both images. It also discarded part of the left image, so that is also different from what our implementation does. The object, which is the Echo dot box, does seem to be similar in both images, but it can be hard to notice when the images are not the same size. Overall, it seems like our implementation of the 8 point rectification algorithm has produced better results than the Loop & Zhang algorithm, particularly in this pair where Loop & Zhang has struggled.

(a) Rectified left image.

(b) Rectified right image.

Figure 10: Loop and Zhang algorithm outputs for the stereo pair.

Results: Task 3

First, we obtain the value of d_{max} from the ground truth disparity map. We follow the instructions and convert to float32, divide by 4 then convert back to int8. We obtained that the d_{max} values for left and right are 55 and 54 respectively. We also need to consider that, even though the images are RGB, we are only interested in using them as grayscale.

Census Transform

So, we start off by padding both images, only considering the window size for the top and bottom padding and also our new border size accounting for d_{max}. We then perform the census transform and matching as explained previously. Our disparity map is then compared to the ground truth, we get the difference between the disparity values and we create a mask where we give the position 255 if the difference is less than δ. We set this threshold value $\delta = 2$. Then to get the accuracy, we count the number of 255 from the difference mask, and divide it by the number of valid positions from the ground truth disparity map. We tried with 4 different window sizes: 5, 15, 25, and 35. We also implemented the matching algorithm for the right image. The only difference is that now we search in the opposite direction, before it was to the left, now we must do it to the right. The results for the left and right different window sizes are shown in Figs. 11 and 12 respectively. The accuracies are shown in Tables 1 and 2 for the left and right disparity map accuracies respectively. The error masks are shown in Figs. 13 and 14.

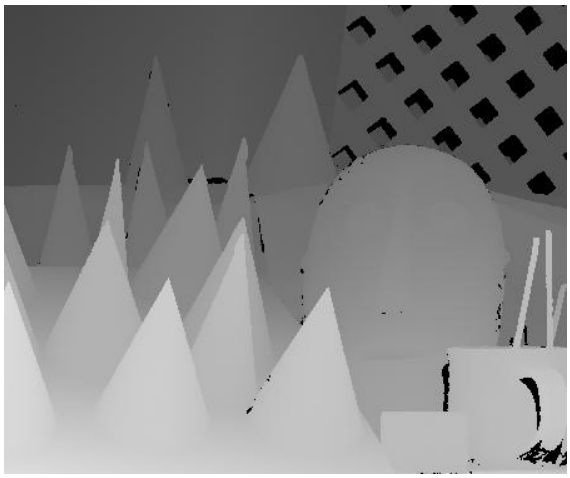
Window Size	Accuracy
5	0.5235517783995934
15	0.7589532270804122
25	0.7597063451730028
35	0.747962601257646

Table 1: Accuracy from the calculated left disparity map.

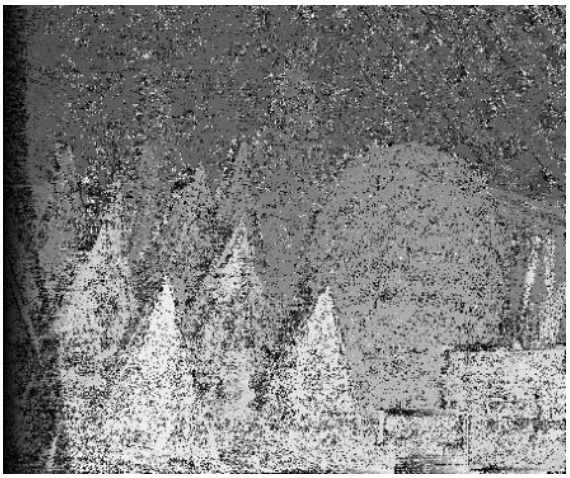
Window Size	Accuracy
5	0.525274549787485
15	0.7598457116183083
25	0.7631992727808761
35	0.7514740928187111

Table 2: Accuracy from the calculated right disparity map.

(a) Original left image.

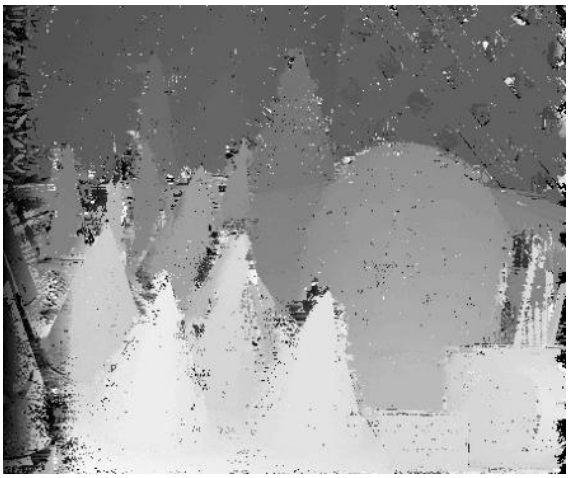


(b) Ground truth disparity map

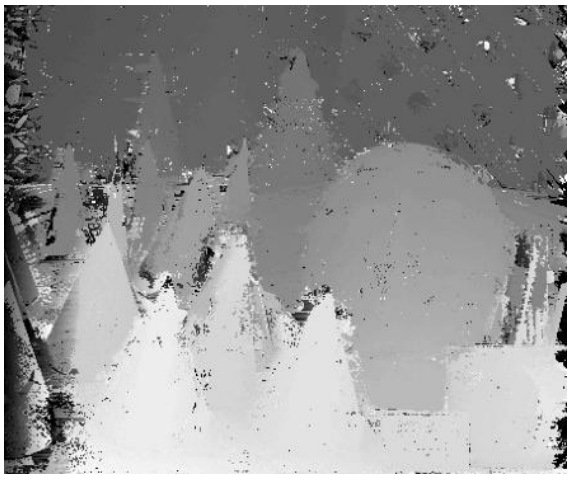


(c) Calculated disparity map with window size of 5.

(d) Calculated disparity map with window size of 15.



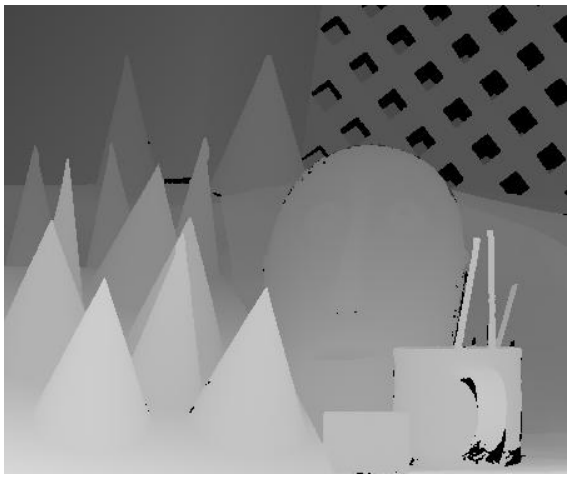
(e) Calculated disparity map with window size of 25.



(f) Calculated disparity map with window size of 35.

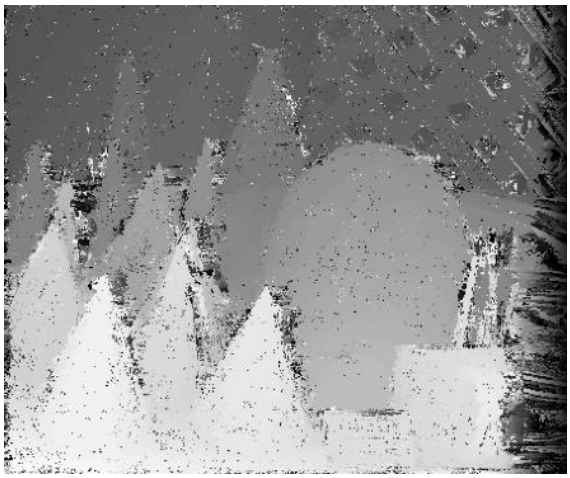
Figure 11: Left image with the ground truth disparity map, and the calculated disparity maps with different window sizes.

(a) Original right image.

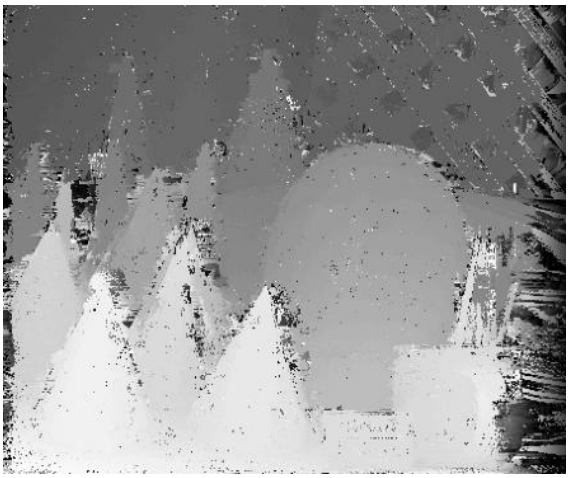


(b) Ground truth disparity map

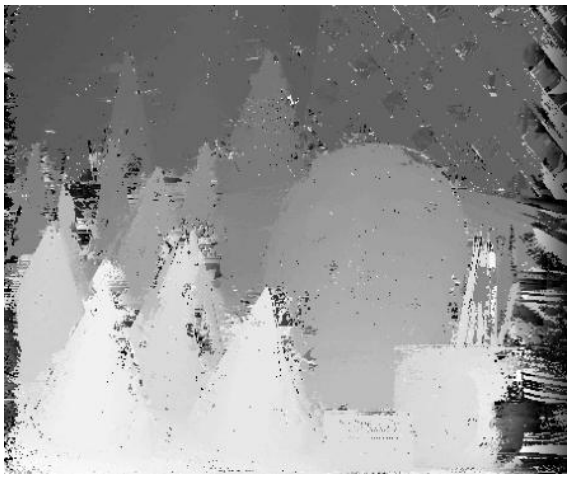
(c) Calculated disparity map with window size of 5.



(d) Calculated disparity map with window size of 15.

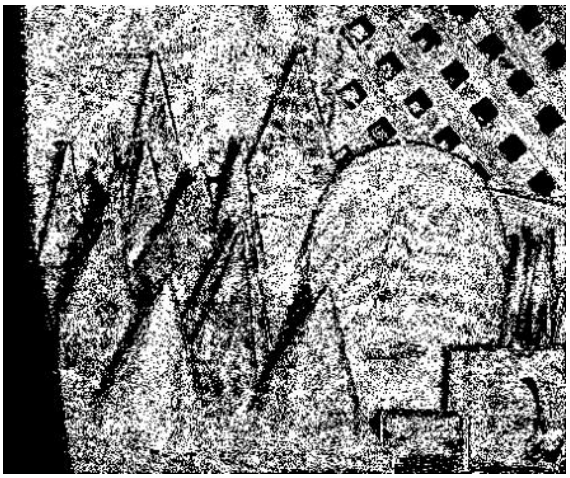


(e) Calculated disparity map with window size of 25.

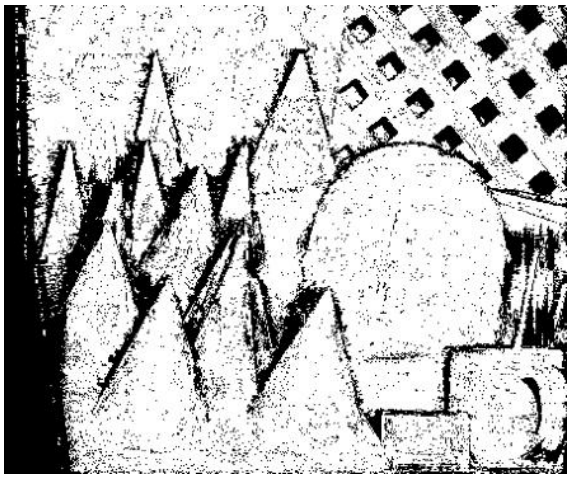


(f) Calculated disparity map with window size of 35.

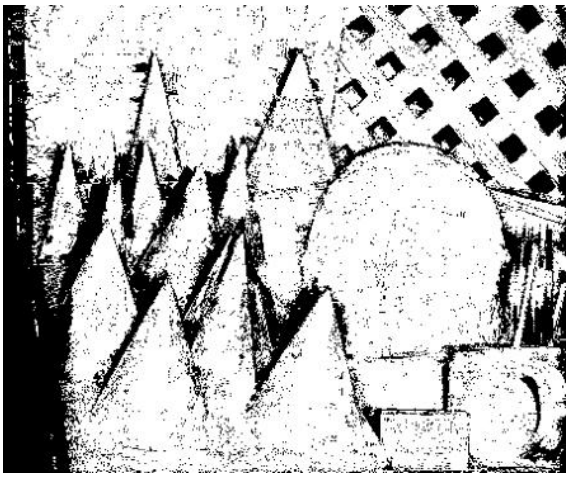
Figure 12: Right original image with the ground truth disparity map, and the calculated disparity maps with different window sizes.



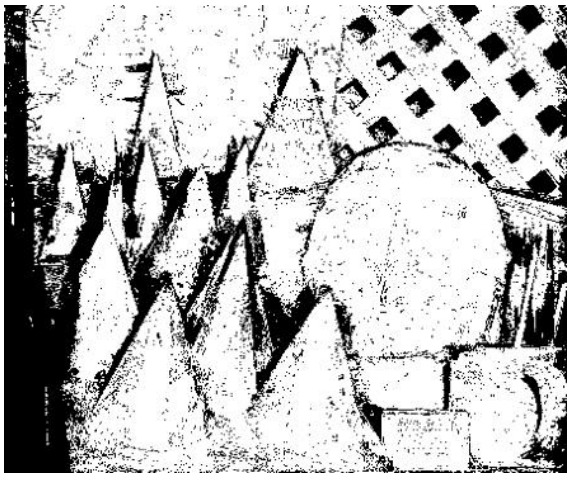
(a) Error mask for window size 5.



(b) Error mask for window size 15.

