# ECE 66100 Homework #9

# by


# Adrien Dubois (dubois6@purdue.edu)


December 3, 2024

## Contents

# 1 Theory:

## 1.1 Task 1: Image Rectification

Image rectification involves a left and right images that are facing the same object, and point correspondences of that object. We denote points on the left imaging plane as $\{x_1, x_2, ..., x_n\}$ and pixel coordinates on the right image as $\{x'_1, x'_2, ..., x'_n\}$.

These pixel coordinates are related to one another through a $3 \times 3$ fundamental matrix $\boldsymbol{F}$:

$$x'Fx = 0$$

where:

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

By expanding the previous equation, initial least-squares estimate of this fundemantal matrix can therefore be calculated as follows:

$$\begin{bmatrix} x_1x'_1 & x'_1y_1 & x'_1 & x_1y'_1 & y_1y'_1 & y'_1 & x_1 & y_1 & 1 \\ x_2x'_2 & x'_2y_2 & x'_2 & x_2y'_1 & y_2y'_2 & y'_2 & x_2 & y_2 & 1 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... \\ x_nx'_n & x'_ny_n & x'_n & x_ny'_n & y_ny'_n & y'_n & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = 0$$

We would therefore need at least 8 $\boldsymbol{x}$ and $\boldsymbol{x'}$ correspondences to estimate an initial solution for F, though using closer to 20 proved more successful for my implementation.

The next step in image rectification requires us to convert the cameras to canonical form. This involves centering the left camera on the 0,0 point, and making the imaging planes parallel to one another vis-a-vis the imaged object. To do this, we will first need to calculate the projective matrices $P$ and $P'$ for the left and right images respectively. By definition in canonical form, we know that the parameters for the left camera are:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \text{ and } e_p = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

For the right camera, we rely on the fundamental matrix. We know that $F^T e'_p = 0$, so the right epipole is a null vector of the fundamental matrix. Then, we can calculate the right projection matrix as follows:

$$P' = \begin{bmatrix} \begin{bmatrix} e' \end{bmatrix}_X F & |e' \end{bmatrix}$$

where $\begin{bmatrix} e' \end{bmatrix}$ is denoted as the skew-symmetric or cross product representation of $e'$:

$$\begin{bmatrix} e' \end{bmatrix}_X = \begin{bmatrix} 0 & -e'_3 & e'_2 \\ e'_3 & 0 & -e'_1 \\ -e'_2 & e'_1 & 0 \end{bmatrix}$$

One issue with this estimation of $P'$ is that it is based on a linear least-squares estimate of the fundamental matrix. Instead we can use the Levenberg-Marquadt algorithm to better refine P'. To do so, we must first calculate the world points created by the intersection of the rays passing through the centers of projections of each camera and the image pixel correspondences. We do so through the following equation, where $P_i$ will denote the i-th row of the P matrix.

$$\begin{bmatrix} x_iP_3^T - P_1^T \\ y_iP_3^T - P_2^T \\ x'_iP_3'^T - P_1'^T \\ y'_iP_3'^T - P_1'^T \end{bmatrix} X_i = 0$$

This gives us a 4 dimensional homogenous coordinate which we can convert to physical coordinate by dividing by the last element. After computing this initial linear least-squares estimate for the world coordinates, we can pass the following paramters to the L.M. optimizer:

$$\text{learnable parameters: } \begin{bmatrix} P' & X_1 & X_2 & ... & X_N \end{bmatrix}$$

And we minimize the following geometric cost function:

$$\epsilon_{geo} = \sum_{i=1}^{N} \left( ||x_i - \hat{x}_i||^2 + ||x_i' - \hat{x}_i'||^2 \right)$$

where $\hat{x}$ and $\hat{x}'$ are the back-projected world coordinates to the respective image planes:

$$\hat{x}_i = PX_i$$

$$\hat{x}_i' = P'X_i$$

After acquiring a non-linear estimate for $P'$, also know the right epipole to be its fourth column due to the canonical form setup. Additionally, we can refine our estimate for F using this improved $P'$:

$$F = \begin{bmatrix} e' \end{bmatrix}_X P'P+$$

Next, we now calculate the homographies that would rectify the left and right images. This would make it so that corresponding points in the left and right images can be found on the same rows. We first calculate the right homography:

1. $\theta = arctan2(-\left(e_2' - \frac{h}{2}\right), -\left(e_1' - \frac{w}{2}\right))$
2. $R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$
3. $\tilde{e} = R\tilde{e}' = \begin{bmatrix} \tilde{e}_1 & \tilde{e}_2 & \tilde{e}_3 \end{bmatrix}$
4. $f = \frac{\tilde{e}_1}{\tilde{e}_3}$
5. $G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{-1}{f} & 0 & 1 \end{bmatrix}$
6. $T_2 = \begin{bmatrix} 1 & 0 & \frac{w}{2} \\ 0 & 1 & \frac{h}{2} \\ 0 & 0 & 1 \end{bmatrix}$
7. $T = \begin{bmatrix} 1 & 0 & -\frac{w}{2} \\ 0 & 1 & -\frac{h}{2} \\ 0 & 0 & 1 \end{bmatrix}$
8. $H = T_2GRT$

On the other hand, to compute the left rectification homography, we follow these steps:

1. $M = P'P+$
2. $H_0 = H'M$
3. $H_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
4. The parameters a, b, and c are calculated through the following manner:
   (a) $z_i = H_0 x_i = \begin{bmatrix} \xi_i & \nu_i & 1 \end{bmatrix}$
   (b) $z_i' = H'x_i' = \begin{bmatrix} \xi_i' & \nu_i' & 1 \end{bmatrix}$

(c) Solve the following linear least-square equation:

$$\begin{bmatrix} \xi_1 & \nu_1 & 1 \\ ... & ... & ... \\ \xi_N & \nu_N & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \xi_1' \\ \nu_1' \\ 1 \end{bmatrix}$$

5. $H = H_A H_0$

6. Normalize the homographies by their last entry ($h_{3,3}$)

It is important to note that this code only produced solid results when all homogenous matrices were normalized. I did this by dividing all definitions of $e', P, P', H, H', z_i, z_i'$ by their scaling coordinate (the last entry in each vector/matrix).

Additionally, I used a canny operator to create an edge map from the rectified image. It is important to note that there was significant noise in extracting matching points from this edge map due to the canny operator extracting the border of the images. Therefore, I used a HoughLinesP transform to extract the straight lines from these edge maps, and inpainted the lines in the edge map with black pixels to erase them. This helped to reduce the amount of matching points found on the image borders when using SSD.

Additionally, since I know that I am working on rectified images, I prune the points to have a low absolute difference in width ($diff\,(|x_1 - x_2|) < 4$) and absolute difference in height ($diff\,(|y_1 - y_2|) < 4$) value. It is clear that setting an upper bound height difference between the coordinate matches would work since both images are rectified. However, I found anecdotally that my images were also located in around the same location in the camera plane so I set a much looser boundary on those points to also speed up the computation of SSD.

Lastly, we need to create the 3D reconstruction from the point matches on the rectified edge map. First, I remapped the point matches to the original, unrectified images by applying the corresponding inverse homography. Then, I re-ran the steps described above up until the LM optimization. Then, I do one last world point reconstruction using linear least-squares with the final P and $P'$ matrices and plot those points in 3 dimensions.

## 1.2 Task 2: Loop and Zhang's Algorithm

The Loop and Zhang's algorithm works by decomposing the rectifying homographies calculated just above into three seperate components:

$$H = H_{sh} H_{sim} H_p \text{ and } H' = H'_{sh} H'_{sim} H'_p$$

where $H_{sh}$ is a shearing homography, $H_{sim}$ is a similarity transformation and $H_p$ is the purely projective portion of the rectifying homography. The role of $H_p$ is therefore to send the corresponding epipole to infinity in each image plane. On the other hand, Next, the similarity transformations will rotate the epipoles to coincide with the world x-axis as required by canonical form. Lastly, the shear homographies are there to reduce, as much as possible, the projective distortion.

## 1.3 Task 3: Dense Stereo Matching using the Census Transform

Dense stereo matching involves finding per-pixel correspondences in a given stereo-rectified image pair. In this way, we calculate a disparity map by searching through all pixels in the right image for the one that matches the pixel in the left. We then define the disparity for that pixel as the difference in the x-coordinates between the pair.

More formally, we define the problem as such:

1. For each pixel in the left image $(x_i, y_i)$, we define a candidate pixel in the same row by: $(x_i' - d, y_i')$

2. For each candidate pixel, as scanned along the row indexed by $y_i$, we consider a $M \times M$ window centered on $(x_i' - d, y_i')$

3. We then create two bitvectors of shape $M \times M$ (one for each images). The values in these bitvectors are defined as follows:

    - 1 if the pixel in the window is stricly larger than the center pixel

- 0 otherwise

4. We then XOR these two bitvectors together and the sum of the digits of the resulting bitvector (the number of ones) is our data cost

5. By finding the x-coordinate that minimizes the data-cost, we can set the correct disparity value d and create the disparity map accordingly.

### 1.4 Task 4: Dense Correspondences using Depth Maps

*Explain in your own words the automatic extraction of dense correspondences using depth maps and why it is useful* A depth map is created by calculating the distance between each real world coordinate in the image, and the camera's principle plane. These are especially useful to real time navigation based computer vision such as person-tracking or autonomous vehicles. In this way, we can use these depth maps to check whether our extracted dense correspondences are accurate.

This is done by first calculating an inverse of the camera matrix $K_A$:

$$K^{-1} = \begin{bmatrix} \frac{1}{\alpha_x} & -\frac{s}{\alpha_x \alpha_y} & \frac{sy_0 - \alpha_y x_0}{\alpha_x \alpha_y} \\ 0 & \frac{1}{\alpha_y} & -\frac{y_0}{\alpha_y} \\ 0 & 0 & 1 \end{bmatrix}$$

We can then calculate the 3D coordinate of any point on the ray corresponding to the camera pixel $x_i$:

$$X_{cam} = K^{-1} x_i$$

We can get then get this 3D coordinate in terms of the image coordinate frame by multiplying it by its depth value at that point. Afterwards, using the rotation and translation matrices we convert the 3D image coordinate to the world coordinate frame.

$$X_{worldhc} = T^{-1} X_{camhc}$$

Next, we estimate the depth value in the right image by calculating $X^B_{cam} = \begin{bmatrix} R_B & |t_B \end{bmatrix} X_{world}$. The Z value, is therefore this estimated depth value. We can also determine the true depth value by projecting the 3D world coordinate onto image B and getting the depth value at that point.

It is the estimated depth coordinate, and the true depth coordinate that we compare for accuracy. If the difference between the two points is less than a given threshold, we consider the two points correspondences.

## 2 Results: Task 1

### 2.1 Manually Selected Points:

It is important to note that my implementation of stereo rectification ended up being much more robust to large angular changes compared to the provided Loop and Zhang code. Therefore, the angle between the images looks very small so that Loop and Zhang works. For demonstration, I will also include an image with a much larger difference between the left and right images in the rectification section; however, since that image did not work for Loop and Zhang I do not report its results everywhere.


(a) Points chosen for the left image


(b) Points chosen for the right image

## 2.2 Improvement in F caused by LM:

To check whether my algorithm was working correctly, I calculated $x_i' F x_i$ for all points i and report the mean absolute value. Therefore, I could confirm that my LM was working not just through the decreased cost function, but also by this error value getting closer to 0.

| **Metric** | **Before LM** | **After LM** |
|---|---|---|
| Cost | 1.6670e+02 | 3.3692e+00 |
| Error | 0.086739 | 0.0753223 |

Table 1: Comparison of metrics before and after applying LM.

