Task 1: Projective Stereo Reconstruction

1.1 Background and Methodology

Brief Summary of the process:
The process is divided into 3 main parts:

- Image Rectification: find the homographies $\mathbf{H}$ and $\mathbf{H}'$ for the stereo images to bring the same world point to appears on the same row
    1. An initial estimate of the fundamental matrix $\mathbf{F}$ was obtained by the normalized 8-point algorithm. Further detail on this method is given below.
    2. Left and right epipoles: $\mathbf{e}$ and $\mathbf{e}'$ are estimated using the previously calculated $\mathbf{F}$
    3. Initial estimate of the projection matrices in canonical form: $\mathbf{P}$ and $\mathbf{P}'$ is estimated using the epipoles and the fundamental matrix.
    4. The fundamental matrix $\mathbf{F}$ is refined using the Levenberg-Marquardt nonlinear least squares method to reduce the geometric error d^2_{geom}.
    5. The right homography matrix $\mathbf{H}'$ is estimated using the refined epipoles.
    6. The left homography matrix $\mathbf{H}$ is estimated using the refined epipoles and the right homography matrix $\mathbf{H}'$.
    7. Apply the homographies to rectify the images
- Interest Point Detection: Get large set of correspondences between the two images.
    1. Canny edge detector is used to extract edge features in the rectified images.
    2. Find correspondences between features from each images. The best candidate for each pair of correspondences is selected by using SSD or NCC metric.
    3. Draw the correspondences and connect them with lines
- Projective Reconstruction: Back-project Canny points into their world coordinates.
    1. The correspondences are back-projected into their world coordinates and plotted on a 3D plot.
    2. The manually selected correspondences are also back-projected into their world coordinates and plotted on a 3D plot.

Normalized 8-point algorithm:
To compute the initial estimate of the fundamental matrix $\mathbf{F}$ the 8 point correspondences that were manually picked are used. The relation between the correspondences and the fundamental matrix is $\mathbf{x}_i^T \mathbf{F} \mathbf{x}_i = 0$.

First, the points are normalized by transforming the coordinates to $\hat{\mathbf{x}}_i = \mathbf{T}\mathbf{x}_i$ and $\hat{\mathbf{x}}'_i = \mathbf{T}\mathbf{x}'_i$. $\mathbf{T}$ and $\mathbf{T}'$ are computed as:

$$\mathbf{T} = \begin{bmatrix} s\mathbf{I}_2 & -s\tilde{x} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\tilde{x}$ is the mean of the given points and the scale $s = \frac{\sqrt{2}}{\frac{1}{n}\sum_{i=1}^{n} \|x_i - \tilde{x}\|}$.

Using the normalized points the fundamental matrix is computed by linear least squares method. To construct the linear homogeneous system for $\hat{\mathbf{x}}_i^T \hat{\mathbf{F}} \hat{\mathbf{x}}_i = 0$, the equations are expressed as $\mathbf{A}\vec{f} = 0$:

$$\mathbf{A}\vec{f} = \begin{bmatrix} \hat{x}'_1\hat{x}_1 & \hat{x}'_1\hat{y}_1 & \hat{x}'_1 & \hat{y}'_1\hat{x}_1 & \hat{y}'_1\hat{y}_1 & \hat{y}'_1 & \hat{x}_1 & \hat{y}_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hat{x}'_n\hat{x}_n & \hat{x}'_n\hat{y}_n & \hat{x}'_n & \hat{y}'_n\hat{x}_n & \hat{y}'_n\hat{y}_n & \hat{y}'_n & \hat{x}_n & \hat{y}_n & 1 \end{bmatrix} \vec{f} = 0$$

where $\hat{\mathbf{x}}_i = \begin{bmatrix} \hat{x} \\ \hat{y} \\ 1 \end{bmatrix}$, $\hat{\mathbf{x}}'_i = \begin{bmatrix} \hat{x}' \\ \hat{y}' \\ 1 \end{bmatrix}$ and $\vec{f} = \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{21} & f_{22} & f_{23} & f_{31} & f_{32} & f_{33} \end{bmatrix}^T$.

To make sure $\hat{\mathbf{F}}$ is of rank 2, it is conditioned by performing a SVD to obtain $\mathbf{UDV}^T$, setting the smallest singular value in D to 0.