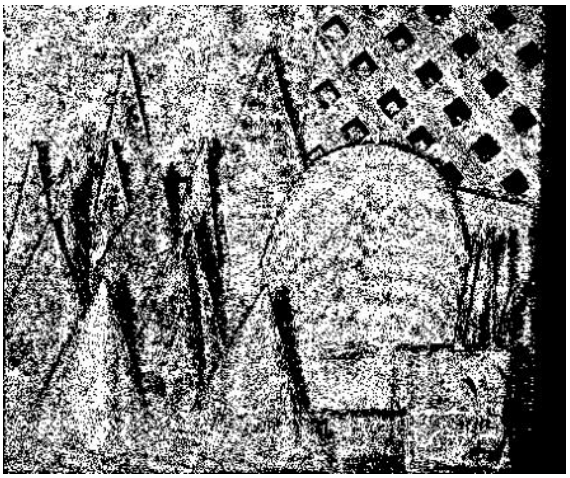


(c) Error mask for window size 25.

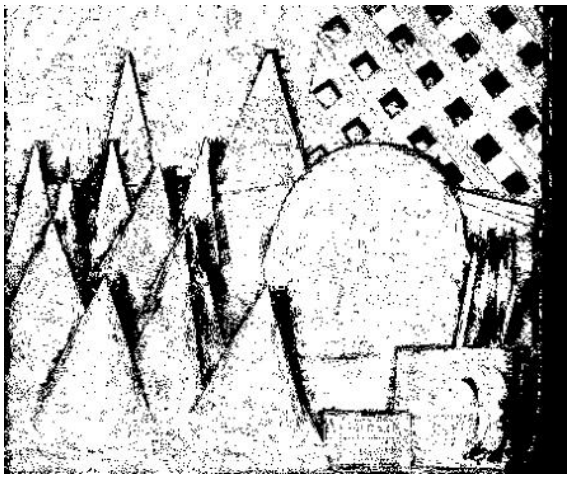


(d) Error mask for window size 35.

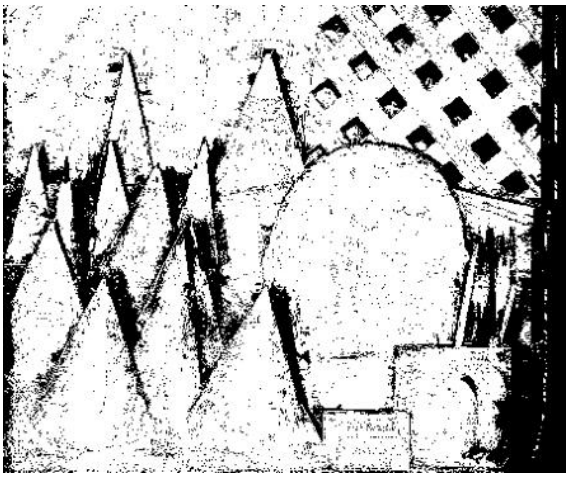
Figure 13: Left error masks for different window sizes.



(a) Error mask for window size 5.



(b) Error mask for window size 15.



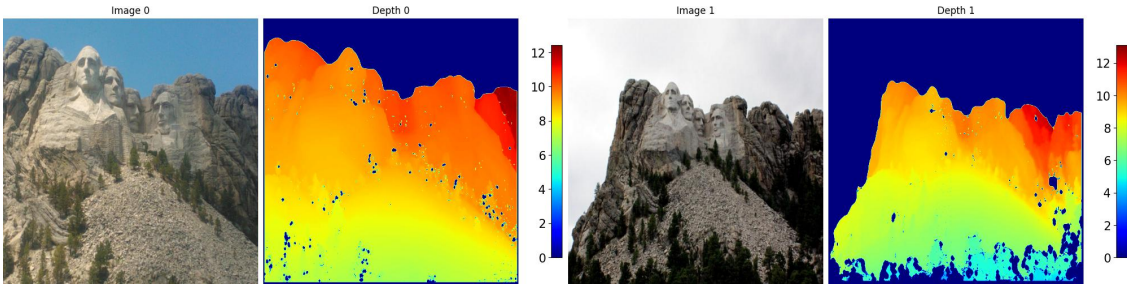
(c) Error mask for window size 25.

(d) Error mask for window size 35.

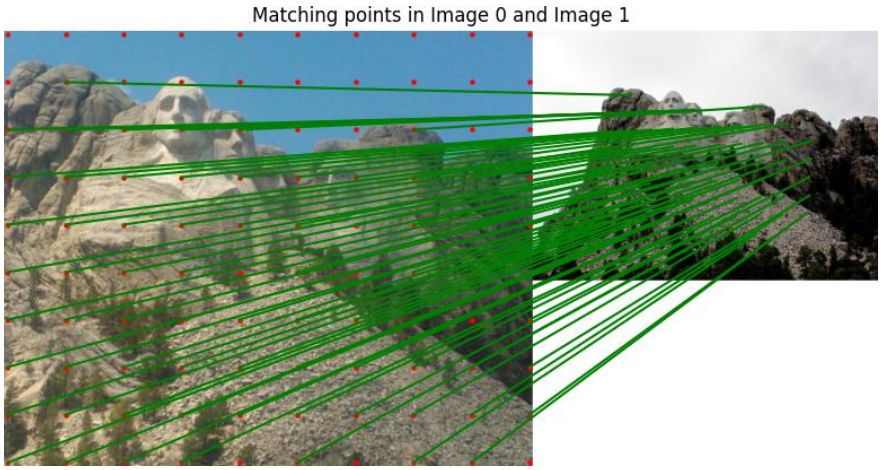
Figure 14: right error masks for different window sizes.

Results: Task 4

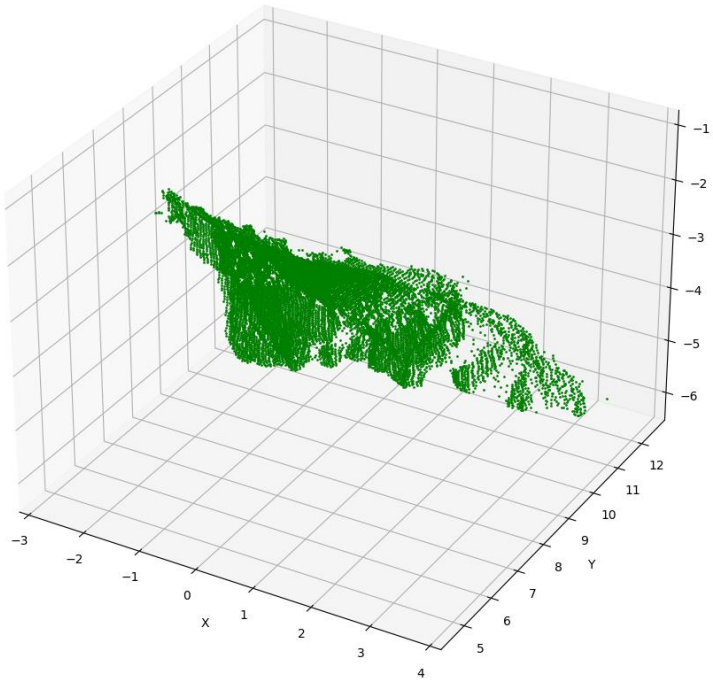
We follow the equations shown in the HW9.pdf exactly. The only thing that was changed from the comments of the code is that for us, understanding the positions x_i is easier as a $(N, 3)$ shape array rather than a $(3, N)$ shape array, thus in the code we do not follow the convention of our world/camera coordinates as $(3, N)$ shape. And the way we get the 3D depth map is by following a similar process to depth matching, we make a meshgrid, however this time we make it far more denser (100 by 100 points). We make use of this meshgrid to get the world coordinates like for depth matching, and we repeat the entire process for the 2nd image in the pair. We then make a scatterplot of the combination of the points obtained from image 1 and the points from image 2 for each pair. Notice how for the 3D scatterplot, we can actually see the shape of the faces of Mt. Rushmore, near the bottom of the structure, it is the spherical shapes.



(a) Image pair and their corresponding depth map.

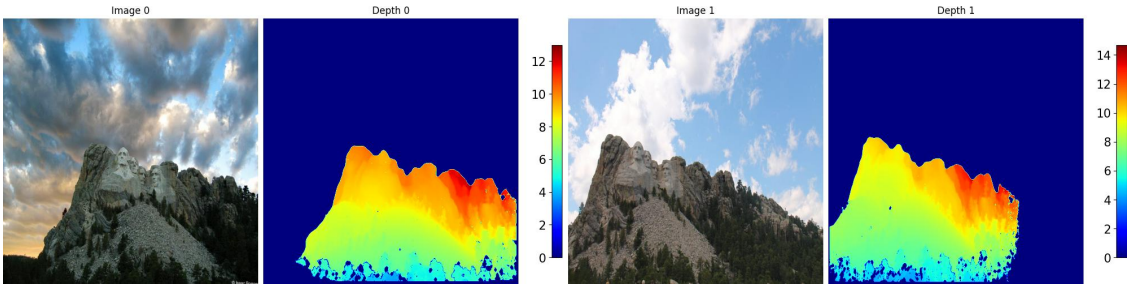


(b) Depth correspondence for this pair.

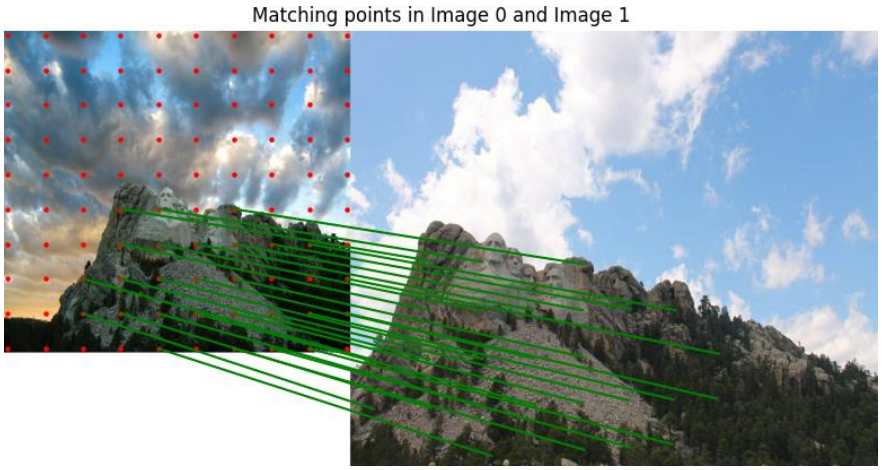


(c) 3D scatter plot built from both images.

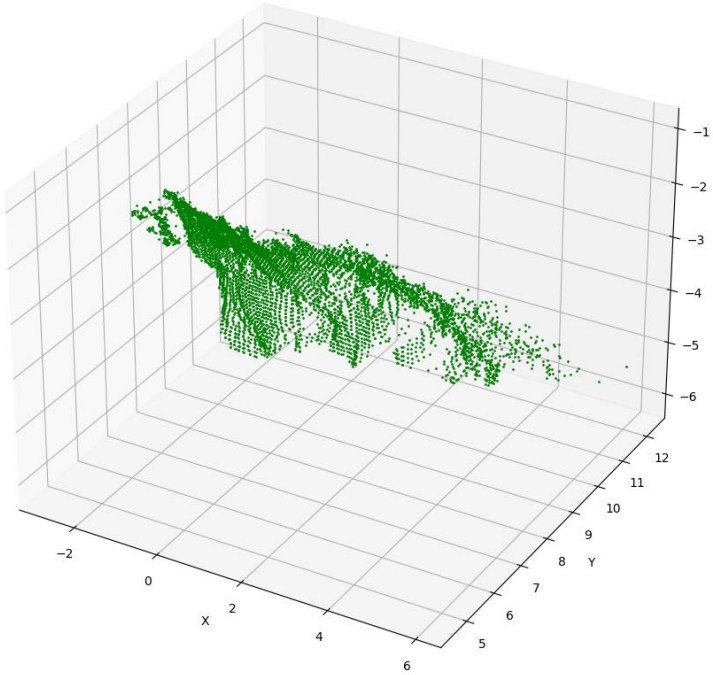
Figure 15: Results for image pair 0.



(a) Image pair and their corresponding depth map.

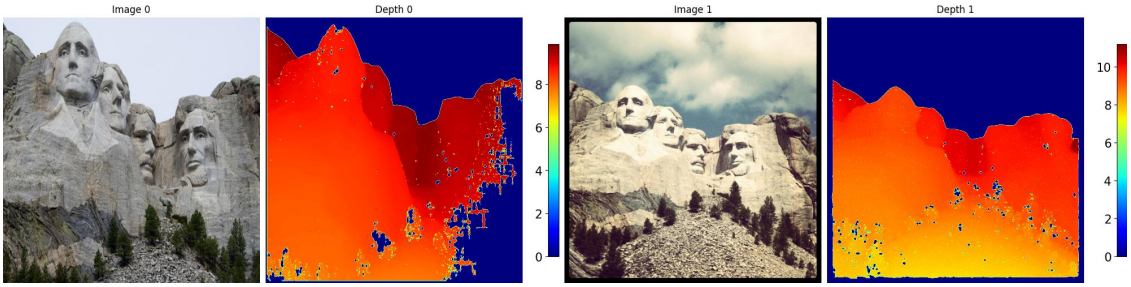


(b) Depth correspondence for this pair.

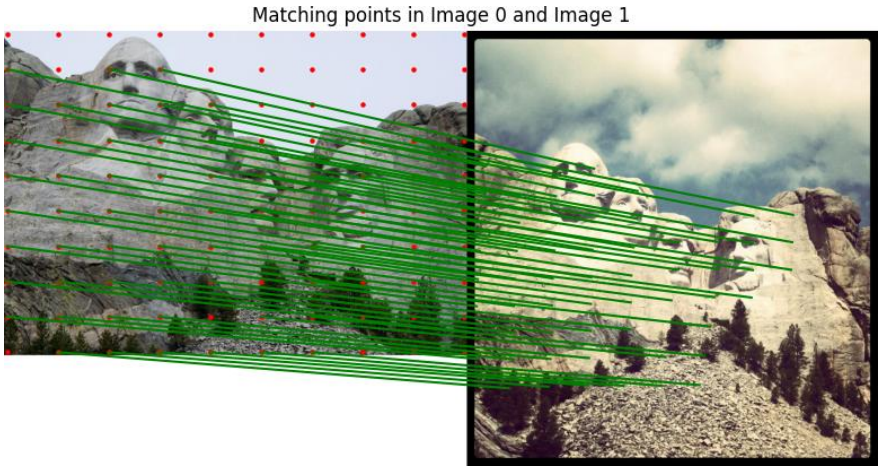


(c) 3D scatter plot built from both images.