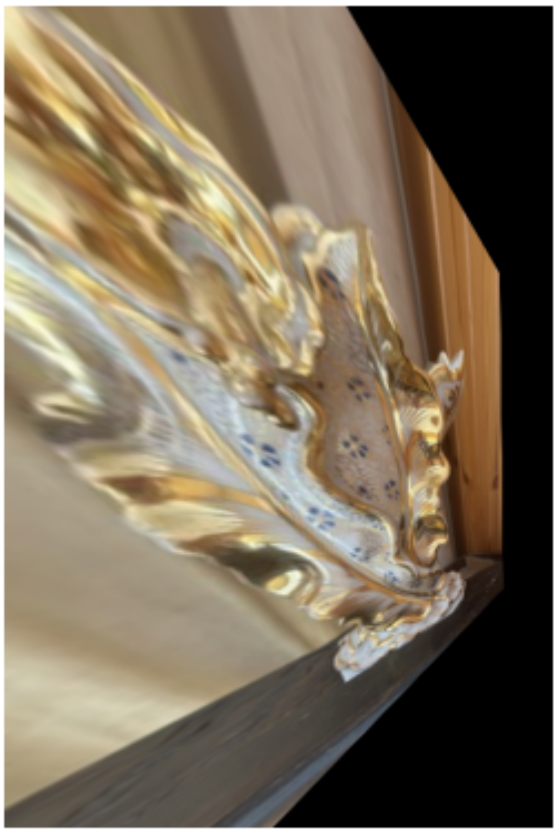
## 2.3 Stereo images before and after rectification:

### 2.3.1 Before Rectification:

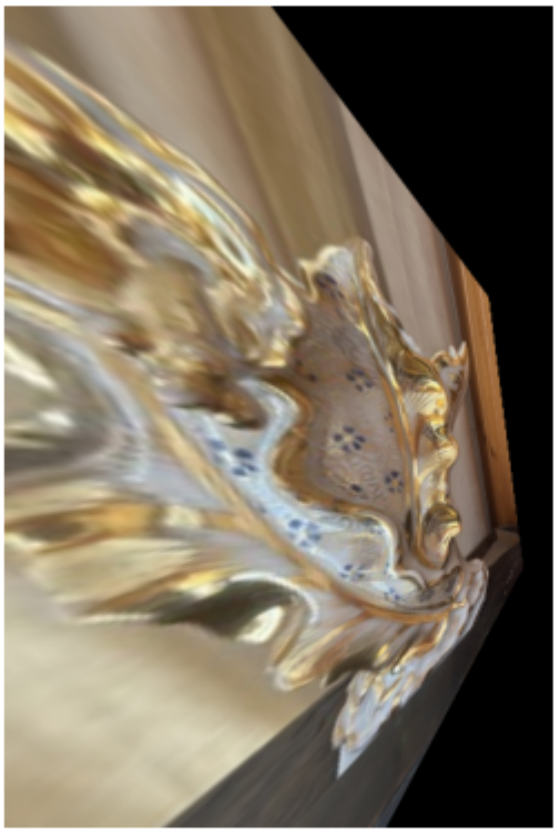
(a) Left image before rectification

(b) Right image before rectification

### 2.3.2 Post Rectification:



(a) Left image after rectification



(b) Right image after rectification

Please note that I applied a translation homography to these images for display; however, this was not applied for the next images to stay consistent with the instructions.

## 2.4 Point Matches

To support my point that my algorithm was more robust to larger angle changes than the Loop and Zhang implementation, I also report point matches with another abject, and greater rotation, which caused errors with the provided Loop and Zhang code.

### 2.4.1 Rectified point matches using a small angle

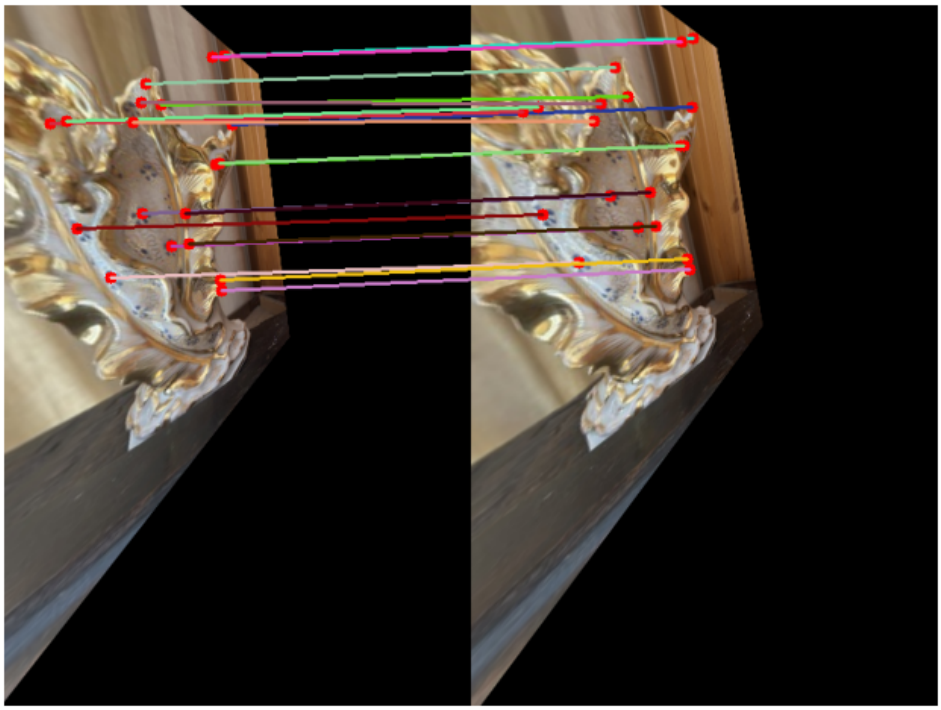


Figure 4: Point matches printed on the rectified images with a small angle between the left and right images.

### 2.4.2 Rectified point matches using a large angle

Original Images:

(a) Left image before rectification

(b) Right image before rectification

Rectified Images:

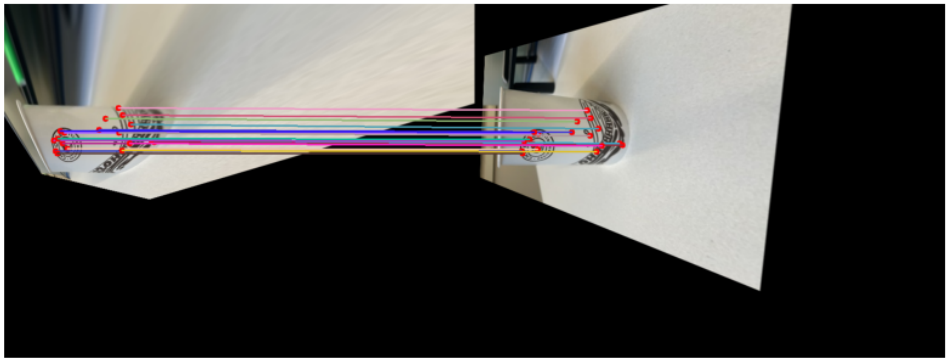


Figure 6: Point matches printed on the rectified images with a large angle between the left and right images.

## 2.5 Correspondences using the Canny operator on rectified images:

### 2.5.1 Output of the canny operator:

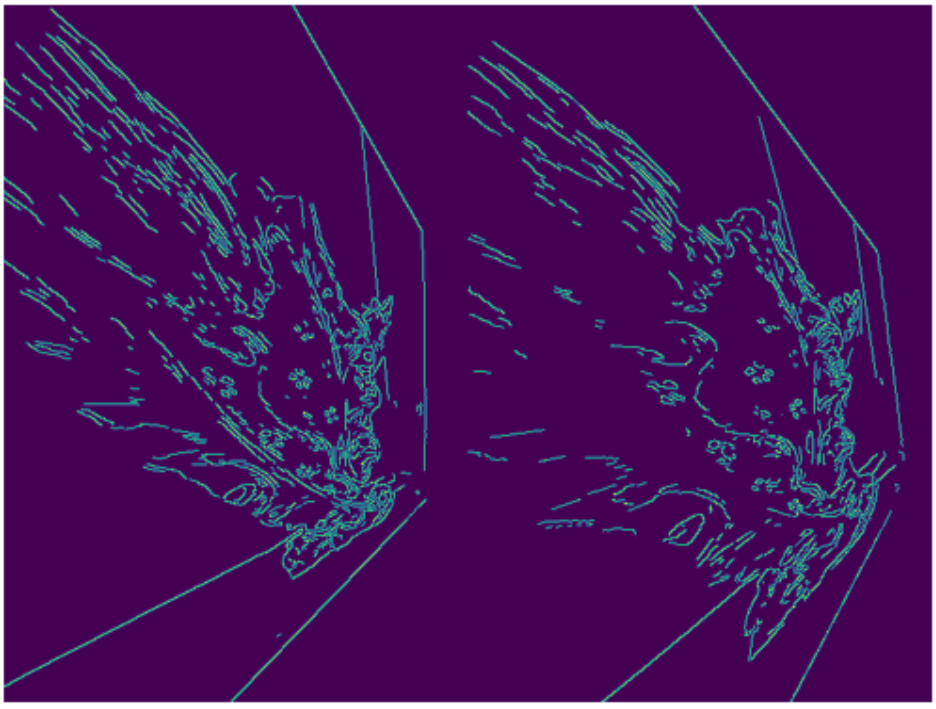


Figure 7: Output of the canny operator with thresholds 150, 200 applied on the rectified image pairs.

### 2.5.2 Removal of lines on the output of the Canny operator using HoughLinesP:

After applying the Canny operator to get the edges of the image, I used my previous implementation of the SSD point matching algorithm to collect 200 point matches between the two images. It is important to note that the canny operator had one main issue in this context which was that it extract lines at the borders of the images due to the perspective transform we applied. This means that all of the top 200 point matches actually occured on the edges of the image instead of the object in question. This issue is demonstrated in the image below.

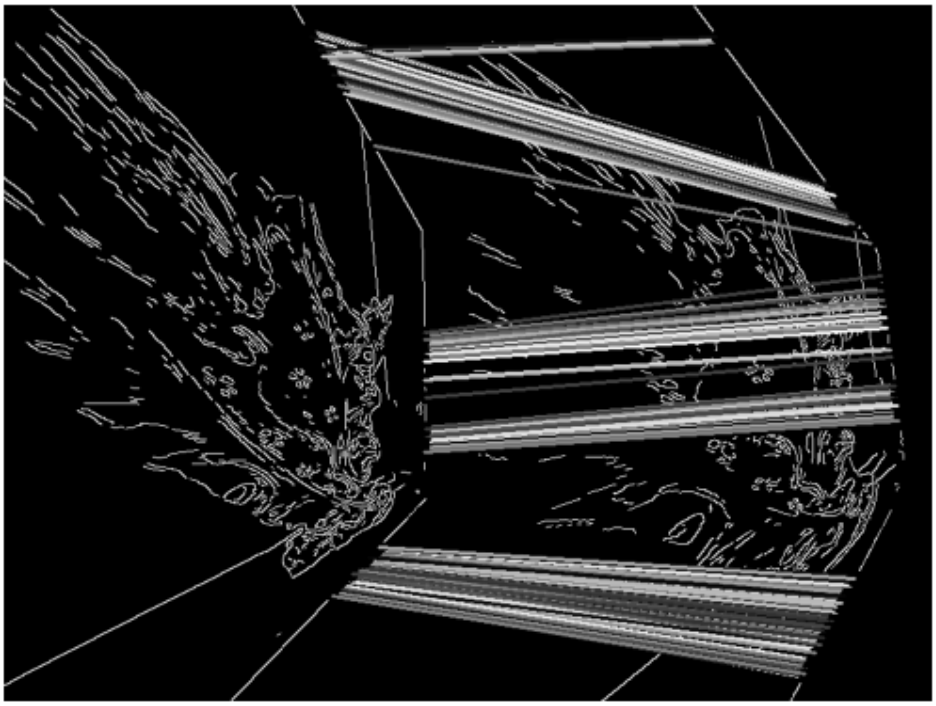


Figure 8: Top 200 point matches from the edged images.

However, to get a good 3D reconstruction of the object, we obviously want the point matches to be detected on the object in question and not on the boders of the image. Therefore, we use a HoughLine transform to detect the location of straight lines on the image, and remove these lines from the edge image using inpainting with black pixels. With this method, I purposefully over-remove lines from the image to ensure that as much of the image border edges are removed. In this way, not point matches will be detected on the borders, and there are still more than enough points left in the canny edge image for hundreds of solid point matches.

The parameters I used for HoughLinesP to remove the lines were:

| **Parameter** | **Value** |
|---|---|
| `rho` | 1 (pixel resolution) |
| `theta` | $\pi/180$ |
| `threshold` | 5 |
| `minLineLength` | 40 (minimum line length in pixels) |
| `maxLineGap` | 10 (maximum gap between segments to connect as a single line) |

Table 2: Parameters used in the `cv2.HoughLinesP` function call.

and the resulting edge map was:

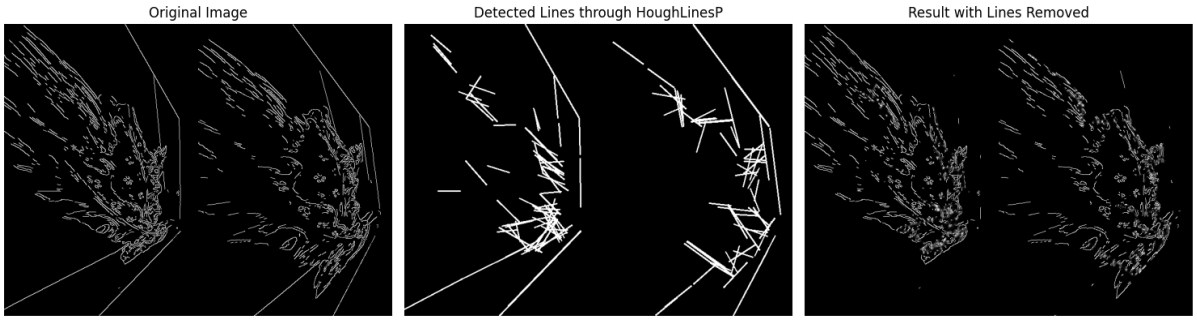



Figure 9: The original edges, the lines removed and the final edges with straight lines removed.