Once the fundamental matrix $\hat{\mathbf{F}}$ has been computed, it is "de-normalized" to get $\mathbf{F}$. This is done using the relation: $\mathbf{F} = \mathbf{T}'^{\,\mathrm{T}} \hat{\mathbf{F}} \mathbf{T}$.

Epipoles e and e′:
The epipoles e and e′ are calculated using the following relation:

$$\mathbf{F}\vec{e} = 0 \qquad \vec{e}'\mathbf{F} = 0$$

This is done by calculating the right and left null vector of the fundamental matrix $\mathbf{F}$.

Projection matrices P and P′:
The projection matrices P and P′ in canonical forms are computed using the following equations:

$$\mathrm{P} = \begin{bmatrix} I_3 \mid \vec{0} \end{bmatrix}$$

$$\mathrm{P}' = \begin{bmatrix} \begin{bmatrix} \vec{e}' \end{bmatrix}_{\mathrm{x}} \mathbf{F} \mid \vec{e}' \end{bmatrix}$$

where $\vec{e}' = \begin{bmatrix} \mathrm{e}_1 \\ \mathrm{e}_2 \\ \mathrm{e}_3 \end{bmatrix}$ and $\begin{bmatrix} \vec{e}' \end{bmatrix}_{\mathrm{x}} = \begin{bmatrix} 0 & -\mathrm{e}_3 & \mathrm{e}_2 \\ \mathrm{e}_3 & 0 & -\mathrm{e}_1 \\ -\mathrm{e}_2 & \mathrm{e}_1 & 0 \end{bmatrix}$.

Refining the fundamental matrix F:
The initial estimate of the fundamental matrix may not be as accurate as needed. Hence it is refined using Levenberg-Marquardt non-linear least squares method. The cost function computes world coordinate of each pair of correspondences and back-projects them to calculates the geometric error. The LM method reduces the cost function to refine **F**. The cost function:

$$d^2_{geom} = \sum_i \left(\|\vec{x}_i - \vec{\hat{x}}_i\|^2 + \|\vec{x}'_i - \vec{\hat{x}}'_i\|^2 \right)$$

Finding the world coordinates:
Given a correspondence $(\vec{x}', \vec{x})$ and the projection matrices P and P′, the corresponding world point $\vec{\mathrm{X}}$ is computed by "triangulation" such that $\vec{x} = \mathrm{P}\vec{\mathrm{X}}$ and $\vec{x}' = \mathrm{P}'\vec{\mathrm{X}}$. The homogeneous system of equations is formed as $A\vec{\mathrm{X}} = \vec{0}$

$$A\vec{\mathrm{X}} = \begin{bmatrix} x\vec{\mathrm{P}}_3^T - \vec{\mathrm{P}}_1^T \\ y\vec{\mathrm{P}}_3^T - \vec{\mathrm{P}}_2^T \\ x'\vec{\mathrm{P}}_3'^T - \vec{\mathrm{P}}_1'^T \\ y'\vec{\mathrm{P}}_3'^T - \vec{\mathrm{P}}_2'^T \end{bmatrix} \vec{\mathrm{X}} = \vec{0}$$

where $\mathrm{P} = \begin{bmatrix} \vec{\mathrm{P}}_1^T \\ \vec{\mathrm{P}}_2^T \\ \vec{\mathrm{P}}_3^T \end{bmatrix}$.
The system is solved with linear least squares method and the solution $\vec{\mathrm{X}}$ is given by the smallest eigenvector of $A^T A$.

Calculating Homographies:
The homographies **H** and **H′** transform the epipoles to infinity along the x-axis. This process rectifies the images. The homographies are computed in the following process, by computing **H′** first and then computing **H**.

H′ is computed as:

$$\mathbf{H}' = \mathbf{T_2 G R T_1}$$

- $\mathbf{T_1}$ is a translation matrix that moves the image center to origin and is computed as:

$$\mathbf{T_1} = \begin{bmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{bmatrix}$$

  where w and h are width and height of the image.

- **R** is a rotation matrix that rotates the epipoles onto the x-axis and is computed as:

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

  where $\theta = tan^{-1}\left(\dfrac{e'_y - h/2}{-e'_x - w/2} \right)$.