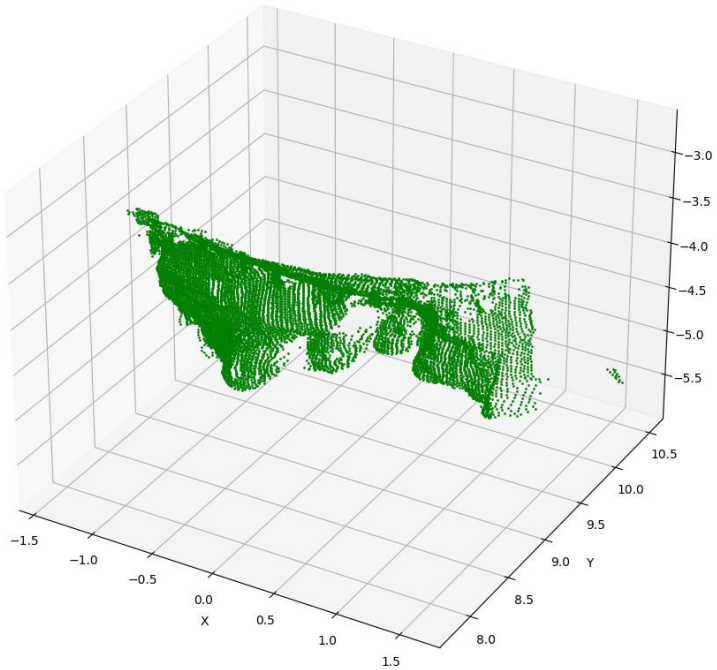
Figure 16: Results for image pair 1.



(a) Image pair and their corresponding depth map.

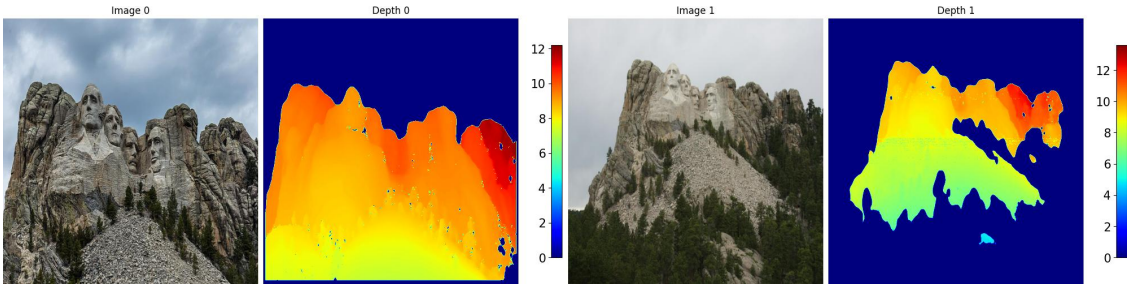


(b) Depth correspondence for this pair.

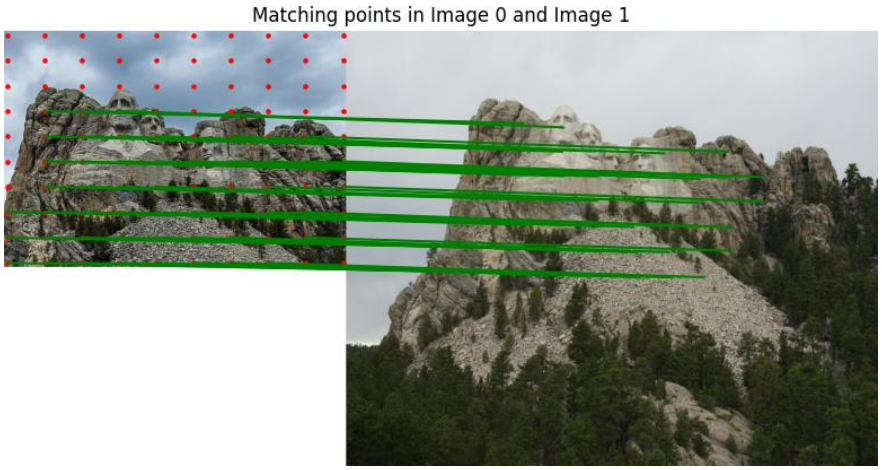


(c) 3D scatter plot built from both images.

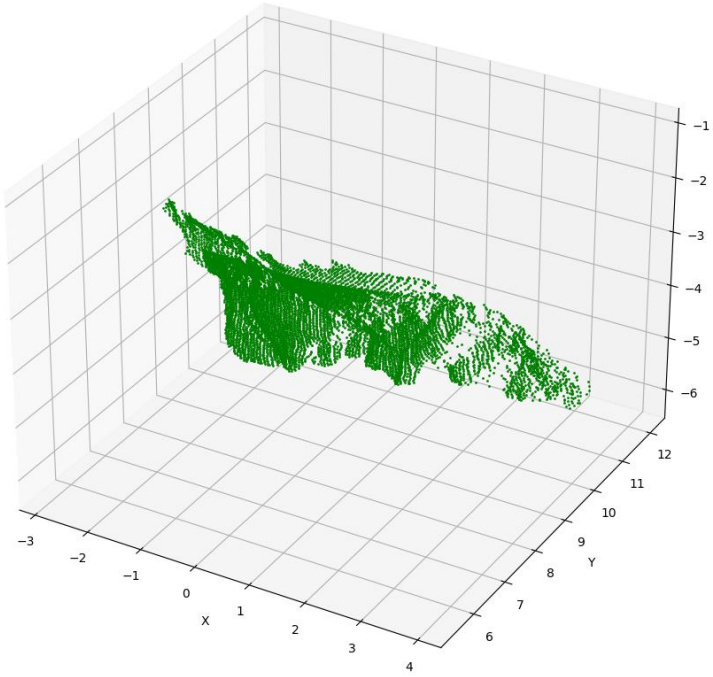
Figure 17: Results for image pair 2.



(a) Image pair and their corresponding depth map.

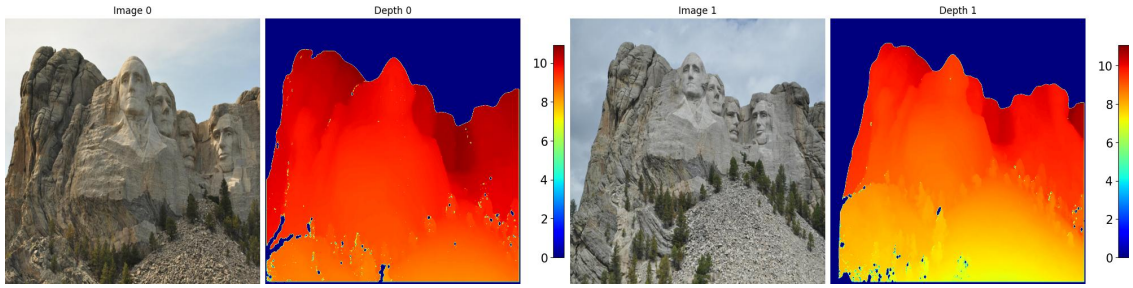


(b) Depth correspondence for this pair.

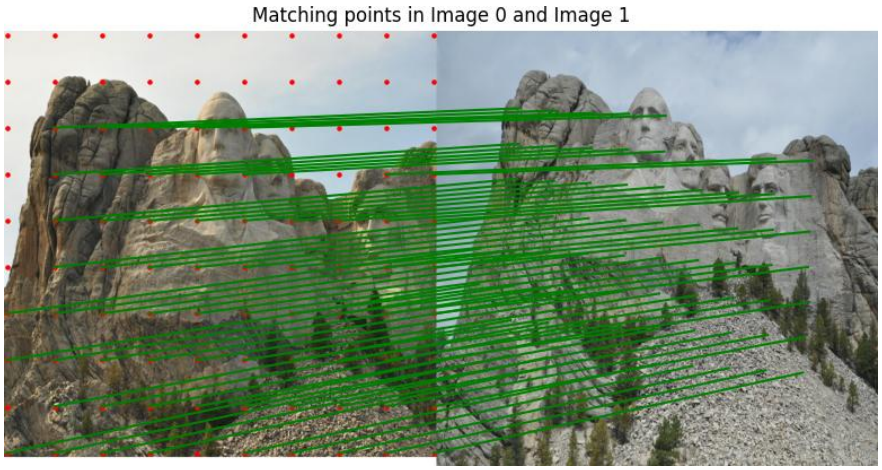


(c) 3D scatter plot built from both images.

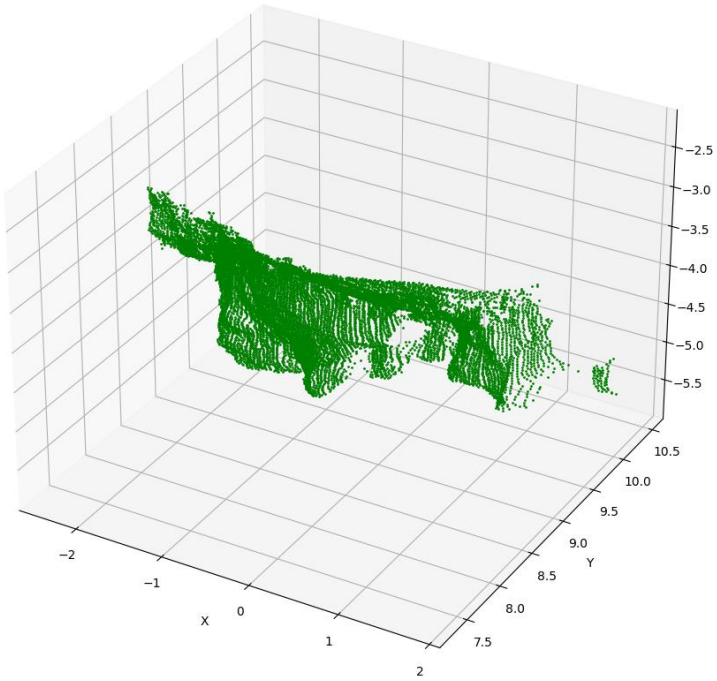
Figure 18: Results for image pair 3.



(a) Image pair and their corresponding depth map.

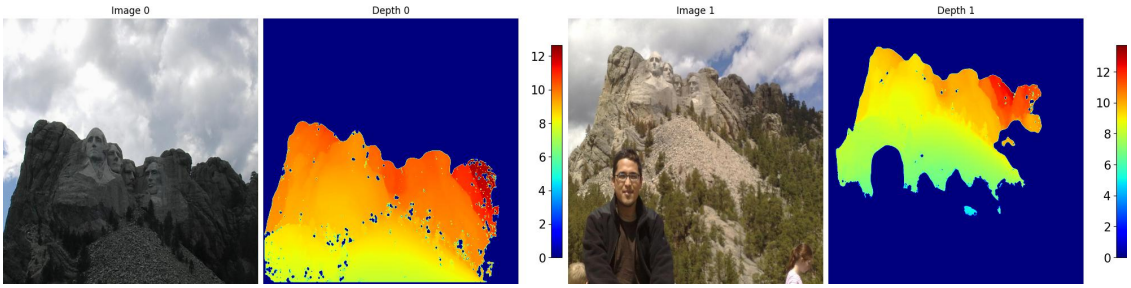


(b) Depth correspondence for this pair.

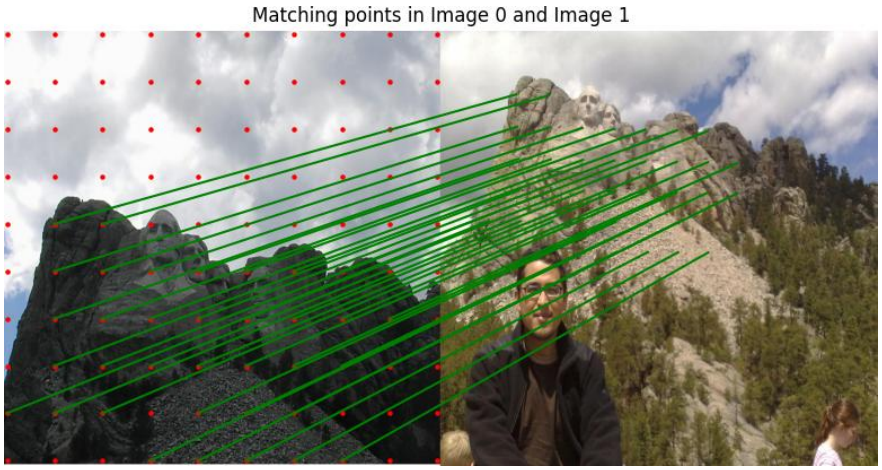


(c) 3D scatter plot built from both images.

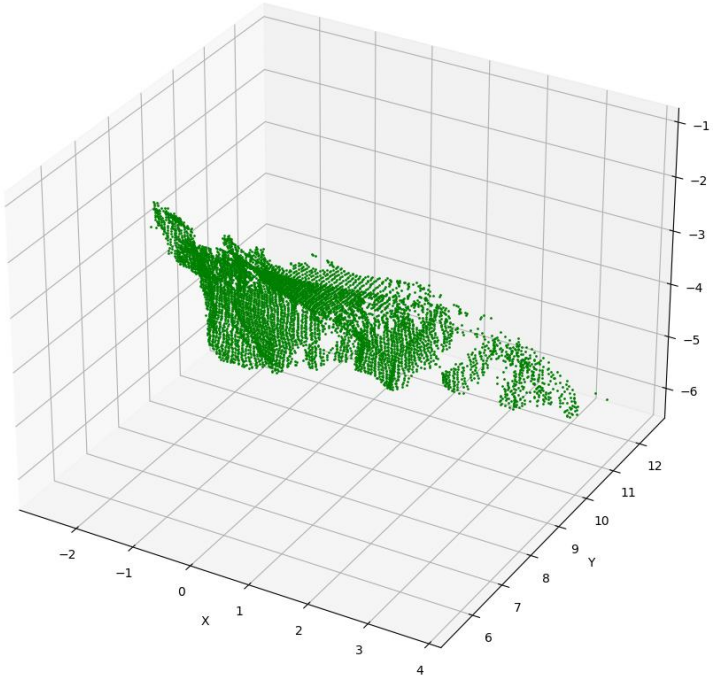
Figure 19: Results for image pair 4.