### 2.5.3 Point matches using SSD on the Canny edges:

Included below are the point matches found before and after removing the lines using HoughLinesP. For the first image, I report 200 matches while I report 1000 matches in the final image with lines removed.

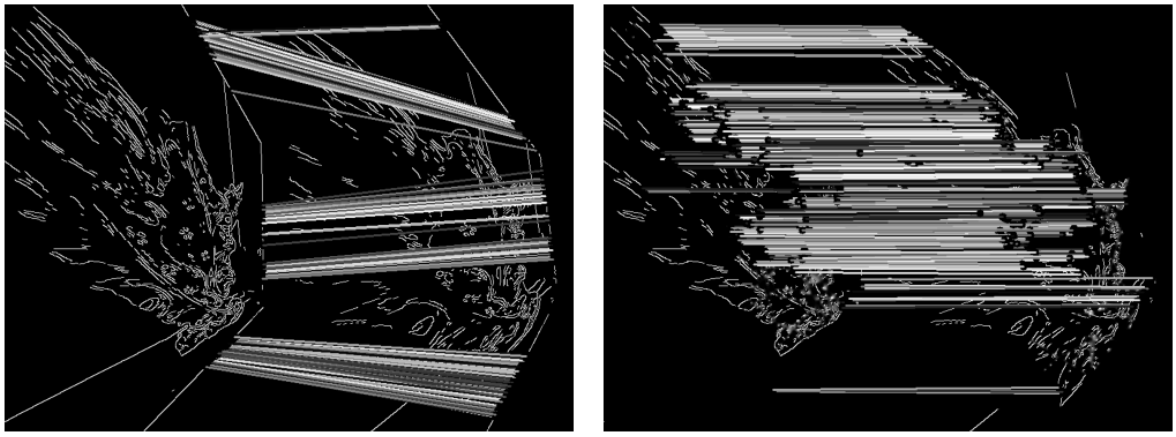


Figure 10: The the point matches before (left) and after (right) removing the image borders using houghlinesP.

### 2.5.4 Point matches from the warped scene mapped back to the original images

By applying the inverse homographies ($H^{-1}$ and $H^{'-1}$) to the point matches on the rectified images, we can find their corresponding locations in the original images. The result of this operation is shown below. This allows us to re-run the previous operations completed in task one but with almost 1000 matching points instead of around 20 manually selected matches.

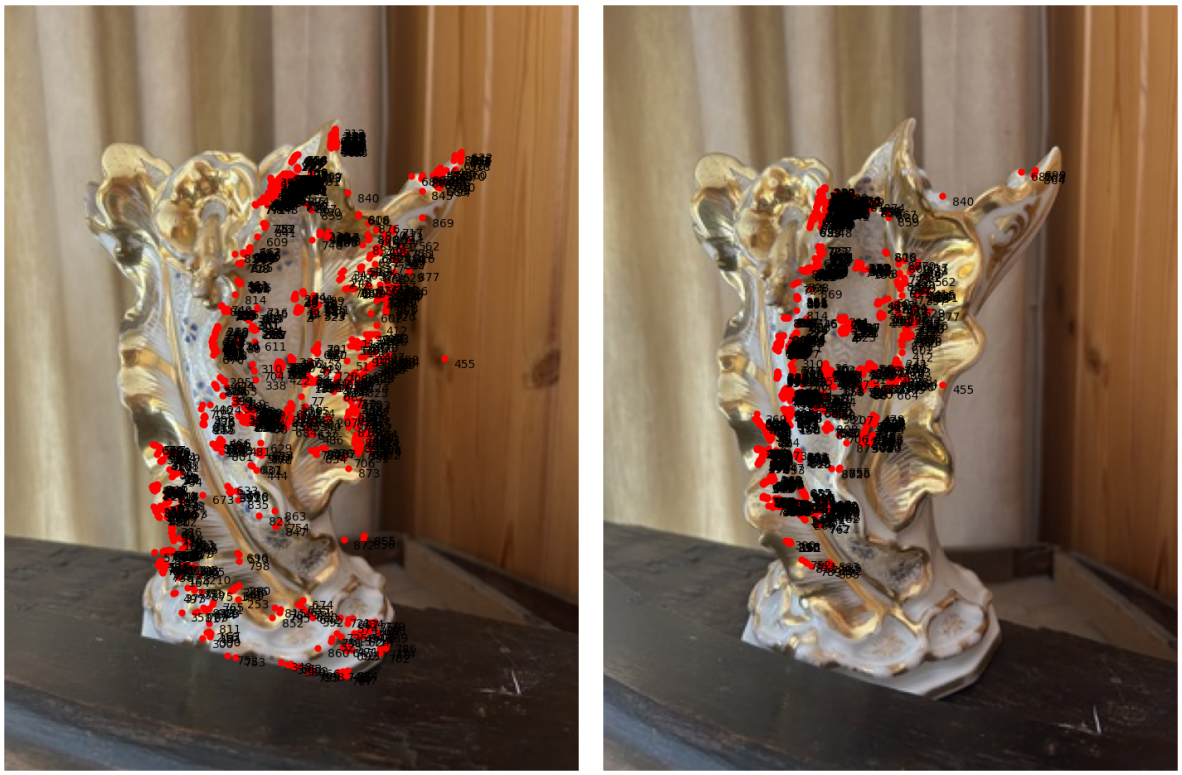


Figure 11: Point matches from removing the homographies on the canny point matches.

## 2.6 3D reconstructed scene:

**Since the error $x_i' F x_i$ was already reducing from 0.0753223 to 0.01464 through the use 1000 point matches, I only ran my LM algorithm on 100 random points samples from the point matches**. I then use the improved P and P' to do a final linear least-squares estimate of all the world coordinates. While it would have been possible to run LM on all 1000 points, the time constraints were not feasible in this setting as you would need to optimize over 6012 parameters for each run of the cost function.

### 2.6.1 2 views of the 3D scene:

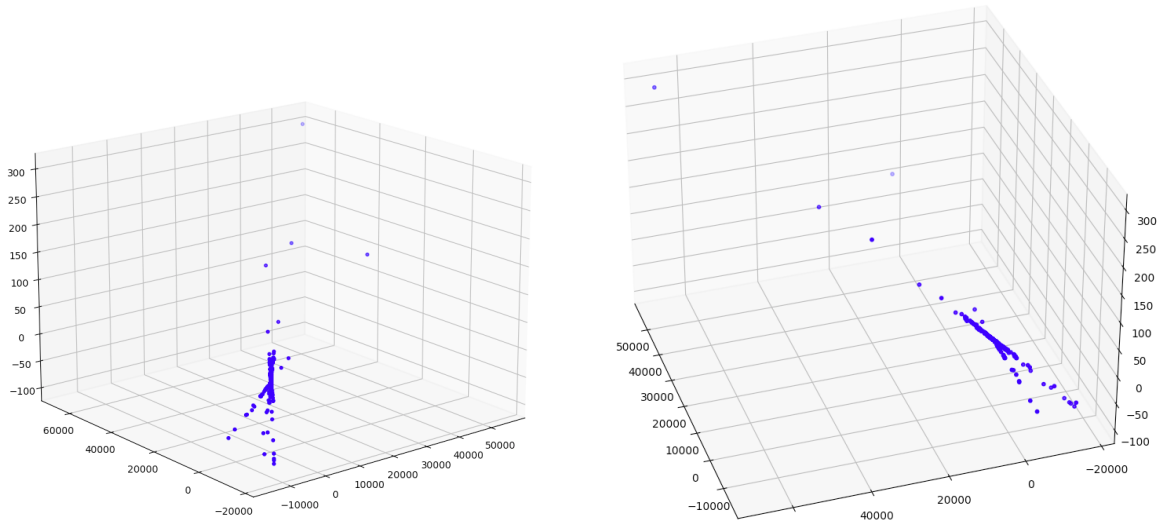



Figure 12

### 2.6.2 Both left and right image points with their corresponding world points:

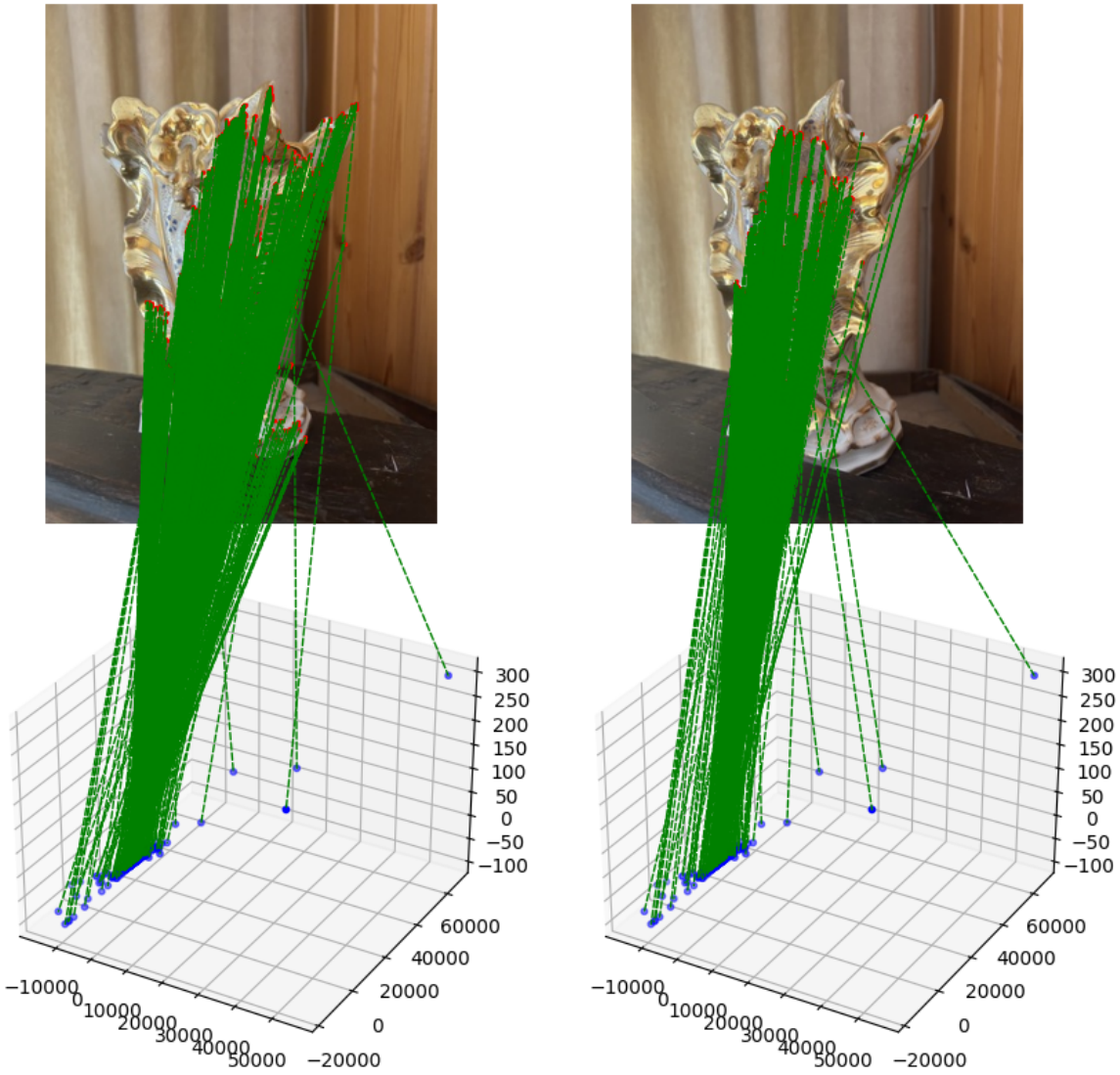
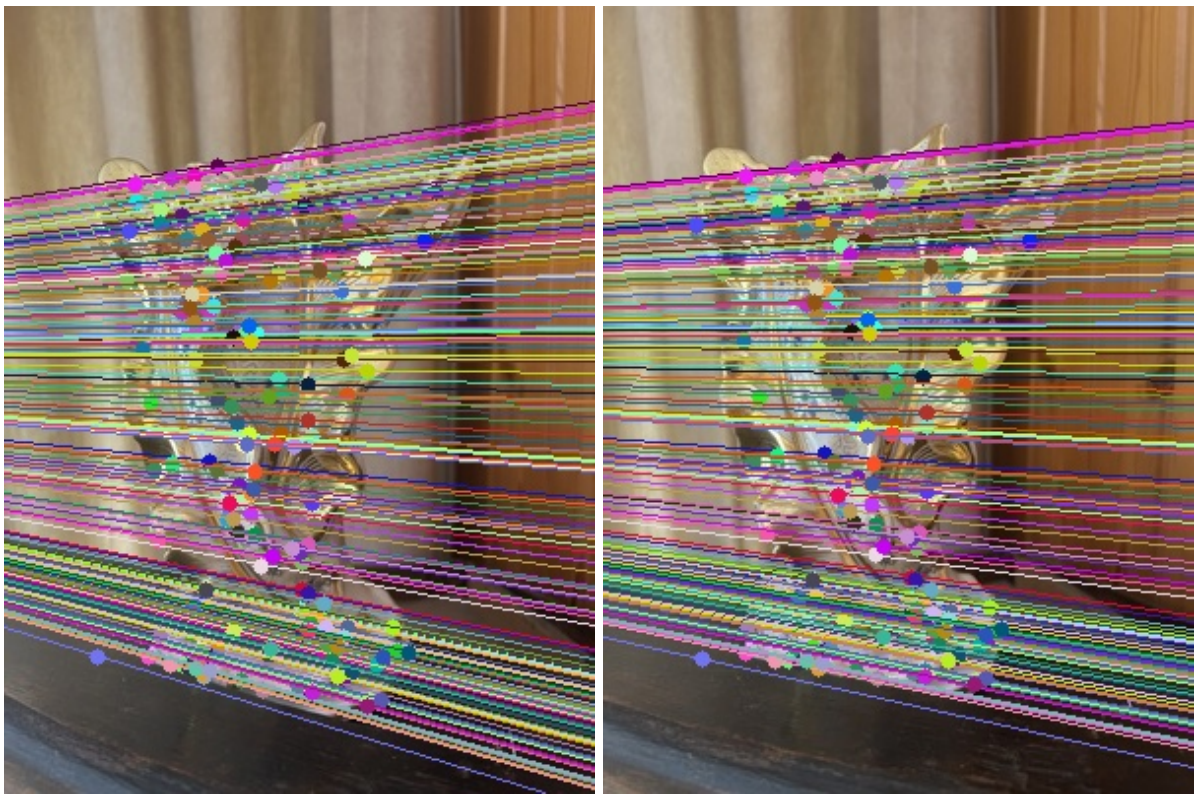