(a) Image pair and their corresponding depth map.

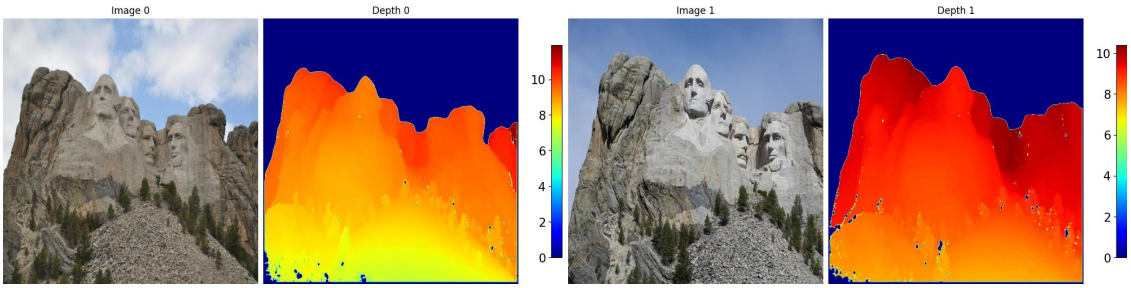


(b) Depth correspondence for this pair.

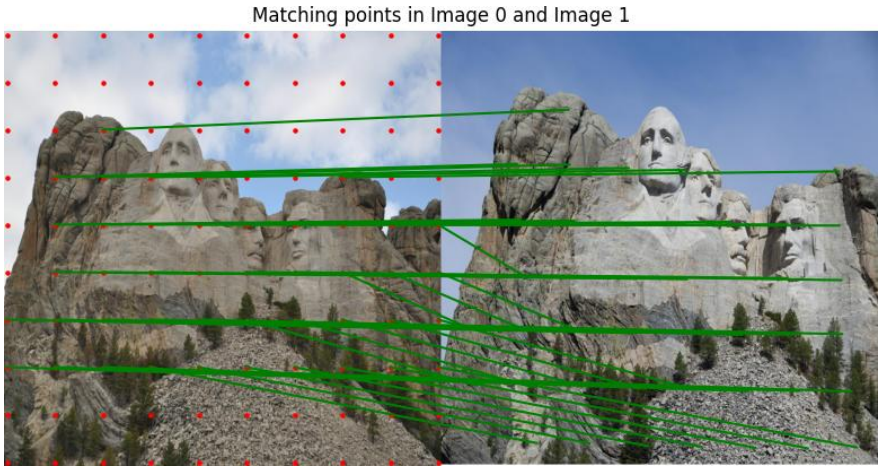


(c) 3D scatter plot built from both images.

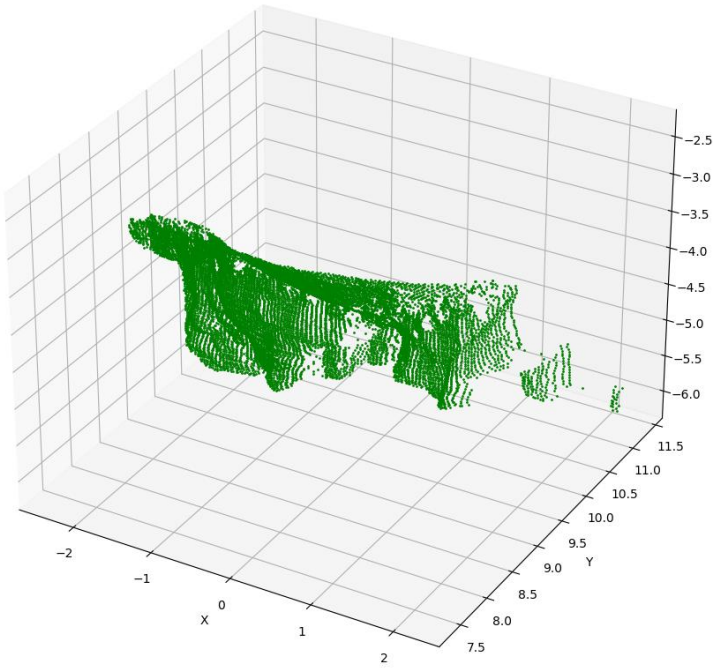
Figure 20: Results for image pair 5.



(a) Image pair and their corresponding depth map.

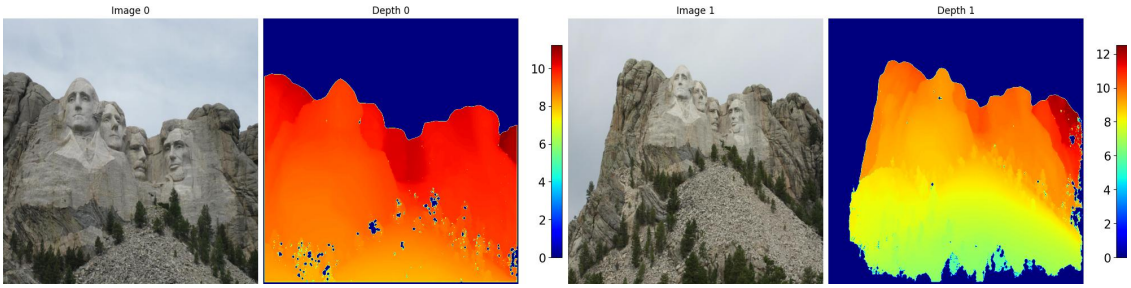


(b) Depth correspondence for this pair.

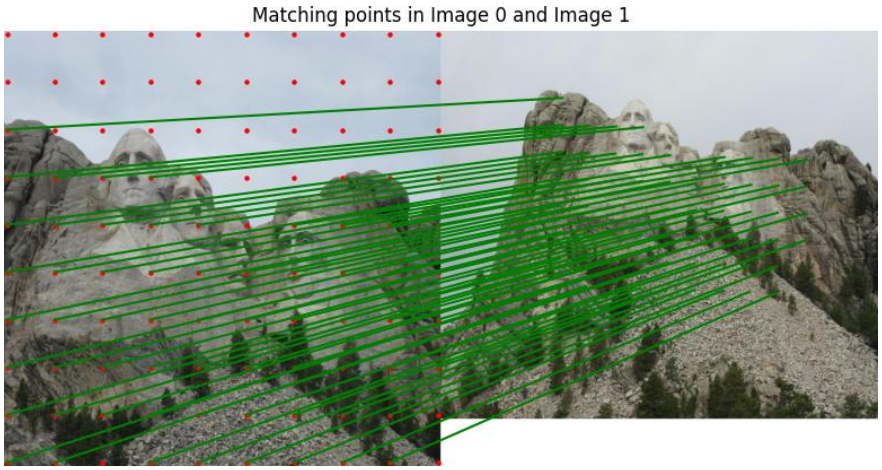


(c) 3D scatter plot built from both images.

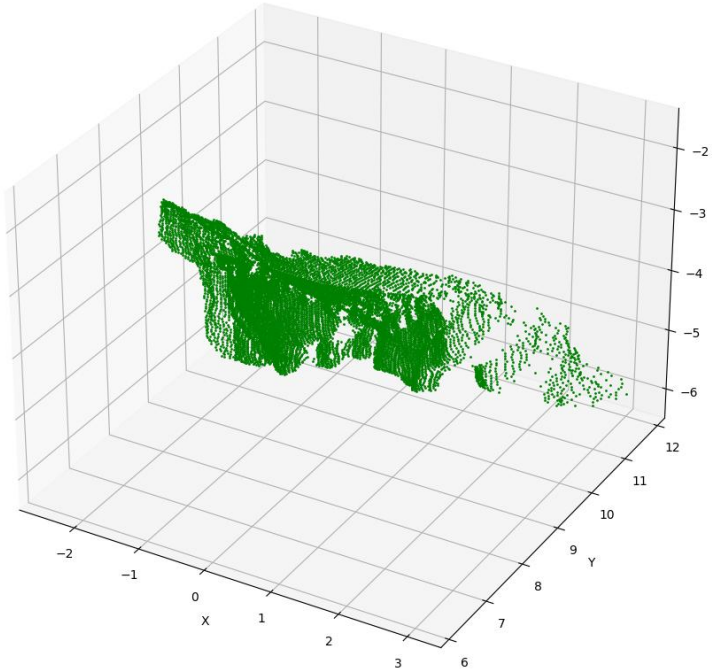
Figure 21: Results for image pair 6.



(a) Image pair and their corresponding depth map.

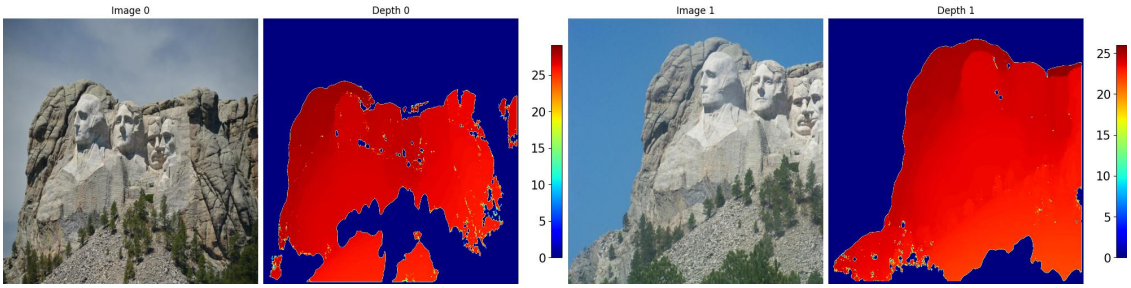


(b) Depth correspondence for this pair.

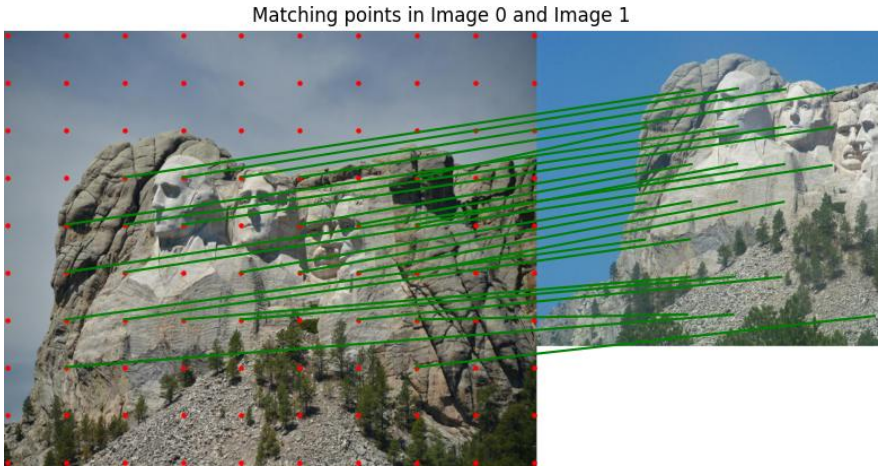


(c) 3D scatter plot built from both images.

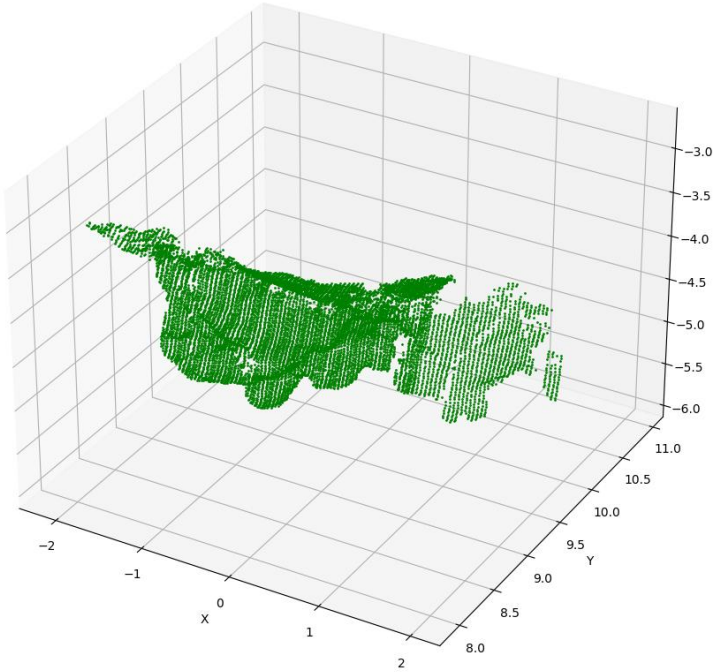
Figure 22: Results for image pair 7.



(a) Image pair and their corresponding depth map.

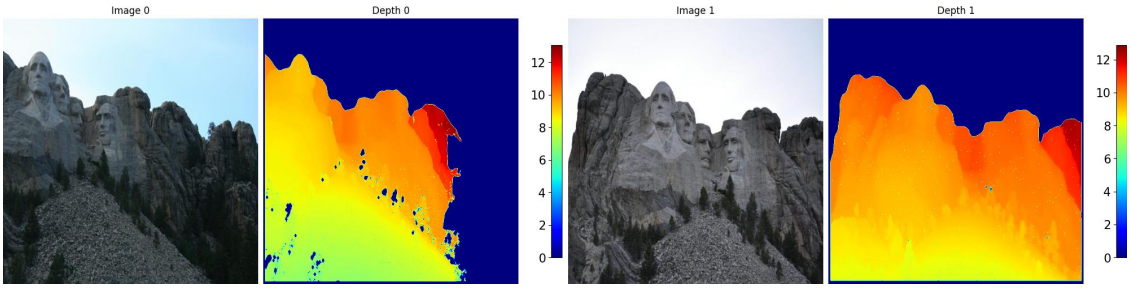


(b) Depth correspondence for this pair.

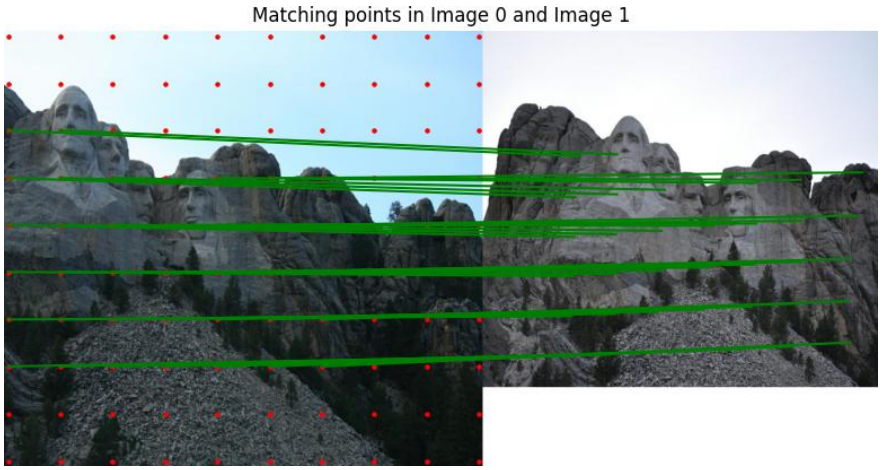


(c) 3D scatter plot built from both images.

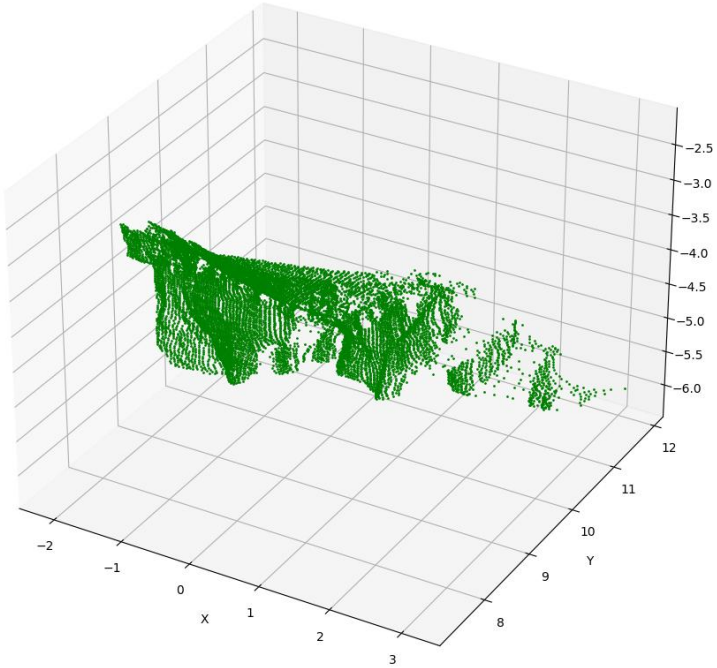
Figure 23: Results for image pair 8.



(a) Image pair and their corresponding depth map.



(b) Depth correspondence for this pair.



(c) 3D scatter plot built from both images.

Figure 24: Results for image pair 9.

Source code

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares
import json
from numpy.random import default_rng
# we use these to plot the camera poses
from mpl_toolkits.mplot3d.art3d import *
from matplotlib.patches import Rectangle
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
import matplotlib
from matplotlib.patches import ConnectionPatch

def add_number(array,n):
    # this function takes in an array and adds any number
    # this is mainly for adding a 0, for z=0 in the real life plane coordinates
    ones_np = np.ones(array.shape[:-1])[...,None]
    ones_np *= n
    # need to expand the dimensions by 1 to be able to concatenate it
    return np.append(array,ones_np,axis=-1)

def get_normalization_matrix(points):
    # follows from the logic
    centroid = points[:,None,:].mean(axis=0)[0]
    shifted_points = points - centroid
    dist = np.linalg.norm(shifted_points,axis=1).mean()
    scale = np.sqrt(2)/dist
    T = np.array([[scale, 0, -scale*centroid[0]],
                  [0, scale, -scale*centroid[1]],
                  [0, 0, 1]])
    return T

def get_normalized_points(points):
    # this is just to get the normalization matrix and get the normalized points
    T = get_normalization_matrix(points)
    normed_points = apply_homography(points, T)
    return normed_points

def point2point_fmatrix(domain, range_):
    # follows from the logic
    # we changed this function to accept an arbitrary number of points,
    # this is because most of the systems we will solve have more than 4 data points
    num_points = len(domain)
    mat1 = np.empty((0,9),dtype=float)

    for i in range(num_points):
        x = domain[i,0]
```

```
        x_prime = range_[i,0]

        y = domain[i,1]
        y_prime = range_[i,1]

        mat1 = np.append(mat1,np.array([[x_prime*x, x_prime*y, x_prime,
    y_prime*x, y_prime*y, y_prime, x,y, 1]]),axis=0)
    return mat1
def linear_least_squares_homogeneous(mat1):
    # and this is for homogeneous least squares
    _, _, v_t = np.linalg.svd(mat1)
    return v_t[-1]

def get_F_matrix(domain,range_):
    # this is to obtain our F matrix
    T_left = get_normalization_matrix(domain)
    T_right = get_normalization_matrix(range_)
    mat1 = point2point_fmatrix(domain,range_)
    F = linear_least_squares_homogeneous(mat1)
    F = F.reshape((3,3))
    F = F/F[2,2]

    # condition the matrix
    u, d, v_t = np.linalg.svd(F)
    # zero the minimum eigenvalue, take advantage of the fact that svd
    gives them in decreasing order
    d[-1] = 0
    F_conditioned = u @ np.diag(d) @ v_t
    F_conditioned = T_right.T @ F_conditioned @ T_left
    F_conditioned /= F_conditioned[2,2]
    return F_conditioned

def get_epipoles(F):
    # this is to get the epipoles
    u, d, v_t = np.linalg.svd(F)
    e = v_t[-1]
    e_prime = u[:,-1]
    return e/e[2], e_prime/e_prime[2]

def skew_symmetric_matrix(array):
    # creates skew symmetric matrix, input has to be a (3,) shape vector
    return np.array([[0, -array[2],array[1]],
                     [array[2],0,-array[0]],
                     [-array[1],array[0],0]])

def get_canonical_P(F, e, e_prime):
    # follows from the logic
    P = np.eye(3)
    P = add_number(P,0)
    sk = skew_symmetric_matrix(e_prime)
    P_p = sk @ F
    #print(P_p, e_prime)
    # we only care about the first 3 columns of P
    P_prime = np.column_stack((P_p[:,0],P_p[:,1],P_p[:,2],e_prime))
```

```
    #print(P_prime)
    #print(np.linalg.matrix_rank(P),np.linalg.matrix_rank(P_prime))
    return P, P_prime

def triangulate(P, P_prime, domain, range_):
    # triangulation, follows from the logic, we also compute all the points using this
    num_points = len(domain)
    rl_coords = []
    P_1 = P[0,:]
    P_2 = P[1,:]
    P_3 = P[2,:]
    Pp_1 = P_prime[0,:]
    Pp_2 = P_prime[1,:]
    Pp_3 = P_prime[2,:]

    for i in range(num_points):
        x = domain[i,0]
        y = domain[i,1]
        x_prime = range_[i,0]
        y_prime = range_[i,1]
        A = np.array([(x*P_3) - P_1,
                      (y*P_3 )- P_2,
                      (x_prime*Pp_3) - Pp_1,
                      (y_prime*Pp_3) - Pp_2])
        X_rl = linear_least_squares_homogeneous(A)
        X_rl = X_rl/X_rl[3]
        rl_coords.append(X_rl)
    return np.array(rl_coords)[:,:3]

def reconstruct_matrix_worldpoints(params):
    # function used to reconstruct P and the triangulated world coordinates for our LM
    P_prime = params[:12].reshape((3,4))
    worldpoint_list = []
    for idx in range(12, len(params),3):
        X_x = params[idx]
        X_y = params[idx+1]
        X_z = params[idx+2]
        worldpoint = np.array([X_x,X_y,X_z])
        worldpoint_list.append(worldpoint)
    worldpoint_list = np.array(worldpoint_list)
    return P_prime, worldpoint_list

def get_params(P_prime, world_points):
    # get the parameters to be optimized using LM
    params = []
    P_p_flat = P_prime.flatten()
    for p in P_p_flat:
        params.append(p)
    for point in world_points:
        params.append(point[0])
        params.append(point[1])
        params.append(point[2])
```

```
    return np.array(params)

def apply_homography(positions, H):
    # reused from hw5
    # this gets the homography transformation for all the coordinates in the image that we get from the get_positions function
    # we do it in a way that exploits broadcasting, so we don't need to use for loops
    temp_pos = add_number(positions,1)
    new_pos = (H @ temp_pos.T).astype(float)
    new_pos /= new_pos[-1,:]
    new_pos = new_pos.T
    return new_pos[:,:-1]

def error_f(params, P, domain, range_):
    # function to be optimized with LM
    P_prime, world_points = reconstruct_matrix_worldpoints(params)
    num_points = len(domain)
    error_total = []
    for i in range(num_points):
        rl_img1 = apply_homography(np.array([world_points[i]]), P)[0]
        rl_img2 = apply_homography(np.array([world_points[i]]), P_prime)[0]
        err_1 = np.abs(rl_img1 - domain[i])
        error_total.append(err_1[0])
        error_total.append(err_1[1])
        error_total.append(0)
        err_2 = np.abs(rl_img2 - range_[i])
        error_total.append(err_2[0])
        error_total.append(err_2[1])
        error_total.append(0)

    err_total = np.array(error_total)
    return err_total.ravel()

def lm_optim_Pprime(P, P_prime, domain, range_):
    # our LM function
    world_points = triangulate(P, P_prime, domain, range_)
    params = get_params(P_prime, world_points)
    params_op = least_squares(error_f, params, method='lm', args=(P, domain, range_),verbose=False).x
    P_prime_op, worldPts = reconstruct_matrix_worldpoints(params_op)
    P_prime_op = P_prime_op/np.linalg.norm(P_prime_op[:,-1])
    return P_prime_op

def get_F_refined(P, P_prime_op):
    # this is to obtain our refined F using the P_prime we got from LM
    e_op = P_prime_op[:,-1]
    sk = skew_symmetric_matrix(e_op)
    F_op = sk @ P_prime_op @ np.linalg.pinv(P)
    return F_op[:,:3]/F_op[2,2]

def get_translation_matrix(h, w):
    # translation matrix mentioned in the logic
```

```
    T = np.array([[1, 0, -w/2],
                  [0, 1, -h/2],
                  [0, 0, 1]])
    return T
def get_rotation_matrix(e, h, w):
    # rotation matrix for our rectifying homography, follows from the logic
    theta = np.arctan(-(e[1] - h/2)/(e[0] - w/2))
    R = np.array([[np.cos(theta), -np.sin(theta), 0],
                  [np.sin(theta), np.cos(theta), 0],
                  [0, 0, 1]])
    return R, theta
def get_transformation_epipole2infinity(e, theta, h, w):
    # G matrix, from logic
    f = np.abs((e[0] - w/2) * np.cos(theta) - (e[1] - h/2) * np.sin(theta))

    G = np.array([[1, 0, 0],
                  [0, 1, 0],
                  [-1/f, 0, 1]])
    return G

def get_rectifying_homography(image, e):
    # this is to get the right homography, follows from the logic
    h, w, _ = image.shape
    T = get_translation_matrix(h, w)
    R, theta = get_rotation_matrix(e, h, w)
    G = get_transformation_epipole2infinity(e, theta, h, w)

    H_c = G @ R @ T
    center = np.array([[w/2,h/2]])
    rectified_center = apply_homography(center,H_c)[0]
    T_rc = np.array([[1, 0, w/2],
                     [0, 1, h/2],
                     [0,0,1]])
    H = T_rc @ H_c
    H /= H[2,2]

    return H

def point2point_Hleft(domain, range_):
    # modified form the other function to solve our left homography system
    # we changed this function to accept an arbitrary number of points,
    # this is because most of the systems we will solve have more than 4 data points
    num_points = len(domain)
    mat1 = np.empty((0,3),dtype=float)
    mat2 = np.empty((0,1),dtype=float)

    for i in range(num_points):
        x = domain[i,0]
        x_prime = range_[i,0]
        y = domain[i,1]
        y_prime = range_[i,1]
```

```python

        mat1 = np.append(mat1,np.array([[x, y, 1.]]),axis=0)
        mat2 = np.append(mat2, x_prime)

    return mat1, mat2[:,None]

def linear_least_squares(mat1, mat2):
    res = np.linalg.inv(mat1.T @ mat1) @ mat1.T @ mat2
    return res



def get_left_homography(P, P_prime, image_left, img_left_pts, img_right_pts, H_right):
    # this is to get our left homography
    h, w, _ = image_left.shape
    M = P_prime @ np.linalg.pinv(P)
    H_ini = H_right @ M
    H_ini /= H_ini[2,2]
    x = apply_homography(img_left_pts, H_ini)
    x_prime = apply_homography(img_right_pts, H_right)

    m1, m2 = point2point_Hleft(x, x_prime)
    H_abc_ = linear_least_squares(m1, m2)
    H_abc = np.eye(3)
    H_abc[0,:] = H_abc_[:,0]
    H_final = H_abc @ H_ini
    H_final /= H_final[2,2]

    return H_final

# the functions here are all for rectifying our images
def get_new_corners(img,H):
    # get the new corners after transformation
    if len(img.shape) == 2:
        h, w = img.shape
    else:
        h, w, _ = img.shape
    corners = np.array([[0,0],[w,0],[w,h],[0,h]])
    new_corners = apply_homography(corners,H)
    return new_corners
def paint_image(panorama,image, old_positions, new_positions,x_min,y_min):
    # reused from hw5, it is to paint the blank canvas
    #print(new_positions)
    for idx, n_pos in enumerate(new_positions):
        if 0 <= old_positions[idx,0] < image.shape[1] and 0 <= old_positions[idx,1] < image.shape[0]: #and 0 < new_positions[idx,1] - x_min < panorama.shape[1] and 0 <:
            panorama[new_positions[idx,1] - y_min,new_positions[idx,0] - x_min] = image[int(old_positions[idx,1]),int(old_positions[idx,0])]
    return panorama
def get_binary_mask(image, H):
    # reused from hw5
```