Figure 13

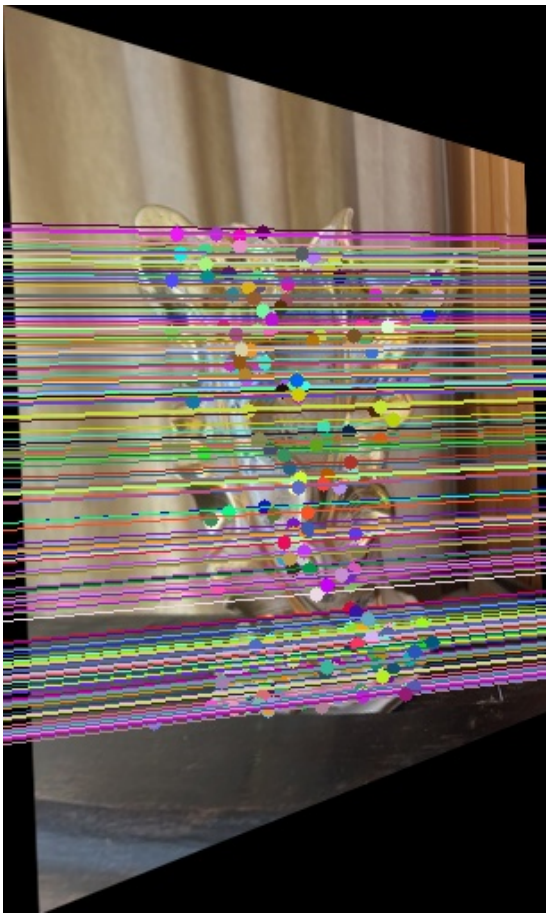
# 3 Results: Task 2

## 3.1 Before Rectification:

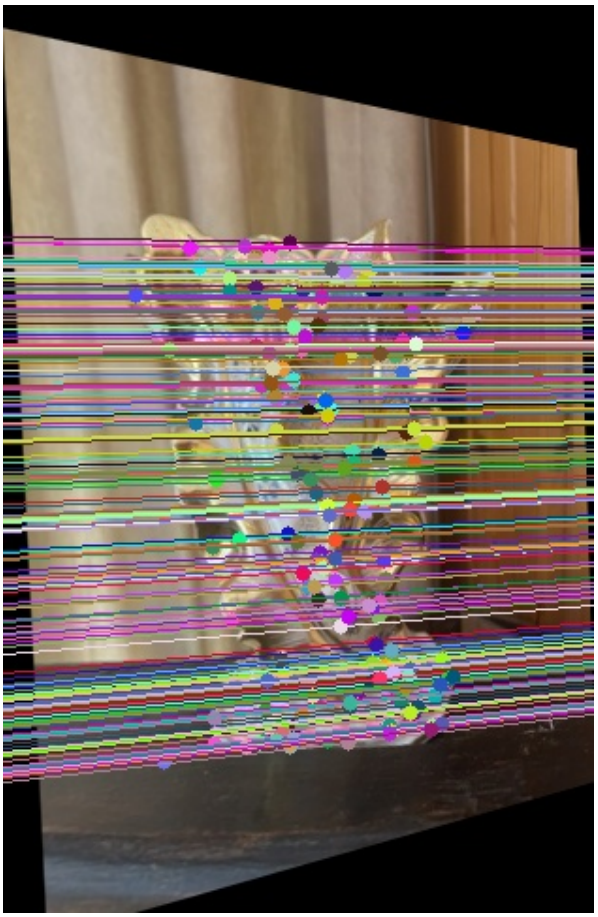


(a) Left image with point matches before rectification (b) Right image with point matches before rectification

### 3.2 After Rectification:



(a) Left image with point matches after rectification



(b) Right image with point matches after rectification

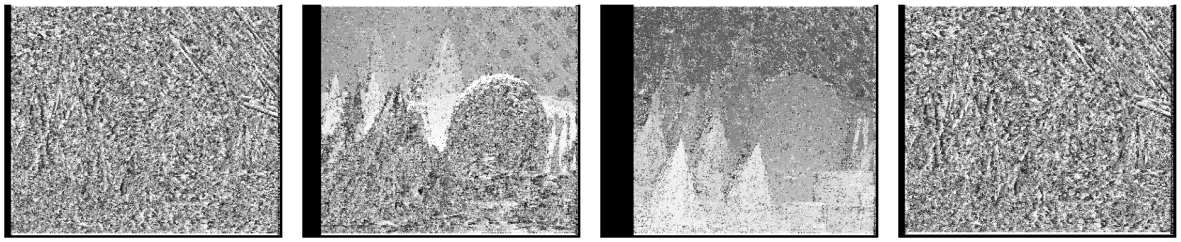
### 3.3 Commentary on difference in results from my implementation and Loop & Zhang

When the Loop and Zhang implementation worked, it found a lot more point matches and the lines were much more horizontal without requiring any pruning. On the other hand, my implementation ran into some erros when the point correspondences ended up outside of the valid pixel range which required manual pruning, and a translocation homography to move more of the image back into the frame. On the other hand, as shown in the cup image results section of task 1, Loop and Zhang did not work on some images that worked for my implementation. The error reported by the provided code pointed to a too larger distance between the left and right images which caused floating point errors in their equations. Therefore, the ideal option for 3D reconstruction would be lots of images with small angle differences, and rectification done using Loop and Zhang's algorithm. If only a few images of the scene were taken, and the angle between them was therefore larger I would recommend trying our implementation with hand picked correspondences for rectifiation.

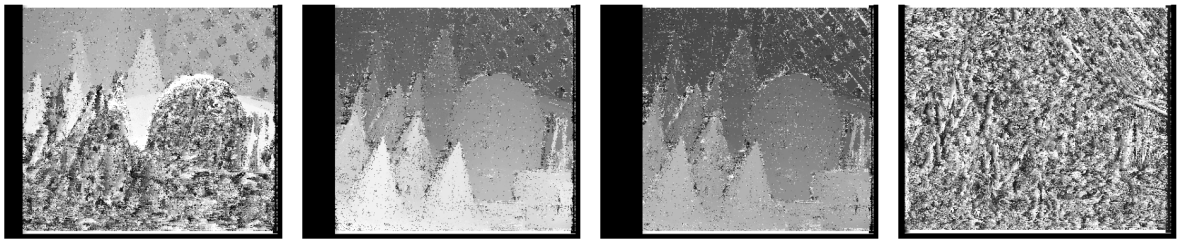
## 4 Results: Task 3

For this section, I report results from a range of window sizes and $d_max$ values. Note that the maximum value found in the ground truth disparity map was 54; however, I wanted to test different ranges to see how this would affect the results.

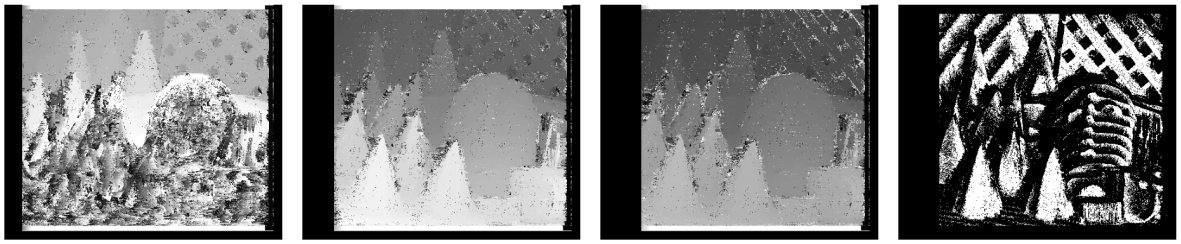
## 4.1 Estimated disparity maps for different window sizes:



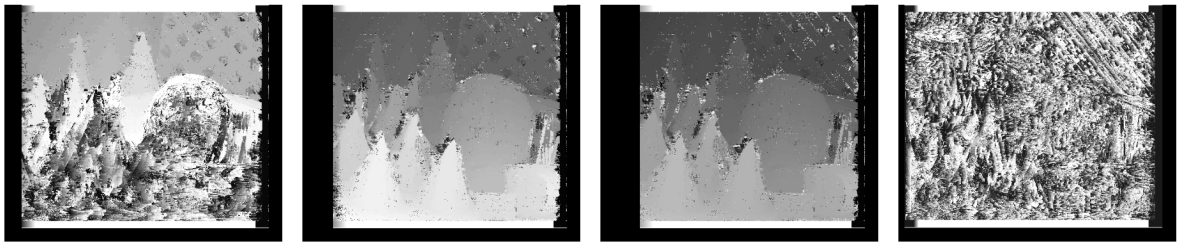
(a) Disparity maps with a window size of **5** and a dmax of [10, 30, 54, 70]



(b) Disparity maps with a window size of **9** and a dmax of [10, 30, 54, 70]

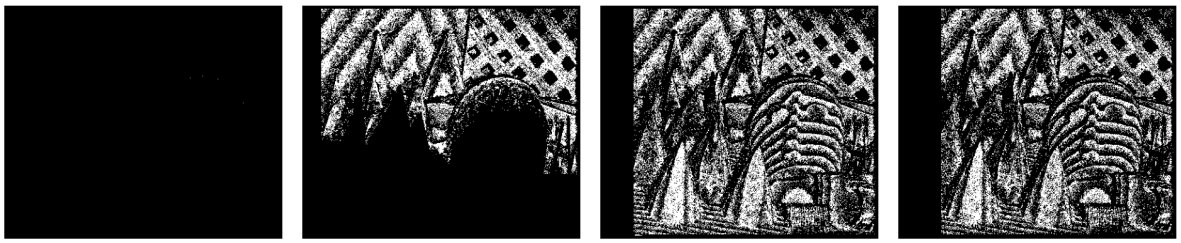


(c) Disparity maps with a window size of **16** and a dmax of [10, 30, 54, 70]

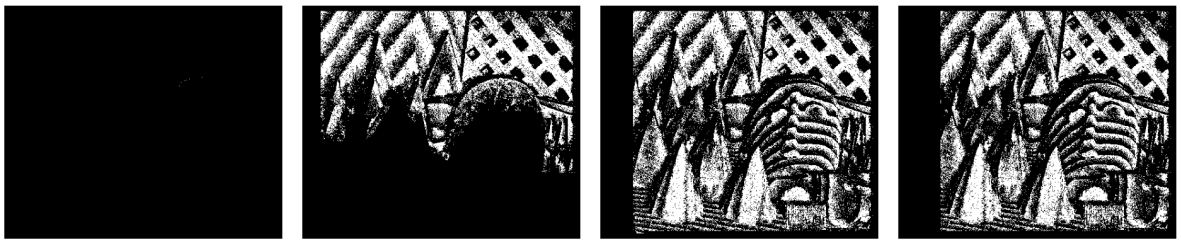


(d) Disparity maps with a window size of **25** and a dmax of [10, 30, 54, 70]