```
    # we want these masks so we only iterate through the parts with the mask,
    # otherwise we need to iterate through the whole final panorama, which we don't want
    new_corners = get_new_corners(image,H)
    new_corners = np.ceil(new_corners).astype(int)
    x_min = new_corners[:,0].min()
    y_min = new_corners[:,1].min()
    new_corners[:,0] -= x_min
    new_corners[:,1] -= y_min

    x_max = new_corners[:,0].max()
    y_max = new_corners[:,1].max()


    canvas = np.zeros((y_max, x_max))
    empty = np.zeros((y_max, x_max,3)).astype(np.uint8)

    canvas = cv2.fillPoly(canvas,pts=[new_corners],color=255)
    return canvas, empty, x_min, y_min

def rectify_images(img_left, img_right, H_left, H_right):
    # recycled from hw5, somewhat
    # first get the blank canvas and also the roi map to project the image onto it
    # first we get all the corners, and get the new image shapes from these
    if len(img_left.shape) == 2:
        c = None
    else:
        c = 3
    corners_left = np.ceil(get_new_corners(img_left,H_left)).astype(int)
    x_min_left = corners_left[:,0].min()
    x_max_left = corners_left[:,0].max()
    w_left = x_max_left - x_min_left

    y_min_left = corners_left[:,1].min()
    y_max_left = corners_left[:,1].max()
    h_left = y_max_left - y_min_left

    corners_right = np.ceil(get_new_corners(img_right,H_right)).astype(int)

    x_min_right = corners_right[:,0].min()
    x_max_right = corners_right[:,0].max()
    w_right = x_max_right - x_min_right

    y_min_right = corners_right[:,1].min()
    y_max_right = corners_right[:,1].max()
    h_right = y_max_right - y_min_right

    x_min = np.array([x_min_left, x_min_right]).min()
    y_min = np.array([y_min_left, y_min_right]).min()

```

```
    x_max = np.array([x_max_left, x_max_right]).max()
    y_max = np.array([y_max_left, y_max_right]).max()

    w_final = x_max - x_min
    h_final = y_max - y_min
    if c == None:
        canvas_left = np.zeros((h_final, w_final)).astype(np.uint8)
    else:
        canvas_left = np.zeros((h_final, w_final, 3)).astype(np.uint8)
    roi_left = np.zeros((h_final, w_final))
    corners_left[:,0] -= x_min
    corners_left[:,1] -= y_min
    roi_left = cv2.fillPoly(roi_left,pts=[corners_left],color=255)

    if c == None:
        canvas_right = np.zeros((h_final, w_final)).astype(np.uint8)
    else:
        canvas_right = np.zeros((h_final, w_final, 3)).astype(np.uint8)
    roi_right = np.zeros((h_final, w_final))
    corners_right[:,0] -= x_min
    corners_right[:,1] -= y_min
    roi_right = cv2.fillPoly(roi_right,pts=[corners_right],color=255)

    new_positions_left = (np.argwhere(roi_left == 255) + np.array([[y_min, x_min]]))[:,[1,0]]
    old_positions_left = apply_homography(new_positions_left, np.linalg.inv(H_left))

    new_positions_right = (np.argwhere(roi_right == 255) + np.array([[y_min, x_min]]))[:,[1,0]]
    old_positions_right = apply_homography(new_positions_right, np.linalg.inv(H_right))


    rect_left = paint_image(canvas_left, img_left, old_positions_left, new_positions_left, x_min, y_min)
    rect_right = paint_image(canvas_right, img_right, old_positions_right, new_positions_right, x_min, y_min)
    # we also return the offset as we will make use of it later
    return rect_left, rect_right, x_min, y_min

def plot_correspondences(image_l, image_r, l_pts, r_pts, figname):
    # this is a modified function from superglue_ece661.py, plot_keypoints in particular
    # reused from hw5
    tot_img = np.hstack((image_l, image_r))
    tot_img = cv2.cvtColor(tot_img, cv2.COLOR_BGR2RGB)
    _, w0, _ = image_l.shape
    ms = 0.5
    lw = 0.5

    plt.figure()
    #plt.clf()
    plt.imshow(tot_img);
```

```
    # plot inliers in green (match and keypoint)
    for ikp0 in range(len(l_pts)):
        color_random = np.random.random(3)
        plt.plot(l_pts[ikp0,0], l_pts[ikp0,1], marker="o",color=color_random, markersize=ms)
        plt.plot(r_pts[ikp0,0]+w0, r_pts[ikp0,1], marker="o", color=color_random, markersize=ms)
        plt.plot((l_pts[ikp0,0], r_pts[ikp0,0]+w0), (l_pts[ikp0,1], r_pts[ikp0,1]), color=color_random,linestyle='--', linewidth=lw, markersize=ms)
    plt.axis('off');
    plt.savefig(figname, bbox_inches='tight', pad_inches=0, dpi=300)
    plt.close()
    return

# since we used labelme, the points are saved as json dict, so we use this to load them
with open('image_left.json', "r") as f:
    data = json.load(f)

img_left_points = np.array(data["shapes"][0]["points"])

with open('image_right.json', "r") as f:
    data = json.load(f)

img_right_points = np.array(data["shapes"][0]["points"])

img_left = cv2.imread("image_left.jpeg")
img_right = cv2.imread("image_right.jpeg")

# get correspondences
plot_correspondences(img_left, img_right, img_left_points, img_right_points, '8point_correspondence.png')
# get F and normalize it
F_m = get_F_matrix(img_left_points, img_right_points)
e, e_prime = get_epipoles(F_m)
P, P_prime = get_canonical_P(F_m, e, e_prime)
P_prime_op = lm_optim_Pprime(P, P_prime, img_left_points, img_right_points)
F_op = get_F_refined(P, P_prime_op)

F_op /= F_op[2,2]
# get the epipoles
e_op, e_prime_op = get_epipoles(F_op)
# now the left and right homographies
H_right = get_rectifying_homography(img_right, e_prime_op)
H_right /= H_right[2,2]

H_left = get_left_homography(P, P_prime_op, img_left, img_left_points, img_right_points, H_right)
H_left /= H_left[2,2]

rect_left, rect_right, x_offset, y_offset = rectify_images(img_left, img_right, H_left, H_right)
```

```python

together = np.hstack((rect_left, rect_right))
cv2.imwrite("both_rectified.png",together)

rect_pts_left = apply_homography(img_left_points,H_left)

rect_pts_right = apply_homography(img_right_points,H_right)
# this is where we use the offsets
rect_pts_left -= np.array([[x_offset, y_offset]])
rect_pts_right -= np.array([[x_offset, y_offset]])

plot_correspondences(rect_left, rect_right, rect_pts_left, rect_pts_right, '8point_correspondence_rectified.png')

gray_left = cv2.imread("image_left.jpeg", cv2.IMREAD_GRAYSCALE)
gray_right = cv2.imread("image_right.jpeg", cv2.IMREAD_GRAYSCALE)
canny_p1=50
canny_p2=150
edges_l = cv2.Canny(gray_left,canny_p1,canny_p2)
edges_r = cv2.Canny(gray_right,canny_p1,canny_p2)
# rectify the canne output as explained in the report
rect_ed_l, rect_ed_r, x_offset, y_offset = rectify_images(edges_l, edges_r, H_left, H_right)
together_canny = np.hstack((rect_ed_l, rect_ed_r))
cv2.imwrite("rectified_cannys.png",together_canny)
# the next few functions are for the correspondences
def get_neighbours(image_r, image_gray_r, pos, window_size=9, delta=2):
    # reused from hw4
    # this might not be the best way to do it, but this is the fastest idea I could come up with
    # first we get the nonzero positions in image_r
    image_r_nonzero = np.argwhere(image_r != 0)
    # now we check which ones are within a similar row to the position we are looking for
    image_r_filtered = np.argwhere(np.abs(image_r_nonzero[:,0] - pos[0]) < delta)
    image_r_f = image_r_nonzero[image_r_filtered][:,0,:]
    nh = []
    pos_list = []
    for position in image_r_f:
        idx = position[1]
        jdx = position[0]
        window = image_gray_r[jdx-window_size//2:jdx+window_size//2+1,idx-window_size//2:idx+window_size//2+1]
        nh.append(window)
        pos_list.append(position)
    return np.array(nh), np.array(pos_list)
def compute_ssd(template,templates):
    # this is to get the SSD for all 3 channels, and then average them at the end
    #template = template[None,:]

    # Compute the squared differences between the single template and each template
```

```
    ssd = ((templates - template)**2 )
    ssd_ = ssd.sum(axis=(1,2,3))

    return ssd_

image_l_nonzero = np.argwhere(rect_ed_l != 0)

gray_rect_left = cv2.cvtColor(rect_left,cv2.COLOR_BGR2GRAY)
gray_rect_right = cv2.cvtColor(rect_right,cv2.COLOR_BGR2GRAY)

window_size = 25
x_tol = 30
delta=3
# perform ssd matching, delta is just the number of extra rows we scan, 2 up 2 down
matches = []
for l_pos in image_l_nonzero:
    neighs, neigh_pos_list = get_neighbours(rect_ed_r, rect_right,l_pos, window_size=window_size, delta=delta)
    #print(neighs.shape)
    window_l = rect_left[l_pos[0] - window_size//2:l_pos[0] + window_size//2 + 1, l_pos[1] - window_size//2:l_pos[1] - window_size//2 + 1][None,...]

    try:
        ssd_match = compute_ssd(window_l, neighs)
        mat = np.argmin(ssd_match)
        if np.abs(l_pos[1] - neigh_pos_list[mat,1]) < x_tol:

            matches.append([l_pos[0],l_pos[1], neigh_pos_list[mat,0], neigh_pos_list[mat,1], ssd_match[mat]])
        else:

            continue
    except:
        continue


matches = np.array(matches)
sorted_matches = matches[matches[:, 4].argsort()]
# change the y and x due to image coordinates
l_match = sorted_matches[:,[1,0]]
r_match = sorted_matches[:,[3,2]]



rng = default_rng()
numbers = rng.choice(3800, size=500, replace=False) # pick 500 at random

plot_correspondences(rect_left, rect_right, l_match[numbers], r_match[numbers], 'test_matches_rectified.png')

# get x and y from the original image
# first consider the offset
```

```python

l_matches = l_match[numbers]
l_matches[:,0] += x_offset
l_matches[:,1] += y_offset
r_matches = r_match[numbers]
r_matches[:,0] += x_offset
r_matches[:,1] += y_offset

# we can project these into our original image using the inverse homographies

l_matches_og = apply_homography(l_matches, np.linalg.inv(H_left))
r_matches_og = apply_homography(r_matches, np.linalg.inv(H_right))

plot_correspondences(img_left, img_right, l_matches_og, r_matches_og, 'test_matches_projectedog.png')
# the following functions are to optimize only the world coordinates
def triangulate(P, P_prime, domain, range_):

    num_points = len(domain)
    rl_coords = []
    P_1 = P[0,:]
    P_2 = P[1,:]
    P_3 = P[2,:]
    Pp_1 = P_prime[0,:]
    Pp_2 = P_prime[1,:]
    Pp_3 = P_prime[2,:]

    for i in range(num_points):
        x = domain[i,0]
        y = domain[i,1]
        x_prime = range_[i,0]
        y_prime = range_[i,1]
        A = np.array([(x*P_3) - P_1,
                      (y*P_3 )- P_2,
                      (x_prime*Pp_3) - Pp_1,
                      (y_prime*Pp_3) - Pp_2])
        X_rl = linear_least_squares_homogeneous(A)
        X_rl = X_rl/X_rl[-1]
        rl_coords.append(X_rl[:3])
    return np.array(rl_coords)[:,:3]

def reconstruct_worldpoints(params):
    worldpoint_list = []
    for idx in range(0,len(params),3):
        X_x = params[idx]
        X_y = params[idx+1]
        X_z = params[idx+2]
        worldpoint = np.array([X_x,X_y,X_z])
        worldpoint_list.append(worldpoint)
    worldpoint_list = np.array(worldpoint_list)
    return worldpoint_list

def get_params_wp(world_points):
```

```
    params = []
    for point in world_points:
        params.append(point[0])
        params.append(point[1])
        params.append(point[2])
    return np.array(params)

def error_f_wp(params, P, P_prime, domain, range_):
    world_points = reconstruct_worldpoints(params)
    num_points = len(domain)
    error_total = []
    for i in range(num_points):
        rl_img1 = apply_homography(np.array([world_points[i]]), P)[0]
        rl_img2 = apply_homography(np.array([world_points[i]]), P_prime)[0]
        err_1 = np.abs(rl_img1 - domain[i])
        error_total.append(err_1[0])
        error_total.append(err_1[1])
        err_2 = np.abs(rl_img2 - range_[i])
        error_total.append(err_2[0])
        error_total.append(err_2[1])

    err_total = np.array(error_total)
    return err_total.ravel()

def lm_optim_wp(P, P_prime, domain, range_):

    world_points = triangulate(P, P_prime, domain, range_)

    params = get_params_wp(world_points)
    params_op = least_squares(error_f_wp, params, method='lm', args=(P, P_prime, domain, range_),verbose=True).x
    worldPts = reconstruct_worldpoints(params_op)
    return worldPts
# we also place the original correspondences
total_left_pts = np.vstack((l_matches_og, img_left_points))

total_right_pts = np.vstack((r_matches_og, img_right_points))

world_pts = lm_optim_wp(P, P_prime_op, total_left_pts, total_right_pts)

norm_max = world_pts.max()

world_pts = world_pts/norm_max
# these are the original world points
world_pts_og = world_pts[-8:]

def get_camera_pose(P,norm_max):
    # follows from hw8
    K = np.eye(3)
    R_t = np.linalg.inv(K) @ P
    R = R_t[:,:3]
    # we noticed that R was not orthonormal, so we condition it to be orthonormal here
```

```
    u, _, v = np.linalg.svd(R)
    R_conditioned = u @ v
    t = R_t[:,3]
    #R = R/R[2,2]
    C = - R_conditioned.T @ t

    X_xcam = np.array([1,0,0])
    X_ycam = np.array([0,1,0])
    X_zcam = np.array([0,0,1])

    X_x = R_conditioned.T @ X_xcam
    X_y = R_conditioned.T @ X_ycam
    X_z = R_conditioned.T @ X_zcam


    return C/norm_max, X_x, X_y, X_z

c, dx_x, dx_y, dx_z = get_camera_pose(P,norm_max)
c_prime, dx_x_prime, dx_y_prime, dx_z_prime = get_camera_pose(P_prime_op,
    norm_max)



fig = plt.figure(figsize=(11,11))
ax1 = fig.add_subplot(111, projection='3d')
ms = 5
lw = 0.5
color_random = np.random.random((8,3))
for idx in range(len(world_pts)-8):
    ax1.scatter(world_pts[idx,0],world_pts[idx,1],world_pts[idx,2], marker
    ='o', color='g', s=2)
for idx in range(len(world_pts_og)):
    ax1.scatter(world_pts_og[idx,0],world_pts_og[idx,1],world_pts_og[idx
    ,2], marker='o', color=color_random[idx], s=7)
#camera 1
color_random_ = np.random.random(3)
center = c
c_x, c_y, c_z = center
x_dir = dx_x/10
x_x, x_y, x_z = x_dir
y_dir = dx_y/10
y_x, y_y, y_z = y_dir
z_dir = dx_z/10
z_x, z_y, z_z = z_dir

# this is for our direction vectors
ax1.quiver(c_x, c_y, c_z, x_x, x_y, x_z, color='r', linewidth=1)
ax1.quiver(c_x, c_y, c_z, y_x, y_y, y_z, color='g', linewidth=1)
ax1.quiver(c_x, c_y, c_z, z_x, z_y, z_z, color='b', linewidth=1)

# now the camera plane, in here we notice that the corners are made up
    from addition and substractions of the camera center and the directions
    ,
# so we make use to that
```

```
points = [center + x_dir + y_dir,
        center + x_dir - y_dir,
        center - x_dir - y_dir,
        center - x_dir + y_dir]

poly = Poly3DCollection([points], color='cyan', alpha=0.4, edgecolor=None)
ax1.add_collection3d(poly)

#cam 2
color_random_ = np.random.random(3)
center = c_prime
c_x, c_y, c_z = center
x_dir = dx_x_prime/10
x_x, x_y, x_z = x_dir
y_dir = dx_y_prime/10
y_x, y_y, y_z = y_dir
z_dir = dx_z_prime/10
z_x, z_y, z_z = z_dir
# this is for our direction vectors
ax1.quiver(c_x, c_y, c_z, x_x, x_y, x_z, color='r', linewidth=1)
ax1.quiver(c_x, c_y, c_z, y_x, y_y, y_z, color='g', linewidth=1)
ax1.quiver(c_x, c_y, c_z, z_x, z_y, z_z, color='b', linewidth=1)

# now the camera plane, in here we notice that the corners are made up from addition and substractions of the camera center and the directions,
# so we make use to that
points = [center + x_dir + y_dir,
        center + x_dir - y_dir,
        center - x_dir - y_dir,
        center - x_dir + y_dir]

poly = Poly3DCollection([points], color='purple', alpha=0.4, edgecolor=None)
ax1.add_collection3d(poly)

#ax1.view_init(90,-90)
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

ax1.set_xlim(-1,1)
ax1.set_ylim(-1,1)

plt.savefig(f'reconstruction_cameras.png', bbox_inches='tight', pad_inches=0)
plt.close()


fig = plt.figure(figsize=(11,11))
ax0 = fig.add_subplot(311)
ax1 = fig.add_subplot(312, projection='3d')
ax2 = fig.add_subplot(313)
ms = 5
```

```
lw = 0.5
ax0.imshow(cv2.cvtColor(img_left,cv2.COLOR_BGR2RGB))
ax2.imshow(cv2.cvtColor(img_right,cv2.COLOR_BGR2RGB))
color_random = np.random.random((8,3))
for idx in range(len(world_pts)-8):
    ax1.scatter(world_pts[idx,0],world_pts[idx,1],world_pts[idx,2], marker='o', color='g', s=2)
for idx in range(len(world_pts_og)):
    ax0.plot(img_left_points[idx,0], img_left_points[idx,1], marker="o", color=color_random[idx], markersize=ms)
    ax1.scatter(world_pts_og[idx,0],world_pts_og[idx,1],world_pts_og[idx,2], marker='o', color=color_random[idx], s=7)
    ax2.plot(img_right_points[idx,0], img_right_points[idx,1], marker="o", color=color_random[idx], markersize=ms)


#ax1.view_init(0,0)
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

ax1.set_xlim(-0.3,0.7)
ax1.set_ylim(-0.2,1.1)
ax1.set_zlim(-0.0005,0.0018)

for idx in range(len(world_pts_og)):
    x1, y1, _ = proj3d.proj_transform(world_pts_og[idx,0], world_pts_og[idx,1], world_pts_og[idx,2], ax1.get_proj())
    con = ConnectionPatch(
        xyA=(img_left_points[idx,0], img_left_points[idx,1]),  # Point in the first plot
        xyB=(x1, y1),  # Corresponding point in the second plot
        coordsA="data",      # Interpret xyA in data coordinates of ax1
        coordsB="data",      # Interpret xyB in data coordinates of ax2
        axesA=ax0,           # Reference the first axes
        axesB=ax1,           # Reference the second axes
        color=color_random[idx],        # Line color
        linestyle="--"       # Line style
    )
    # Add the connection to the second plot
    fig.add_artist(con)
    con1 = ConnectionPatch(
        xyA=(img_right_points[idx,0], img_right_points[idx,1]),  # Point in the first plot
        xyB=(x1, y1),  # Corresponding point in the second plot
        coordsA="data",      # Interpret xyA in data coordinates of ax1
        coordsB="data",      # Interpret xyB in data coordinates of ax2
        axesA=ax2,           # Reference the first axes
        axesB=ax1,           # Reference the second axes
        color=color_random[idx],        # Line color
        linestyle="--"       # Line style
    )
    fig.add_artist(con1)
    # Add the connection to the second plot
```

```python

plt.savefig(f'reconstruction_view1.png', bbox_inches='tight', pad_inches=0)
plt.close()

from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
import matplotlib
from matplotlib.patches import ConnectionPatch

fig = plt.figure(figsize=(11,11))
ax0 = fig.add_subplot(311)
ax1 = fig.add_subplot(312, projection='3d')
ax2 = fig.add_subplot(313)
ms = 5
lw = 0.5
ax0.imshow(cv2.cvtColor(img_left,cv2.COLOR_BGR2RGB))
ax2.imshow(cv2.cvtColor(img_right,cv2.COLOR_BGR2RGB))
color_random = np.random.random((8,3))
for idx in range(len(world_pts)-8):
    ax1.scatter(world_pts[idx,0],world_pts[idx,1],world_pts[idx,2], marker='o', color='g', s=2)
for idx in range(len(world_pts_og)):
    ax0.plot(img_left_points[idx,0], img_left_points[idx,1], marker="o", color=color_random[idx], markersize=ms)
    ax1.scatter(world_pts_og[idx,0],world_pts_og[idx,1],world_pts_og[idx,2], marker='o', color=color_random[idx], s=7)
    ax2.plot(img_right_points[idx,0], img_right_points[idx,1], marker="o", color=color_random[idx], markersize=ms)


ax1.view_init(0,0)
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

ax1.set_xlim(-0.3,0.7)
ax1.set_ylim(-0.2,1.1)
ax1.set_zlim(-0.0005,0.0018)

for idx in range(len(world_pts_og)):
    x1, y1, _ = proj3d.proj_transform(world_pts_og[idx,0], world_pts_og[idx,1], world_pts_og[idx,2], ax1.get_proj())
    #ax0.plot([img_left_points[idx,0], x1], [img_left_points[idx,1], y1], color=color_random, linestyle='--')
    con = ConnectionPatch(
        xyA=(img_left_points[idx,0], img_left_points[idx,1]),  # Point in the first plot
        xyB=(x1, y1),  # Corresponding point in the second plot
        coordsA="data",      # Interpret xyA in data coordinates of ax1
        coordsB="data",      # Interpret xyB in data coordinates of ax2
        axesA=ax0,           # Reference the first axes
        axesB=ax1,           # Reference the second axes
        color=color_random[idx],        # Line color
```

```
        linestyle="--"       # Line style
    )
    # Add the connection to the second plot
    fig.add_artist(con)
    con1 = ConnectionPatch(
        xyA=(img_right_points[idx,0], img_right_points[idx,1]),  # Point in the first plot
        xyB=(x1, y1),  # Corresponding point in the second plot
        coordsA="data",      # Interpret xyA in data coordinates of ax1
        coordsB="data",      # Interpret xyB in data coordinates of ax2
        axesA=ax2,           # Reference the first axes
        axesB=ax1,           # Reference the second axes
        color=color_random[idx],        # Line color
        linestyle="--"       # Line style
    )
    fig.add_artist(con1)
    # Add the connection to the second plot

plt.savefig(f'reconstruction_view2.png', bbox_inches='tight', pad_inches=0)
plt.close()

fig = plt.figure(figsize=(11,11))
ax0 = fig.add_subplot(311)
ax1 = fig.add_subplot(312, projection='3d')
ax2 = fig.add_subplot(313)
ms = 5
lw = 0.5
ax0.imshow(cv2.cvtColor(img_left,cv2.COLOR_BGR2RGB))
ax2.imshow(cv2.cvtColor(img_right,cv2.COLOR_BGR2RGB))
color_random = np.random.random((8,3))
for idx in range(len(world_pts)-8):
    ax1.scatter(world_pts[idx,0],world_pts[idx,1],world_pts[idx,2], marker='o', color='g', s=2)
for idx in range(len(world_pts_og)):
    ax0.plot(img_left_points[idx,0], img_left_points[idx,1], marker="o", color=color_random[idx], markersize=ms)
    ax1.scatter(world_pts_og[idx,0],world_pts_og[idx,1],world_pts_og[idx,2], marker='o', color=color_random[idx], s=7)
    ax2.plot(img_right_points[idx,0], img_right_points[idx,1], marker="o", color=color_random[idx], markersize=ms)
    x1, y1, _ = proj3d.proj_transform(world_pts_og[idx,0], world_pts_og[idx,1], world_pts_og[idx,2], ax1.get_proj())
    print(x1,y1)


ax1.view_init(60,45)
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

ax1.set_xlim(-0.001,0.01)
ax1.set_ylim(0.002,0.010)
ax1.set_zlim(0,0.00002)
```

```python

for idx in range(len(world_pts_og)):
    x1, y1, _ = proj3d.proj_transform(world_pts_og[idx,0], world_pts_og[idx,1], world_pts_og[idx,2], ax1.get_proj())
    if x1 < 0.0711:
        #ax0.plot([img_left_points[idx,0], x1], [img_left_points[idx,1], y1], color=color_random, linestyle='--')
        con = ConnectionPatch(
            xyA=(img_left_points[idx,0], img_left_points[idx,1]),  # Point in the first plot
            xyB=(x1, y1),  # Corresponding point in the second plot
            coordsA="data",      # Interpret xyA in data coordinates of ax1
            coordsB="data",      # Interpret xyB in data coordinates of ax2
            axesA=ax0,           # Reference the first axes
            axesB=ax1,           # Reference the second axes
            color=color_random[idx],        # Line color
            linestyle="--"       # Line style
        )
        # Add the connection to the second plot
        fig.add_artist(con)
        con1 = ConnectionPatch(
            xyA=(img_right_points[idx,0], img_right_points[idx,1]),  # Point in the first plot
            xyB=(x1, y1),  # Corresponding point in the second plot
            coordsA="data",      # Interpret xyA in data coordinates of ax1
            coordsB="data",      # Interpret xyB in data coordinates of ax2
            axesA=ax2,           # Reference the first axes
            axesB=ax1,           # Reference the second axes
            color=color_random[idx],        # Line color
            linestyle="--"       # Line style
        )
        fig.add_artist(con1)
    # Add the connection to the second plot

plt.savefig(f'reconstruction_view3.png', bbox_inches='tight', pad_inches=0)
plt.close()

##### THIS IS FOR TASK 3
def census_transformation(img_left, img_right, window_size, d_Max, left=True):
    # we want to pad both images, to not lose out on those borders pixels, however since we are moving from 0 to d_Max we need to consider d_Max in our padding
    # so then the logical choice is to pad by d_Max + half the window size
    h,w = img_left.shape
    disp_map = np.zeros_like(img_left)
    border_size = d_Max + window_size//2
    # technically, if want to compute the left disparity match, we only need to pad the right side of the left image, and the left side of the
```