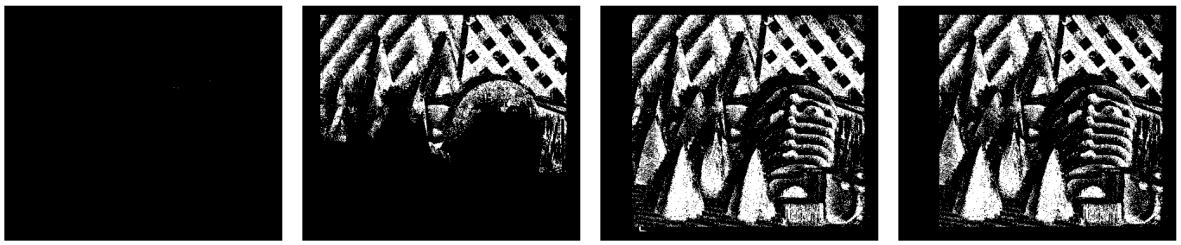
## 4.2 Accuracy and error masks:



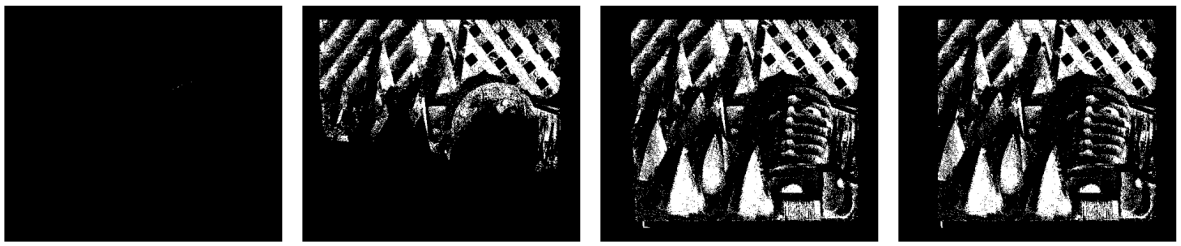
(a) Accuracy maps with a window size of **5** and a dmax of [10, 30, 54, 70]



(b) Accuracy maps with a window size of **9** and a dmax of [10, 30, 54, 70]



(c) Accuracy maps with a window size of **16** and a dmax of [10, 30, 54, 70]



(d) Accuracy maps with a window size of **25** and a dmax of [10, 30, 54, 70]

## 4.3 Observations on the output quality based on window sizes:

Included below are histograms of the impact on changing the window size and $d_max$ values on the accuracy of the disparity map. As you can see, the best results were found with a $d_max$ that matched the maximum in the ground truth since it creates an upper bound for the disparity error per point. Additionally, we can see that the **accuracy is highest around a window size of 9 for all values of** $d_max$. Lastly, as the $d_max$ value went down, the accuracy also went down sharply (much larger effect than increasing $d_max$).

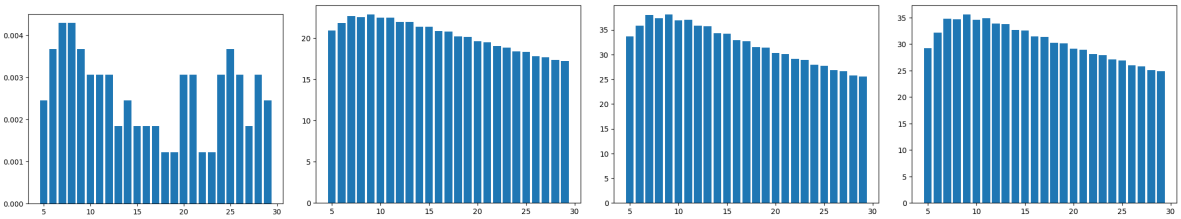


Figure 18: Histogram of the accuracy for using d max values of [10, 30, 54, 70] (top to bottom)

# 5 Results: Task 4

For plotting these graphs, I include 100 probe points for the point matching process, and include 2500 for the 3D reconstruction so that the object can better be seen. Some images show only a few matching or no matching points since the estimated and real depths of the detected point matches were too different. This demonstrates the power of using depth checking for correspondence points to remove noise in the output of an image.

## 5.1 Image and depth map for each image pair

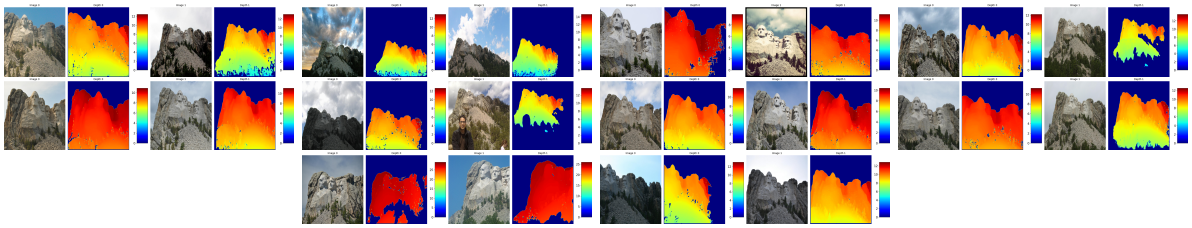


Figure 19: Depth checked images with corresponding matching points.

## 5.2 Depth check process for each image pair

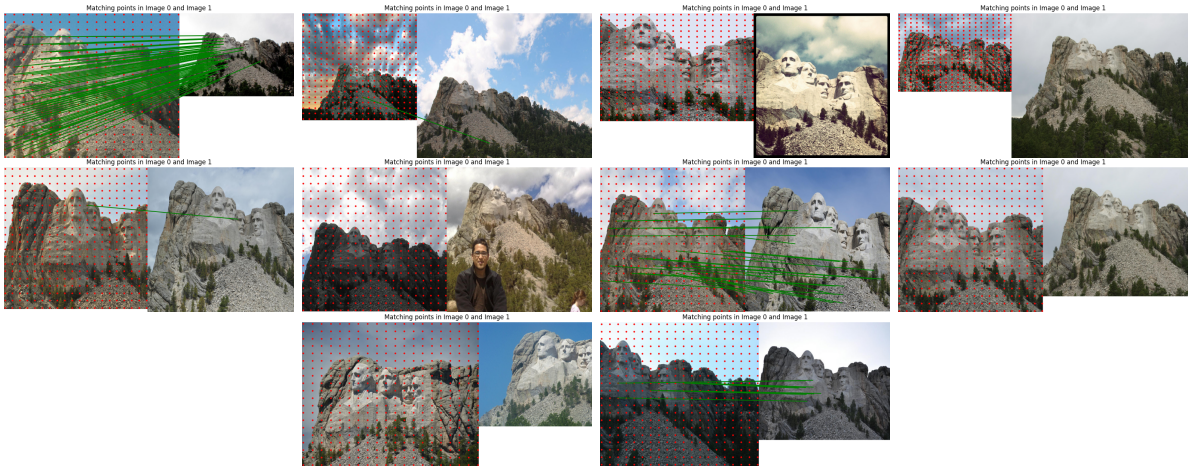



Figure 20: Depth checked images with corresponding matching points.

### 5.3 3D world coordinates for each pair

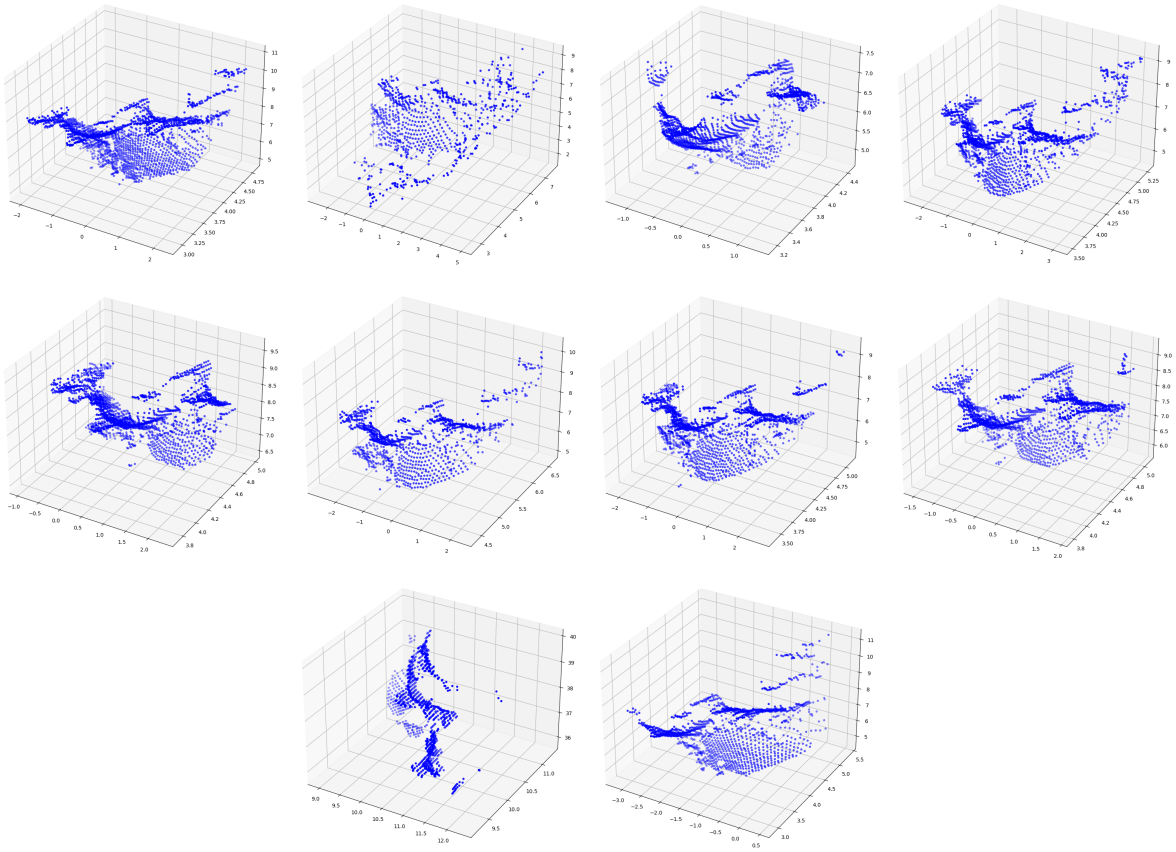


Figure 21: 3D point reconstructions for all image pairs.

## 6 Code Printout:

```
# %%
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares
import pickle
import random
from tqdm import tqdm

# %%
img_dir = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW9/Images/"

# %%
class Point():
    def __init__(self, x, y):
        """Defines a point using its physical space coordinates"""
        self.x = x
        self.y = y
        self.hc = self.get_hc()
    @classmethod
    def from_hc(cls, hc):
        """Defines a point from its representation in homogeneous coordinates"""
        if np.isclose(hc[2],0):
            x = hc[0]
            y = hc[1]
        else:
            x = hc[0] / hc[2]
            y = hc[1] / hc[2]
        return cls(x, y)
```

```
    def get_hc(self):
        """Returns the point in homogeneous coordinates"""
        return np.array([self.x, self.y, 1])
    def apply_homography(self, H):
        self.hc = H @ self.hc
        return self.hc
    def apply_x_y_offset(self, x_offset, y_offset):
        self.__init__(self.x + x_offset, self.y + y_offset)
        return self.hc
    def __repr__(self):
        """To string method for debugging"""
        return f"Point(x={self.x}, y={self.y}, hc={self.hc})"

# %%
def open_image_in_grayscale(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    return img

# %%
# For view 3 and 4:
# Left Image:
# img1_coords = [(135.39086538156292, 142.9686706914469), (153.67407839960097,
    150.35743901737024), (97.53385960303707, 153.29330863104485), (150.01673332962673,
    242.3620485533101), (154.906077176412, 213.23856563984987), (166.59798637524642,
    217.06500865037748), (168.20133062334986, 236.29312439383074), (174.10846178282134,
    228.46617560753106), (158.30688593123517, 251.50398712946978), (140.20318683016325,
    286.359314173019), (137.86601754532887, 291.3675340690927), (123.99692097166292,
    154.54893583465073), (117.26992782683912, 120.09360509287032), (141.2245863425531,
    121.40618912112862), (110.44752003348668, 122.43507091165702), (144.4775692921619,
    133.73249617162207), (156.46386194602724, 141.5855844620856), (110.71053591493266,
    256.13154910371065), (125.36770173155205, 261.28376496652237), (136.11628999707295,
    264.9258485936823)]

# # Right Image:
# img2_coords = [(103.94194116801071, 146.10462964705587), (125.13056380524532,
    155.512106943920315), (76.58935558176239, 150.3423541745028), (131.6160480507922,
    243.6500281999699), (134.20041793306098, 216.25570744792097), (151.7741331324886,
    223.31965179278893), (158.1723424180141, 242.3256019935444), (170.37685143709615,
    236.66553983976723), (143.2063567947179, 253.9351568492766), (125.23883285132553,
    280.4958444177697), (123.20772144902898, 284.4018278837246), (95.91480113148509,
    155.57379406148593), (86.17062836890693, 121.6411453823902), (107.37853379334175,
    125.30953983418433), (81.40673972671068, 123.36838476930289), (112.33528369563939,
    137.254669816577), (126.6729958282682, 146.34430177937818), (97.56755810749733,
    244.93423422613654), (108.28612115429344, 253.4884720423296), (118.69549488243196,
    259.4661322030428)]

# img1_points = [Point(x, y) for (x, y) in img1_coords]
# img2_points = [Point(x, y) for (x, y) in img2_coords]

# %%
# Left image:
img1_coords = [(95.36836337466062, 168.76952141057927), (133.9695606660342,
    162.226945598482O8), (102.56519676796754, 202.13665805227512), (147.05471229022865,
    189.0515064280807), (210.41955903039025, 120.35446040105984), (213.69084693643885,
    120.35446040105984), (162.0044980208708, 191.66853675291958), (165.2757859269194,
    162.8812031796918), (164.8668749386633, 69.32236906670147), (237.48946645294242,
    79.79049036605704), (168.79242042592162, 316.6317347639765), (155.05301122051745,
    308.7806437894599), (71.39112403685812, 80.72231890381454), (104.32980104565014,
    112.07505961218324), (111.64950704760392, 108.04922131110865), (112.37088226634825,
    230.9239916255029), (124.80177630933298, 259.05706761752094), (100.56290877527834,
    314.84556210945397), (85.67466735946616, 303.6793810475948), (78.58502859003177,
    311.3007427247368), (184.86684848498908, 243.86891309260727), (185.6544659729186,
    232.5469117036203), (159.46618449926174, 265.7252983826517)]

# Right iamge
img2_coords = [(89.12020347410777, 171.38655173541815), (124.45011285943275,
    165.49823350453067), (95.66277928620497, 205.40794595832375), (138.84377964604664,
    193.63130949654874), (202.83017108835747, 121.66297556347928), (208.0642317380353,
    119.70020281985012), (153.76085249762832, 194.93982465896818), (157.68639798488664,
    167.46100624815983), (151.16835683208478, 73.24791455395984), (236.22184238934864,
    75.210687297589), (163.59925087506952, 324.4828257384932), (150.51409925087506,
    315.32321960155707), (66.09429952536783, 85.5322304671512), (93.9072102215332,
    115.42739116195997), (100.89762093126461, 112.15528402123464), (103.59034888191881,
```

```
    233.7957134749712), (115.61328702986482, 262.897965627476), (97.57875165958416,
    317.83723897614044), (87.65648352360236, 303.7371737302716), (79.56200162319618,
    312.7455487484656), (178.4377679783561, 252.25864453774142), (179.23394652593706,
    239.5197877764464), (151.6662643159471, 271.6654966350267)]

# img2_coords = [(95, 168), (290, 390)]

img1_points = [Point(x, y) for (x, y) in img1_coords]
img2_points = [Point(x, y) for (x, y) in img2_coords]

# %%
plt.figure(figsize=(16, 12))
plt.imshow(cv2.cvtColor(cv2.imread(img_dir+"view1.jpg"), cv2.COLOR_BGR2RGB))
plt.axis("off")
for i, (x, y) in enumerate(img1_coords):
    plt.plot(x, y, 'ro', markersize=5)  # Plot red circle
    plt.text(x + 5, y + 5, str(i + 1), color="black", fontsize=10)

plt.show()

# %%
class Image_Rectifier():
    def __init__(self, points1, points2, img1_path, img2_path, img1=None, img2=None):
        self.F = None
        self.points1 = np.array([point.hc for point in points1])
        self.points2 = np.array([point.hc for point in points2])
        if img1_path != None:
            self.img1 = open_image_in_grayscale(img1_path)
            self.img2 = open_image_in_grayscale(img2_path)
        else:
            self.img1 = img1
            self.img2 = img2

    def calculate_fundamental_matrix(self):
        points_mat = []
        for (point1, point2) in zip(self.points1, self.points2):
            x, y, _ = point1 # x (left image)
            xp, yp, _ = point2 # x' (right image)

            point_mat_entry = np.array([x*xp, xp*y, xp, x*yp, y*yp, yp, x, y, 1])
            points_mat.append(point_mat_entry)

        points_mat = np.vstack(points_mat)

        # Estimate fundamental matrix as the Null space
        _, _, Vt = np.linalg.svd(points_mat)
        f = Vt[-1]

        F = f.reshape((3,3))
        F = F / F[-1, -1]

        # Ensure rank 2:
        U, D, Vt = np.linalg.svd(F)
        D_rank2 = np.array([[D[0], 0, 0],
                            [0, D[1], 0],
                            [0, 0, 0]])
        self.F = U @ D_rank2 @ Vt
        self.F = self.F / self.F[-1, -1]

    @staticmethod
    def get_cross_form(ep_vec):
        ep1, ep2, ep3 = ep_vec
        ep_cross = np.array([[0, -ep3, ep2],
                             [ep3, 0, -ep1],
                             [-ep2, ep1, 0]])
        return ep_cross

    def get_cononical_form(self):
        # Left projection matrix in canonical form:
        self.P = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0],
                           [0, 0, 1, 0]])
        self.e = np.array([0, 0, 0]) # Unsure if this needs to be in 3D coords or HC
```

```
        # Now to calculate the right epipole and right projection matrix
        # For ep, we know that: F^T ep = 0, so we can get the Null vector to solve this:
        _, _, Vt = np.linalg.svd(self.F.T)
        self.ep = Vt[-1]
        # Normalize e':
        self.ep = self.ep / self.ep[-1]

        # Make it into cross product form:
        ep_cross = Image_Rectifier.get_cross_form(self.ep)

        self.Pp = np.hstack((ep_cross @ self.F, self.ep.reshape(-1, 1)))
        self.Pp = self.Pp / self.Pp[-1, -1]

    @staticmethod
    def cost_func(learnable_params, x_points, xp_points, triangulation=False):
        # Unpack the learnable parameters:
        P = np.array([[1, 0, 0, 0],
                      [0, 1, 0, 0],
                      [0, 0, 1, 0]])
        Pp = learnable_params[:12].reshape((3,4))

        # Every three points past is a 3 element vector for the world points;
        world_points = np.array(learnable_params[12:]).reshape(-1, 3)
        # Add w parameter back into world_points:
        world_points = np.hstack((world_points, np.ones((world_points.shape[0], 1))))

        # Backproject the world points in image hc coordinates
        x_hat_h = (P @ world_points.T).T
        xp_hat_h = (Pp @ world_points.T).T

        # Get the normalized homogenous coordinates
        x_hat = x_hat_h / (x_hat_h[:, 2, np.newaxis] + 1e-8)
        xp_hat = xp_hat_h / (xp_hat_h[:, 2, np.newaxis] + 1e-8)

        # Only compare real pixel coordinates, not homogenous coords since we want it to
     be invariant to a scalar factor of k
        diff_x  = x_points[:, :2]  - x_hat[:, :2]
        diff_xp = xp_points[:, :2] - xp_hat[:, :2]

        residuals = np.hstack((diff_x[:, 0], diff_x[:, 1], diff_xp[:, 0], diff_xp[:, 1])
    ).flatten()
        return residuals

    @staticmethod
    # As we refine pp, we need to re-estimate the world coordinates:
    def estimate_world_coordinates(P, Pp, points1_hc, points2_hc):
        # Create the matrix to find the world points. A_mat @ X = 0:
        world_points = []
        for i, (point1, point2) in enumerate(zip(points1_hc, points2_hc)):
            x1, y1, _ = point1
            xp, yp, _ = point2

            P1, P2, P3 = P[0, :], P[1, :], P[2, :]
            Pp1, Pp2, Pp3 = Pp[0, :], Pp[1, :], Pp[2, :]

            A_mat = np.array([x1*P3  - P1,
                              y1*P3  - P2,
                              xp*Pp3 - Pp1,
                              yp*Pp3 - Pp2])
            _, _, Vt = np.linalg.svd(A_mat)

            # Get the homogenous coordinates:
            # Each point is 1 column, each point has 4 coords: x, y, z, w
            X_h = Vt[-1]

            # Convert to inhomoegenous world points:
            # Each point is 1 column, each point has 4 coords: x, y, z, 1
            X = X_h[:-1] / X_h[-1]
            world_points.append(X)
        world_points = np.array(world_points)
        return world_points

    def refine_proj_prime_matrix_with_nonlinear_leastsquares(self):
```

```
        learnable_params = self.Pp.flatten()
        idx = np.random.choice(self.points1.shape[0], size=75, replace=False)
        points1 = self.points1[idx]
        points2 = self.points2[idx]
        print(points1.shape, points2.shape)
        world_points = Image_Rectifier.estimate_world_coordinates(self.P, self.Pp,
    points1, points2)
        learnable_params = np.hstack((learnable_params, world_points.flatten()))

        # Apply non-linear least squares optimization to the P' matrix:
        optim_1 = least_squares(Image_Rectifier.cost_func, learnable_params, args=(
    points1,points2, False), method='lm', verbose=2)

        learnable_params = optim_1.x
        self.Pp = learnable_params[:12].reshape((3,4))
        self.Pp = self.Pp / self.Pp[-1, -1]

        # # Every three points past is a 3 element vector for the world points;
        world_points = np.array(learnable_params[12:]).reshape(-1, 3)
        # Add w parameter back into world_points:
        self.world_points = np.hstack((world_points, np.ones((world_points.shape[0], 1))
    ))

    def get_refined_fund_matrix_from_Pp(self):
        # The epipole of P' is its 4th column
        self.ep = self.Pp[:, -1]
        # Normalize e':
        self.ep = self.ep / self.ep[-1]

        # We can then calculate a refined value for F:
        ep_cross = Image_Rectifier.get_cross_form(self.ep)

        # Refined version of F: F = [e']x P' P+ (P+ = pseudo-inv of P)
        self.F = ep_cross @ self.Pp @ np.linalg.pinv(self.P)
        self.F = self.F / self.F[-1, -1]

    def get_homographies_from_refined_F(self):
        h, w = self.img1.shape

        # Extract the columns of e' to calculate the angle between the components of e'
        ep1, ep2, _ = self.ep

        # Calculate the rotation matrix for the right camera:
        theta = np.arctan2(-(ep2 - h/2), -(ep1 - w/2))
        self.R = np.array([[np.cos(theta), -np.sin(theta), 0],
                           [np.sin(theta), np.cos(theta), 0],
                           [0, 0, 1]])

        # After applying the rotation matrix, the epipole should align with the x_axis
        # Ideally, we would get: [f, 0, 1] so: f = er1 / er3 (normalize by third coord)
    and er2=0 ideally
        ep_rot = self.R @ self.ep
        print(ep_rot)
        focal_len =ep_rot[0] / ep_rot[2]

        # RIGHT HOMOGRAPHY ###################################################
        # We can now calculate the right homography: H' = G R T
        # Calculate G: the geometric transformation based on the focal length
        G = np.array([[1, 0, 0],
                      [0, 1, 0],
                      [-1/focal_len, 0, 1]])
        h, w = self.img1.shape
        # Calculate T: the translation matrix to shift the center of the image
        T = np.array([[1, 0, -w/2],
                      [0, 1, -h/2],
                      [0, 0, 1]])
        T2 = np.array([[1, 0, w/2],
                      [0, 1, h/2],
                      [0, 0, 1]])
        self.Hp = T2 @ G @ self.R @ T
        self.Hp = self.Hp / self.Hp[-1, -1]

        # LEFT HOMOGRAPHY  ###################################################
```

```
        M = self.Pp @ np.linalg.pinv(self.P)
        M = M / M[-1, -1]
        H_0 = self.Hp @ M
        H_0 = H_0 / H_0[-1, -1]

        # We then compute z_i, z_i'
        z, zp = [], []
        for x, xp in zip(self.points1, self.points2):
            z.append(H_0 @ x)
            zp.append(self.Hp @ xp)
        z = np.array(z)
        zp = np.array(zp)
        # Normalize z, zp coordinates:
        z = z / z[:, -1][:, np.newaxis]
        zp = zp / zp[:, -1][:, np.newaxis]

        # Using these values, we can build a linear system to solve for the left homography:
        # A * x = b
        x = np.linalg.pinv(z) @ zp[:, 0]
        a, b, c = x

        H_A = np.array([[a, b, c],
                        [0, 1, 0],
                        [0, 0, 1]])
        # Calculate the value of H and normalize it by its last value
        self.H = H_A @ H_0
        self.H = self.H / self.H[-1, -1]


# %% [markdown]
# ### Initial estimation of the fundamental matrix, P and P' in canonical form

# %%
# ##########################################################################################################

# ############################## 3.1: Image Rectification ##############################################
img_rectifier = Image_Rectifier(img1_points, img2_points, img_dir+"view1.jpg", img_dir+"view2.jpg")
# Get the Fundamentaal Matrix
img_rectifier.calculate_fundamental_matrix()
print(img_rectifier.F)

# %%
# You can check if your fundamental is correct by checking x'Fx = 0?
x = img_rectifier.points1
xp = img_rectifier.points2
F = img_rectifier.F

errors = []
for x_point, xp_point in zip(x, xp):
    errors.append(np.abs(xp_point @ F @ x_point))
print("Average error: ", np.mean(errors))

# %% [markdown]
# ### Perform non-linear least squares to refine P':

# %%
# Calculate the canonical form of the matrix:
img_rectifier.get_cononical_form()

img_rectifier.refine_proj_prime_matrix_with_nonlinear_leastsquares()
img_rectifier.get_refined_fund_matrix_from_Pp()


# %%
# You can check if your fundamental is correct by checking x'Fx = 0?
x = img_rectifier.points1
xp = img_rectifier.points2
F = img_rectifier.F