```
right image, as we go from the position to d_Max in the right image
 # but since this implementation can be used for both the left or right image, we pad left and right sides.
 # As for the top and bottom padding, we only need to make these half the window size, since the don't "move" in that direction
 img_right = cv2.copyMakeBorder(img_right, window_size//2, window_size//2, border_size, border_size, cv2.BORDER_CONSTANT, 0)
 img_left = cv2.copyMakeBorder(img_left, window_size//2, window_size//2, border_size, border_size, cv2.BORDER_CONSTANT, 0)
 # the way to compute the disparity is a bit different from left and right, so we make the choice here
 if left:
     # like I mentioned, the rows will go from window_size//2 to img_left.shape[0] - window_size//2, to account for the padding
     for jdx in range(window_size//2, h+window_size//2):
         # for the x position, we need to consider the border size we had mentioned,
         for idx in range(border_size, w+border_size):
             # save each error in this array
             # get the left window
             left_window = img_left[(jdx - window_size//2):(jdx + window_size//2 + 1), (idx - window_size//2):(idx + window_size//2 + 1)]
             center_L = img_left[jdx,idx]
             left_window = left_window.ravel()
             # this is our descriptor bitvector
             bit_L = left_window > center_L
             err = []
             for d in range(d_Max+1):
                 # this is where we go from 0 to d_Max in the right image, notice that only the x direction is changing
                 right_window = img_right[(jdx - window_size//2):(jdx + window_size//2 + 1),(idx - d - window_size//2):(idx - d + window_size//2 +1)]
                 right_window = right_window.ravel()
                 center_R = img_right[jdx,idx - d]
                 bit_R = right_window > center_R
                 # logical xor

                 diff = np.logical_xor(bit_L, bit_R)
                 s_diff = diff.sum()
                 err.append(s_diff)
             # and now all that is left is to take the minimum sum value
             disp_map[jdx-window_size//2,idx-border_size] = np.argmin(np.array(err))
 else:
     # we do a similar thing for the right image, however the difference is that in this case, we move in the opposite direction for 0 to d_Max
     for jdx in range(window_size//2, h+window_size//2):
         for idx in range(border_size, w+border_size):
             err = []
             right_window = img_right[(jdx - window_size//2):(jdx + window_size//2 + 1), (idx - window_size//2):(idx + window_size//2 + 1)]
```

```
                center_R = img_right[jdx,idx]
                right_window = right_window.ravel()
                bit_R = right_window > center_R
                err = []
                for d in range(d_Max+1):
                    left_window = img_left[(jdx - window_size//2):(jdx + window_size//2 + 1),(idx + d - window_size//2):(idx + d + window_size//2 +1)]
                    left_window = left_window.ravel()
                    center_L = img_left[jdx,idx + d]
                    bit_L = left_window > center_L

                    diff = np.logical_xor(bit_R, bit_L)
                    s_diff = diff.sum()
                    err.append(s_diff)
                disp_map[jdx-window_size//2,idx-border_size] = np.argmin(np.array(err))
    return disp_map.astype(np.uint8)

def check_error_disp(disparity_map, gt_map, delta=2):
    # get valid positions

    # Now the mask
    disp_mask = np.zeros_like(gt_map).astype(np.uint8)
    diffe = np.abs(disparity_map - gt_map)
    disp_mask[diffe < delta] = 255
    disp_mask[gt_map == 0] = 0

    #acc = (disp_mask == 255).sum()
    #acc = acc/(gt_map.shape[0]*gt_map.shape[1])
    acc = np.count_nonzero(disp_mask)/np.count_nonzero(gt_map)

    return acc, disp_mask


image_left = cv2.imread("Task3Images/im2.png", cv2.IMREAD_GRAYSCALE)
image_right = cv2.imread("Task3Images/im6.png", cv2.IMREAD_GRAYSCALE)

gt_left = cv2.imread("Task3Images/disp2.png", cv2.IMREAD_GRAYSCALE)
gt_right = cv2.imread("Task3Images/disp6.png", cv2.IMREAD_GRAYSCALE)

gt_left = gt_left.astype(np.float32)/4
gt_left = gt_left.astype(np.uint8)

gt_right = gt_right.astype(np.float32)/4
gt_right = gt_right.astype(np.uint8)

d_max_left = gt_left.max()
d_max_right = gt_right.max()


window_sizes = [5,15,25,35]
for w in window_sizes:
    print("-"*10)
```

```
    print(f"Window size is {w}")
    dispmap_left = census_transformation(image_left, image_right, w, d_max_left, left=True)
    dispmap_right = census_transformation(image_left, image_right, w, d_max_right, left=False)
    acc_left, disp_mask_left = check_error_disp(dispmap_left, gt_left, delta=2)
    acc_right, disp_mask_right = check_error_disp(dispmap_right, gt_right, delta=2)

    print(f"Left accuracy is {acc_left}")
    print(f"Right accuracy is {acc_right}")

    plt.imshow(dispmap_left,cmap="gray")
    plt.axis("off")
    plt.savefig(f'images_report/disparity_map_left_wsize{w}.png', bbox_inches='tight', pad_inches=0)
    plt.close()

    plt.imshow(dispmap_right,cmap="gray")
    plt.axis("off")
    plt.savefig(f'images_report/disparity_map_right_wsize{w}.png', bbox_inches='tight', pad_inches=0)
    plt.close()

    plt.imshow(disp_mask_left,cmap="gray")
    plt.axis("off")
    plt.savefig(f'images_report/left_diff_wsize{w}.png', bbox_inches='tight', pad_inches=0)
    plt.close()

    plt.imshow(disp_mask_right,cmap="gray")
    plt.axis("off")
    plt.savefig(f'images_report/right_diff_wsize{w}.png', bbox_inches='tight', pad_inches=0)
    plt.close()
```

Listing 1: Source code for Task 1 & 3

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

"""
A few notes on the scene_info dictionary:
- depth maps are stored as h5 files. Depth is the distance of the object from the camera (ie Z coordinate in camera coordinates). The depth map can contain invalid points (depth = 0) which correspond to points where the depth could not be estimated.
- The intrinsics are stored as a 3x3 matrix.
- The poses [R,t] are stored as a 4x4 matrix to allow for easy transformation of points from one camera to the other. The resulting transformation matrix is a 4x4 matrix is of the form:
```

```
    T = [[R, t]
        [0, 1]] where R is a 3x3 rotation matrix and t is a 3x1 translation vector.
"""

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()

if __name__ == "__main__":
    scene_info = pkl.load(open('./data/scene_info/1589_subset.pkl', 'rb'))

    for i_pair in range(len(scene_info)):
        # 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)
        xx = xx.ravel()[:, None]
        yy = yy.ravel()[:, None]
        msh = np.hstack((xx,yy))
        # make homogeneous coordinates for points0 #[3, N]
        """
        <Student code>
        points0 = ...
        """

        points0 = np.append(msh,np.ones(msh.shape[:-1])[...,None],axis=-1).astype(int)
        #(2) get depth values at points0
        """
        <Student code>
        depth_values0 = ...
```

```
        """
        points0_int = points0[:,:2]
        depth_values0 = []
        for point in points0_int:
            depth_values0.append(depth0[point[1],point[0]])
        depth_values0 = np.array(depth_values0)
        #print(depth_values0)

        # remove points with depth 0 (invalid points)
        """
        <Student code>
        valid_points = ...
        # mask points0 and depth_values0
        """
        valid_points = np.argwhere(depth_values0 > 0.0)
        points0 = points0[valid_points][:,0,:]
        depth_values0 = depth_values0[valid_points][:,0]
        #print(depth_values0.shape)
        #print(points0.shape)

        # (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)
        #print(K0,K0_inv)
        xyz_cam0 = (K0_inv @ points0.T)
        #xyz_cam0 /= xyz_cam0[2,:]
        xyz_cam0 = xyz_cam0[:,:].T
        xyz_cam0 *= depth_values0[:,None]
        #print(xyz_cam0)
        xyz_cam0_hc = np.append(xyz_cam0,np.ones(xyz_cam0.shape[:-1])[...,
    None],axis=-1)
        #print(xyz_cam0_hc)

        xyz_world_hc = (np.linalg.inv(T0) @ xyz_cam0_hc.T)
        #xyz_world_hc /= xyz_world_hc[3,:]
        xyz_world_hc = xyz_world_hc.T
        #print(xyz_world_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.T).astype(float)
        #xyz_cam1_hc /= xyz_cam1_hc[3,:]
        xyz_cam1_hc = xyz_cam1_hc.T
        #print(xyz_cam1_hc)
        xyz_cam1 = xyz_cam1_hc[:,:3]
        estimated_depth_values1 = xyz_cam1[:,2]
        #print(xyz_cam1)
        #xyz_cam1[:,2] = 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 = ...
        """
        # print(xyz_cam1)
        K1 = scene_info[i_pair]['K1']
        points1 = (K1 @ xyz_cam1.T)
        points1 /= points1[2,:]
        points1 = points1.T
        #print(points1)
        #withinbounds = np.argwhere(0 <= points1[:,1] < img1.shape[0] and 0 <= points1[:,0] < img1.shape[1])
        withinbounds_x = points1[:,0] < img1.shape[1]
        withinbounds_x0 = 0 <= points1[:,0]


        withinbounds_y = points1[:,1] < img1.shape[0]
        withinbounds_y0 = 0 <= points1[:,1]
        withinbounds_ = np.logical_and(withinbounds_x,withinbounds_x0,)
        withinbounds__ = np.logical_and(withinbounds_y,withinbounds_y0)
        withinbounds = np.logical_and(withinbounds_, withinbounds__)
        withinbounds_ids = np.argwhere(withinbounds)
        points1 = points1[withinbounds_ids][:,0,:]
        estimated_depth_values1 = estimated_depth_values1[withinbounds_ids][:,0]
        #print(points1.shape, estimated_depth_values1.shape)
        true_depth_values1 = []
        for point in points1.astype(int):
            true_depth_values1.append(depth1[point[1],point[0]])
```

```
        true_depth_values1 = np.array(true_depth_values1)
        #print(withinbounds_ids)
        #swap x and y
        points0 = points0.T
        points1 = points1.T

        #print(points0.shape)

        # (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'../pics/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>
        ...
        """
        """K0_inv = np.linalg.inv(K0)
        #print(K0,K0_inv)
        xyz_cam0 = (K0_inv @ points0.T)
        #xyz_cam0 /= xyz_cam0[2,:]
        xyz_cam0 = xyz_cam0[:,:].T
        xyz_cam0 *= depth_values0[:,None]
        #print(xyz_cam0)
        xyz_cam0_hc = np.append(xyz_cam0,np.ones(xyz_cam0.shape[:-1])[...,None],axis=-1)
        #print(xyz_cam0_hc)

        xyz_world_hc = (np.linalg.inv(T0) @ xyz_cam0_hc.T)
        xyz_world_hc = xyz_world_hc.T"""
```

```
        n_points = 100
        x = np.linspace(10, img0.shape[1]-10, n_points) # ignore a border of 10 pxls
        """
        <Student code>
        # meshgrid of x and y coordinates
        xx, yy = ...
        """
        y = np.linspace(10, img0.shape[0]-10, n_points)
        xx, yy = np.meshgrid(x, y)
        xx = xx.ravel()[:, None]
        yy = yy.ravel()[:, None]
        msh = np.hstack((xx,yy))
        points0 = np.append(msh,np.ones(msh.shape[:-1])[...,None],axis=-1).astype(int)
        points0_int = points0[:,:2]
        depth_values0 = []
        for point in points0_int:
            depth_values0.append(depth0[point[1],point[0]])
        depth_values0 = np.array(depth_values0)
        valid_points = np.argwhere(depth_values0 > 0.0)
        points0 = points0[valid_points][:,0,:]
        depth_values0 = depth_values0[valid_points][:,0]

        K0_inv = np.linalg.inv(K0)
        #print(K0,K0_inv)
        xyz_cam0 = (K0_inv @ points0.T)
        #xyz_cam0 /= xyz_cam0[2,:]
        xyz_cam0 = xyz_cam0[:,:].T
        xyz_cam0 *= depth_values0[:,None]
        #print(xyz_cam0)
        xyz_cam0_hc = np.append(xyz_cam0,np.ones(xyz_cam0.shape[:-1])[...,None],axis=-1)
        #print(xyz_cam0_hc)

        xyz_world_hc = (np.linalg.inv(T0) @ xyz_cam0_hc.T)
        xyz_world_hc = xyz_world_hc.T


        x = np.linspace(10, img1.shape[1]-10, n_points) # ignore a border of 10 pxls
        """
        <Student code>
        # meshgrid of x and y coordinates
        xx, yy = ...
        """
        y = np.linspace(10, img1.shape[0]-10, n_points)
        xx, yy = np.meshgrid(x, y)
        xx = xx.ravel()[:, None]
        yy = yy.ravel()[:, None]
        msh = np.hstack((xx,yy))
        points1 = np.append(msh,np.ones(msh.shape[:-1])[...,None],axis=-1).astype(int)
        points1_int = points1[:,:2]
```

```
        depth_values1 = []
        for point in points1_int:
            depth_values1.append(depth1[point[1],point[0]])
        depth_values1 = np.array(depth_values1)
        valid_points = np.argwhere(depth_values1 > 0.0)
        points1 = points1[valid_points][:,0,:]
        depth_values1 = depth_values1[valid_points][:,0]

        K1_inv = np.linalg.inv(K1)
        #print(K0,K0_inv)
        xyz_cam1 = (K1_inv @ points1.T)
        #xyz_cam0 /= xyz_cam0[2,:]
        xyz_cam1 = xyz_cam1[:,:].T
        xyz_cam1 *= depth_values1[:,None]
        #print(xyz_cam0)
        xyz_cam1_hc = np.append(xyz_cam1,np.ones(xyz_cam1.shape[:-1])[...,
    None],axis=-1)
        #print(xyz_cam0_hc)

        xyz_world_hc_b = (np.linalg.inv(T1) @ xyz_cam1_hc.T)
        xyz_world_hc_b = xyz_world_hc_b.T
        #print(xyz_world_hc)
        # plot X_world^A
        fig = plt.figure(figsize=(11,11))
        ax = fig.add_subplot(projection='3d')
        for idx in range(len(xyz_world_hc)):
            ax.scatter(xyz_world_hc[idx,0],xyz_world_hc[idx,2],
    xyz_world_hc[idx,1], marker='o', color='g', s=1)
        for idx in range(len(xyz_world_hc_b)):
            ax.scatter(xyz_world_hc_b[idx,0],xyz_world_hc_b[idx,2],
    xyz_world_hc_b[idx,1], marker='o', color='g', s=1)
        #ax.view_init(90,-90)
        ax.set_xlabel('X Label')
        ax.set_ylabel('Y Label')
        ax.set_zlabel('Z Label')
        plt.savefig(f'../pics/depth_3d_{i_pair}.png', bbox_inches='tight',
     pad_inches=0)

        plt.close()
```

Listing 2: Source code for Task 4