```

```
errors = []
for x_point, xp_point in zip(x, xp):
    errors.append(np.abs(xp_point @ F @ x_point))

print("Average error: ", np.mean(errors))

# %%
# After finishing LM estimation of F:
img_rectifier.get_homographies_from_refined_F()
H, Hp = img_rectifier.H, img_rectifier.Hp

# %%
H_translation = np.array([[1, 0, 150],
                          [0, 1, 150],
                          [0, 0, 1]])
# H_translation = np.array([[1, 0, 0],
#                           [0, 1, 0],
#                           [0, 0, 1]])

# %%
img1 = cv2.imread(img_dir+"view1.jpg")
img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
warped_img1 = cv2.warpPerspective(img1, H_translation@H, (400, 600))
plt.imshow(warped_img1)
plt.axis("off")
plt.show()

# %%
img2 = cv2.imread(img_dir+"view2.jpg")
img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)
warped_img2 = cv2.warpPerspective(img2, H_translation@Hp, (400, 600))
plt.imshow(warped_img2)
plt.axis("off")
plt.show()

# %%
# Print Point Correspondences on rectified images:
# h, w, _ = warped_img1.shape
img1_pointshc = [H @ point.hc for point in img1_points]
img2_pointshc = [Hp @ point.hc for point in img2_points]

# Get the combined warped image:
combined_image = np.hstack((warped_img1, warped_img2))

for i, (point1, point2) in enumerate(zip(img1_pointshc, img2_pointshc)):
    if i>-1: #i != 12 and i != 15 and i != 16 and i != 22:
        x1, y1 = (point1[:2] / point1[2]).astype(np.uint8)
        x2, y2 = (point2[:2] / point2[2]).astype(np.uint8)
        # x1, y1 = x1+150, y1+150
        # y2+= 150
        # x2 += 150
        x2 = x2 + warped_img1.shape[1]

        cv2.circle(combined_image, (x1, y1), radius=5, color=(0, 0, 255), thickness=-1)
        cv2.circle(combined_image, (x2, y2), radius=5, color=(0, 0, 255), thickness=-1)

        line_color = (random.randint(0, 255), random.randint(0, 255), random.randint(0,
    255))
        cv2.line(combined_image, (x1, y1), (x2, y2), color=line_color, thickness=2)

combined_image = cv2.cvtColor(combined_image, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(10, 6))
plt.imshow(combined_image)
plt.axis("off")
plt.show()

# %% [markdown]
# ### 3.2 Interest Point Detection

# %%
#
    ##########################################################################################################
```

```
# ################################ 3.2: Interest Point Detection ###################################
# Open the images:
img1_bw = np.expand_dims(open_image_in_grayscale(img_dir+"view1.jpg"), axis=-1)
img2_bw = np.expand_dims(open_image_in_grayscale(img_dir+"view2.jpg"), axis=-1)
warped_img1_bw = cv2.warpPerspective(img1_bw, H, (400, 600))
warped_img2_bw = cv2.warpPerspective(img2_bw, Hp, (400, 600))

# Run canny operator for corner detection on the image
edges = cv2.Canny(image=warped_img1_bw, threshold1=150, threshold2=200, apertureSize=3)
edges2 = cv2.Canny(image=warped_img2_bw, threshold1=150, threshold2=200, apertureSize=3)
combined_image = np.hstack((edges, edges2))
plt.imshow(combined_image)
plt.axis("off")
plt.show()

# %%
combined_image = np.hstack((edges, edges2))
# Use Hough Line Transform to detect straight lines
lines = cv2.HoughLinesP(combined_image, 1, np.pi/180, threshold=5, minLineLength=40, maxLineGap=10)

# Create a mask to draw the detected lines
line_mask = np.zeros_like(combined_image)

# Draw the detected lines on the mask
if lines is not None:
    for line in lines:
        x1, y1, x2, y2 = line[0]
        cv2.line(line_mask, (x1, y1), (x2, y2), 255, thickness=2)

# Inpaint the image to remove the lines
img_wlines_removed = cv2.inpaint(combined_image, line_mask, inpaintRadius=3, flags=cv2.INPAINT_TELEA)

# Show the original, mask, and result images
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(combined_image, cmap="gray")
plt.axis("off")

plt.subplot(1, 3, 2)
plt.title("Detected Lines through HoughLinesP")
plt.imshow(line_mask, cmap="gray")
plt.axis("off")

plt.subplot(1, 3, 3)
plt.title("Result with Lines Removed")
plt.imshow(img_wlines_removed, cmap="gray")
plt.axis("off")

plt.tight_layout()
plt.show()

# %%
edge_to_remove = edges.copy()
# Use Hough Line Transform to detect straight lines
lines = cv2.HoughLinesP(edge_to_remove, 1, np.pi/180, threshold=5, minLineLength=40, maxLineGap=10)

# Create a mask to draw the detected lines
line_mask = np.zeros_like(edge_to_remove)

# Draw the detected lines on the mask
if lines is not None:
    for line in lines:
        x1, y1, x2, y2 = line[0]
        cv2.line(line_mask, (x1, y1), (x2, y2), 255, thickness=2)

# Inpaint the image to remove the lines
left_edge = cv2.inpaint(edge_to_remove, line_mask, inpaintRadius=3, flags=cv2.INPAINT_TELEA)
```

```
480
481
482 # %%
483 edge_to_remove = edges2.copy()
484 # Use Hough Line Transform to detect straight lines
485 lines = cv2.HoughLinesP(edge_to_remove, 1, np.pi/180, threshold=5, minLineLength=40,
        maxLineGap=10)
486
487 # Create a mask to draw the detected lines
488 line_mask = np.zeros_like(edge_to_remove)
489
490 # Draw the detected lines on the mask
491 if lines is not None:
492     for line in lines:
493         x1, y1, x2, y2 = line[0]
494         cv2.line(line_mask, (x1, y1), (x2, y2), 255, thickness=2)
495
496 # Inpaint the image to remove the lines
497 right_edge = cv2.inpaint(edge_to_remove, line_mask, inpaintRadius=3, flags=cv2.
        INPAINT_TELEA)
498
499 # %%
500 # Extract edge points from the canny edge map:
501 kp1 = np.column_stack(np.where(left_edge > 254))
502 kp2 = np.column_stack(np.where(right_edge > 254))
503
504 # %%
505 # I now need to run NCC on the points in the canny edge detector:
506 # The follwoing code is from my solution for hw4:
507 def get_masked_corners(corners, area_bound, img):
508     return corners[(corners[:, 0] >= area_bound) & (corners[:, 0] <= img.shape[0] -
        area_bound) &
509                                (corners[:, 1] >= area_bound) & (corners[:, 1] <= img.
        shape[1] - area_bound)]
510
511 def SSD(m, img1, corners1, img2, corners2, num_matches):
512     area_bound = m+1
513     # Remove boundary points that are outside of the neighborhood search area
514     # Make sure to check for the boundaries on all sides of the images
515     # Corners is a list of points: [(x1,y1), (x2,y2), (x3,y3), ...]
516     masked_corners1 = get_masked_corners(corners1, area_bound, img1)
517     masked_corners2 = get_masked_corners(corners2, area_bound, img2)
518
519     matching_points = []
520     for coord1 in masked_corners1:
521         # Points are (y,x)
522         # Get (m+1) by (m+1) area in the BW image around that corner point
523         neighborhood1 = img1[coord1[0]-(area_bound):coord1[0]+(area_bound), coord1[1]-(
        area_bound):coord1[1]+(area_bound)]
524         for coord2 in masked_corners2:
525             # Points are meant to be on a horizontal line
526             if np.abs(coord2[0] - coord1[0]) < 4 and np.abs(coord2[1] - coord1[1] < 10):
527                 neighborhood2 = img2[coord2[0]-(area_bound):coord2[0]+(area_bound),
        coord2[1]-(area_bound):coord2[1]+(area_bound)]
528
529                 # Calculate the sum of squared distances and add the result to a list to
         later get the topk
530                 ssd = np.sum((neighborhood1 - neighborhood2)**2)
531                 matching_points.append([ssd, [coord1, coord2]])
532
533     # Get the top num_matches corner matches (with the highest SSD values first)#
        Convert to array for easier manipulation
534     matching_points = np.array(matching_points, dtype=object)
535     print("Number of matches found: ", matching_points.shape)
536
537     if num_matches > len(matching_points):
538         num_matches = len(matching_points)
539     # Sort matching points by SSD (smallest first)
540     sorted_matches = sorted(matching_points, key=lambda x: x[0])
541     # To store the top correspondences without repeats
542     topk_matches = []
543     used_coords1 = set()
544     used_coords2 = set()
```

```
545
546     # Iterate over sorted matches and select the top num_matches with unique points
547     for match in sorted_matches:
548         coord1, coord2 = match[1]
549         # Ensure no coordinate from coord1 or coord2 is repeated
550         if tuple(coord1) not in used_coords1 and tuple(coord2) not in used_coords2:
551             topk_matches.append([coord1, coord2])
552             used_coords1.add(tuple(coord1))
553             used_coords2.add(tuple(coord2))
554
555         # # Stop when we have enough matches
556         if len(topk_matches) == num_matches:
557             break
558     return topk_matches
559
560 # %%
561 kp_matches = SSD(20, warped_img1, kp1, warped_img2, kp2, num_matches=1000)
562
563 # %%
564 kp_matches = np.array(kp_matches)
565
566 # %%
567 with open("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW9/kp_matches_no_translate.pkl", "wb")
      as file:
568     pickle.dump(kp_matches, file)
569
570 # %%
571 with open("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW9/kp_matches_no_translate.pkl", "rb")
      as file:
572     kp_matches = pickle.load(file)
573
574 # %%
575 combined_edges = np.hstack((left_edge, right_edge))
576 for kp_match in kp_matches:
577     y1, x1 = kp_match[0]
578     y2, x2 = kp_match[1]
579     x2 += right_edge.shape[1]
580     cv2.circle(combined_edges, (x1, y1), radius=5, color=(0, 0, 255), thickness=-1)
581     cv2.circle(combined_edges, (x2, y2), radius=5, color=(0, 0, 255), thickness=-1)
582
583     line_color = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)
      )
584     cv2.line(combined_edges, (x1, y1), (x2, y2), color=line_color, thickness=2)
585 plt.imshow(combined_edges, cmap="gray")
586 plt.axis("off")
587 plt.show()
588
589 # %% [markdown]
590 # ### 3D Reconstruction:
591
592 # %%
593 # Recreate the points from the rectified images onto the original image
594 # I also remove border points again which helps to remove the final errors. I do this by
      applying a mask around the border of the image
595
596 img_1 = cv2.imread(img_dir+"view1.jpg")
597 img_2 = cv2.imread(img_dir+"view2.jpg")
598
599 og_points1, og_points2 = [], []
600 for points in kp_matches:
601     y1, x1 = points[0]
602     y2, x2 = points[1]
603
604     hc1 = np.array([x1, y1, 1])
605     hc2 = np.array([x2, y2, 1])
606
607     og_coord1 = np.linalg.inv(H) @ hc1
608     og_coord2 = np.linalg.inv(Hp) @ hc2
609
610     og_point1 = Point.from_hc(og_coord1)
611     og_point2 = Point.from_hc(og_coord2)
612
613     x1, y1, _ = og_point1.hc
```

```
    x2, y2, _ = og_point2.hc

    if (y1 < img_1.shape[0] - 30 and x1 < img_1.shape[1] - 30) and (y2 < img_2.shape[0]
    - 30 and x2 < img_2.shape[1] - 30):
        og_points1.append(og_point1)
        og_points2.append(og_point2)

# Remove the homography on the points so that they apply to the original images:
# kp1_points = [Point.from_hc(Point(x, y).apply_homography(np.linalg.inv(H))) for (x, y)
     in kp1_coords]
# kp2_points = [Point.from_hc(Point(x, y).apply_homography(np.linalg.inv(Hp))) for (x, y
    ) in kp2_coords]


# %%
plt.figure(figsize=(16, 12))
img_2 = cv2.cvtColor(cv2.imread(img_dir+"view2.jpg"), cv2.COLOR_BGR2RGB)
plt.imshow(img_2)
plt.axis("off")
for i, points in enumerate(og_points2):
    x, y, _ = points.hc
    plt.plot(x, y, 'ro', markersize=5)  # Plot red circle
    plt.text(x + 5, y + 5, str(i + 1), color="black", fontsize=10)

plt.show()

# %%
# Repeat steps 1-4:
img_rectifier_final = Image_Rectifier(og_points1, og_points2, img_dir+"view1.jpg",
    img_dir+"view2.jpg")#, warped_img1, warped_img2)
# Get the Fundamentaal Matrix
img_rectifier_final.calculate_fundamental_matrix()
print(img_rectifier_final.F)

# %%
# Calculate the canonical form of the matrix:
img_rectifier_final.get_cononical_form()

img_rectifier_final.refine_proj_prime_matrix_with_nonlinear_leastsquares()
img_rectifier_final.get_refined_fund_matrix_from_Pp()

# %%
# You can check if your fundamental is correct by checking x'Fx = 0?
x = img_rectifier_final.points1
xp = img_rectifier_final.points2
F = img_rectifier_final.F

errors = []
for x_point, xp_point in zip(x, xp):
    errors.append(np.abs(xp_point @ F @ x_point))

print("Average error: ", np.mean(errors))

# %%
# Reconstruct the points:
P = img_rectifier_final.P
Pp = img_rectifier_final.Pp
x = [point.hc for point in og_points1]
xp = [point.hc for point in og_points2]
world_points = Image_Rectifier.estimate_world_coordinates(P, Pp, x, xp)

# %%
with open("./world_points.pkl", "wb") as file:
    pickle.dump(world_points, file)

# %%
from matplotlib.patches import ConnectionPatch
from mpl_toolkits.mplot3d import proj3d

# Create a figure
fig = plt.figure(figsize=(10, 8))

img_points = np.array([point.hc[:2] for point in og_points2])
```

```

# Plot 2D image with points
ax1 = fig.add_subplot(2, 1, 1)  # Top subplot
ax1.imshow(img_2)
ax1.scatter(img_points[:, 0], img_points[:, 1], c='red', s=2, label='2D Points')  # Note
    x, y swap
ax1.axis("off")


# Plot 3D points
ax2 = fig.add_subplot(2, 1, 2, projection='3d')  # Bottom subplot

# Plot 3D points in the world frame
ax2.scatter(
    world_points[:, 0],
    world_points[:, 1],
    world_points[:, 2],
    c='b', s=10, label='3D points (world frame)'
)

# Draw lines connecting points from 2D to 3D across subplots
# This part of the code was based on a Piazza post question 91 on how to plot lines
    between both figure axes.
for i in range(len(img_points)):
    # Get 2D point coordinates
    x2d, y2d = img_points[i, 0], img_points[i, 1]

    # Get 3D point in screen coordinates
    x3d, y3d, _ = proj3d.proj_transform(world_points[i, 0], world_points[i, 1],
    world_points[i, 2], ax2.get_proj())

    # Create a ConnectionPatch from the 2D point to the 3D screen coordinates
    con = ConnectionPatch(
        xyA=(x2d, y2d),
        coordsA=ax1.transData,
        xyB=(x3d, y3d),
        coordsB=ax2.transData,
        color="green", linestyle="--", linewidth=1
    )
    fig.add_artist(con)

plt.tight_layout()
plt.show()

# %%
```

```

# %%
import cv2
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import pickle

# %%
def open_image_in_grayscale(img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    return img

# %%
task3_img_dir = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW9/Task3Images/"

left_img_path = task3_img_dir+"im2.png"
right_img_path = task3_img_dir + "im6.png"

accuracy_maps = []
for M in [5, 9, 16, 25]:
    accuracy_map = []
    for d_max in [10, 30, 54, 70]:
        # Maximum disparity from original image is 54
        window_shape = (M,M)

        left_img = open_image_in_grayscale(left_img_path)
        right_img = open_image_in_grayscale(right_img_path)
```

```
28
29      # Pad the left and right images such that the windows are created for all image
    pixels
30      left_img = np.pad(left_img, M)
31      right_img = np.pad(right_img, M)
32
33      # Create windows from the images. This outputs a shape of (H, W, M, M) for an
    MxM window centered on each pixel
34      left_windows = np.lib.stride_tricks.sliding_window_view(left_img, window_shape)
35      right_windows = np.lib.stride_tricks.sliding_window_view(right_img, window_shape
    )
36
37      # Get the center pixels for each window:
38      left_center_pixels = left_windows[:, :, M//2, M//2][..., np.newaxis, np.newaxis]
39      right_center_pixels = right_windows[:, :, M//2, M//2][..., np.newaxis, np.
    newaxis]
40
41      # Create new windows with only the center pixel values:
42      left_center_windows = np.ones_like(left_windows) * left_center_pixels
43      right_center_windows = np.ones_like(right_windows) * right_center_pixels
44
45      # For each entry in both windows, check if the entry in the img window is
    strictly larger than the center pixel value
46      # If strictly greater than, put in a 1, otherwise put in a 0 (right now these
    are True and False values)
47      # I do this for entries from (x-d,y) to (x,y)
48      disparity_map = np.zeros_like(left_img)
49
50      for row_idx in range(left_windows.shape[0]):
51          for col_idx in range(left_windows.shape[1]):
52              if col_idx > d_max:
53                  left_window = left_windows[row_idx, col_idx, ...]
54                  left_center_pixel_window = left_center_pixels[row_idx, col_idx, ...]
55                  bitvec_sums = []
56                  for d in range(d_max, 0, -1):
57                      right_window = right_windows[row_idx, col_idx-d]
58                      right_center_pixel_window = right_center_pixels[row_idx, col_idx
    -d, ...]
59
60                      # Now I can compare the two windows
61                      left_bitvec_comparator = left_window > left_center_pixel_window
62                      right_bitvec_comparator = right_window >
    right_center_pixel_window
63
64                      # XOR the two comparators. This is treated as a bitwise XOR
    since the bits are represented as logical True/False values
65                      # This results in a list of windows: (M, M)
66                      xor_bitvec = np.logical_xor(left_bitvec_comparator,
    right_bitvec_comparator)
67                      # I can then get the sum of the bitvector. This is the data cost
68                      bitvec_sums.append(np.sum(xor_bitvec))
69
70                  # Calculate the disparity value:
71                  disparity = d_max - np.argmin(bitvec_sums)
72                  # Fill in that location in the disparity map:
73                  disparity_map[row_idx, col_idx] = disparity
74
75      ground_truth_dmap = open_image_in_grayscale(task3_img_dir+"disp2.png")
76      # Lower the resolution to match out images:
77      ground_truth_dmap = ground_truth_dmap.astype(np.float32) / 4
78      ground_truth_dmap = ground_truth_dmap.astype(np.uint8)
79      # Pad the ground_truth_dmap as needed:
80      ground_truth_dmap = np.pad(ground_truth_dmap, M)
81
82      # Calculate the error (H, W), we only want to compute for non-black (padded)
    pixels
83      non_black_mask = ground_truth_dmap > 0
84      diff = np.abs(ground_truth_dmap - disparity_map)
85      # We also allow for an error of a few pixels between the ground truth and the
    computed disparity map
86      accuracy_value = np.count_nonzero(diff[non_black_mask] < 3) / np.count_nonzero(
    non_black_mask)
87      accuracy_map.append(accuracy_value)
```

```
        accuracy_printout = np.zeros_like(diff)
        accuracy_printout[non_black_mask] = (diff[non_black_mask] < 3) * 255

        plt.imshow(accuracy_printout, cmap="gray")
        plt.axis("off")
        plt.show()
        plt.close()

        print(f"M: {M}, Dmax: {d_max}, accuracy: {np.round(accuracy_value * 100, 4)}")
    accuracy_maps.append(accuracy_map)

# %%
np_accuracy_maps = np.array(accuracy_maps)
with open("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW9/accuracy_maps.pkl", "wb") as file:
    pickle.dump(np_accuracy_maps, file)

# %%
plt.bar(range(5, 30), np_accuracy_maps[:, 3]*100)
plt.show()

# %%
plt.imshow(disparity_map, cmap="gray")
plt.axis("off")
plt.show()

# %%
ground_truth_dmap = open_image_in_grayscale(task3_img_dir+"disp2.png")
# Lower the resolution to match out images:
ground_truth_dmap = ground_truth_dmap.astype(np.float32) / 4
ground_truth_dmap = ground_truth_dmap.astype(np.uint8)
# Pad the ground_truth_dmap as needed:
ground_truth_dmap = np.pad(ground_truth_dmap, M)

# Calculate the error, we only want to compute for non-black (padded) pixels
non_black_mask = ground_truth_dmap > 0
diff = np.abs(ground_truth_dmap - disparity_map)
# We also allow for an error of a few pixels between the ground truth and the computed
    disparity map
accuracy_value = np.count_nonzero(diff[non_black_mask] < 3) / np.count_nonzero(
    non_black_mask)

print(np.round(accuracy_value * 100, 4))
```

```
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'])
        print("Hi, images read")
        # 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)

        print("Images are plotted")
        #(1) make meshgrid of points in image 0
        x = np.linspace(10, img0.shape[1]-10, 20) # ignore a border of 10 pxls
        y = np.linspace(10, img0.shape[0]-10, 20)
        """
        <Student code>
        # meshgrid of x and y coordinates
        xx, yy = ...
        """
```

```
        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_like(xx.flatten()))).
    transpose(1, 0)

        #(2) get depth values at points0
        """
        <Student code>
        depth_values0 = ...
        """
        x_coords = points0[:, 0].astype(int)
        y_coords = points0[:, 1].astype(int)
        depth_values0 = depth0[y_coords, x_coords]

        # remove points with depth 0 (invalid points)
        """
        <Student code>
        valid_points = ...
        # mask points0 and depth_values0
        """
        valid_mask = depth_values0 > 0
        points0 = points0[valid_mask, :].T
        depth_values0 = depth_values0[valid_mask]

        # (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)

        # Get the camera coords
        xyz_cam0 = (K0_inv @ points0).T
        xyz_cam0 = xyz_cam0 / xyz_cam0[:, 2][:, np.newaxis]
        # Get the points at depth:
        xyz_cam0 = xyz_cam0 * depth_values0[:, np.newaxis]

        # Make homogenous coordinates (and [4,N]):
        xyz_cam0_hc = np.hstack((xyz_cam0, np.ones((xyz_cam0.shape[0], 1)))).T
        # Conver to wolrd frame:
        xyz_world_hc = 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 = ...
        """
        # Cam frame 1 and camera coords:
        xyz_cam1_hc = np.linalg.inv(T1) @ xyz_world_hc
        xyz_cam1 = xyz_cam1_hc[:3, :] / xyz_cam1_hc[3, :]
```

```
        estimated_depth_values1 = xyz_cam1[2, :]

        # 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"]
        T1 = scene_info[i_pair]["T1"]
        points1 = K1 @ xyz_cam1
        #Normalize by last row:
        points1 = points1[:2, :] / points1[2, :]

        # Create mask for valid points similar to hw3:
        valid_x = (points1[0, :] >= 0) & (points1[0, :] < img1.shape[1])
        valid_y = (points1[1, :] >= 0) & (points1[1, :] < img1.shape[0])
        valid_mask = valid_x & valid_y

        # Get the depth values at the valid points
        points1 = points1[:, valid_mask]
        estimated_depth_values1 = estimated_depth_values1[valid_mask]
        true_depth_values1 = depth1[points1[1, :].astype(int), points1[0, :].astype(int)
    ]

        # (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>
        ...
        """
        fig = plt.figure(figsize=(10, 10))
        ax = fig.add_subplot(111, projection='3d')

        # Plot 3D points in the world frame
        ax.scatter(
            xyz_world_hc[0, :],
            xyz_world_hc[1, :],
            xyz_world_hc[2, :],
            c='b', s=10, label='3D points (world frame)'
        )
        # Save the plot
```

```
236         plt.savefig(f'../pics/3D_points_pair_{i_pair}.png', bbox_inches='tight',
    pad_inches=0)
237         plt.close()
```