- $\mathbf{G}$ is a homography matrix that takes the epipoles to infinity and is computed as:

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

  where $f = (e'_x - w/2) * cos(\theta) - (e'_y - h/2) * sin(\theta)$

- $\mathbf{T_2}$ is a translation matrix that takes the image center back to its original place and is computed as:

$$\mathbf{T_2} = \begin{bmatrix} 1 & 0 & w/2 - \tilde{x} \\ 0 & 1 & h/2 - \tilde{y} \\ 0 & 0 & 1 \end{bmatrix}$$

  where $(\tilde{x},\ \tilde{y})$ is the rectified image center when $\mathbf{H'_{center}} = \mathbf{GRT_1}$ is applied to (x, y) original image center.

- Finally, $\mathbf{H'} = \mathbf{T_2 H'_{center}}$

$\mathbf{H}$ is computed as:

- Matrices $\mathbf{GRT_1}$ are computed using image 1 and its epipole using the same steps shown above. An initial homography matrix $\mathbf{H_0} = \mathbf{GRT_1}$ is computed.

- $\mathbf{H}$ is computed such that it not only sends the epipole to infinity on the x-axis, but also makes the epipolar lines in the left camera match up with those in the right camera. To do this, $\mathbf{H}$ is computed to minimize the distance between $\mathbf{H_0} x_i$ and $\mathbf{H'} x'_i$. This can be represented as:

$$cost = \sum_i (dist(\mathbf{H_0} x_i, \mathbf{H'} x'_i))$$

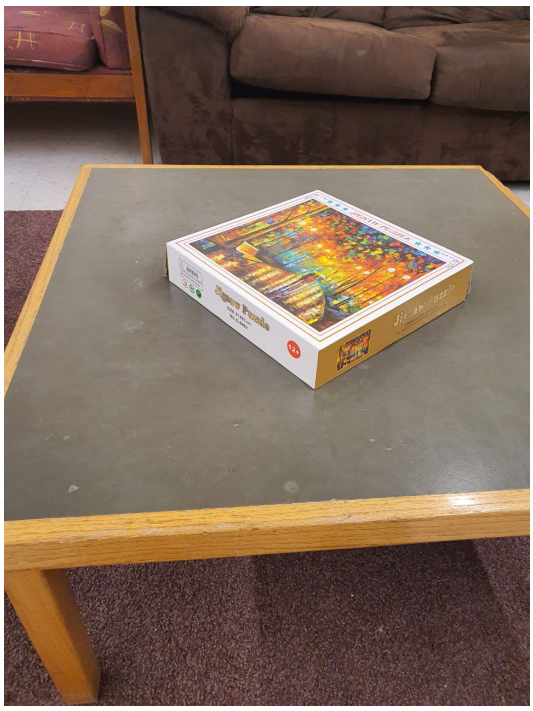
$$cos = \sum_i (a x_i + b y_i + c - x'_i)$$

  Hence, $\mathbf{H_A} = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ can be formed and applied to $\mathbf{H_0}$. This gives $\mathbf{H} = \mathbf{H_A H_0}$. Similar to how $\mathbf{T_1}$ is calculated above, compute the $\mathbf{T_1}$ to move the center of the image back to its original place and apply it to give $\mathbf{H} = \mathbf{T_1 H_A H_0}$.

Applying Homographies:
For this process, OpenCV's "warpPerspective" function was used to apply homographies to the images. The results are shown below.

1.2 Outputs and Results

The left and right images used for this task:

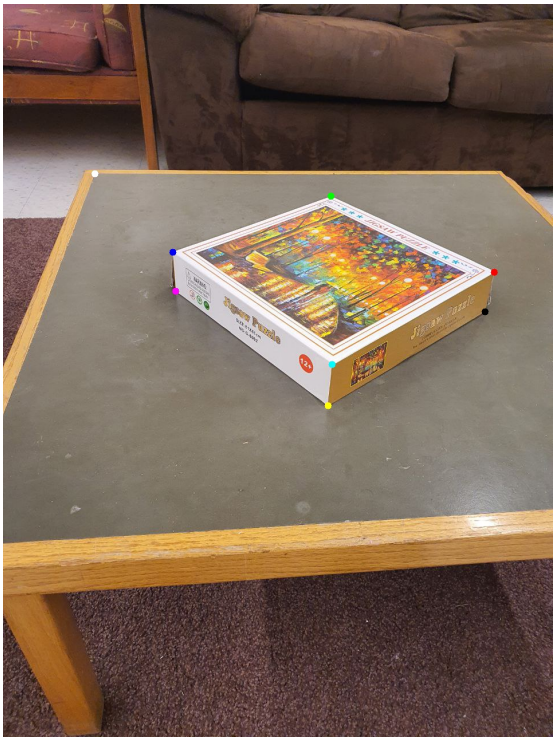


(a) Left

(b) Right

Figure 1: Input Images

The 8 manually picked correspondences:

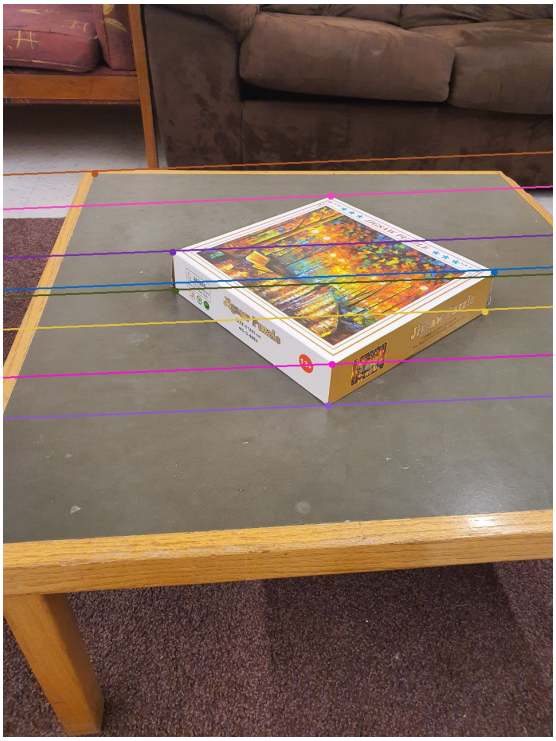


(a) Left

(b) Right

Figure 2: Manually Picked Correspondences

The Epipolar lines for each images:



(a) Left

(b) Right

Figure 3: Epipolar Lines

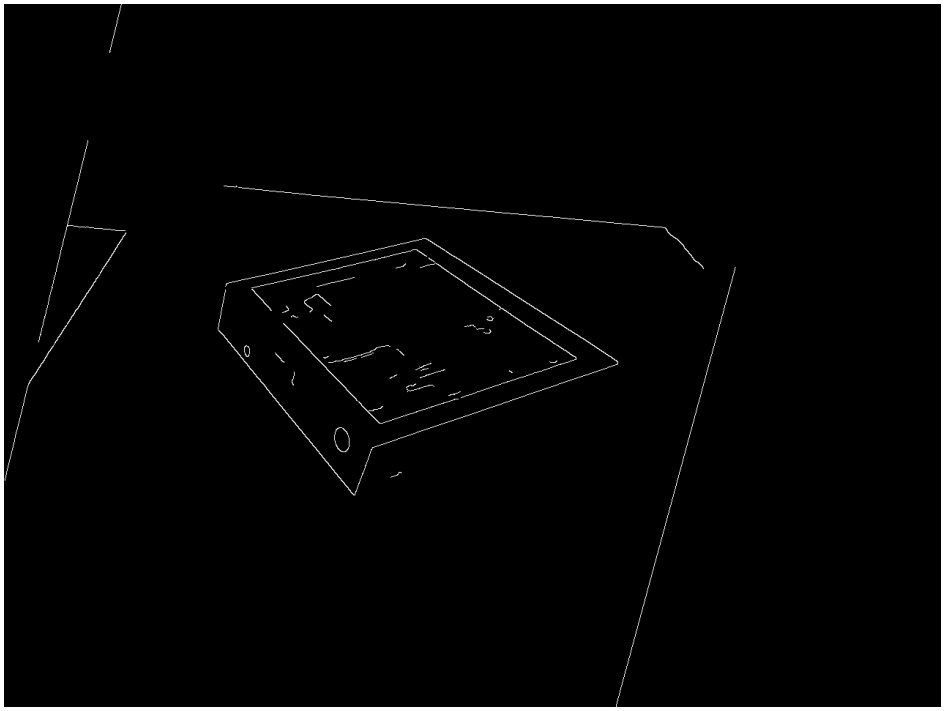
The rectified images:

(a) Left

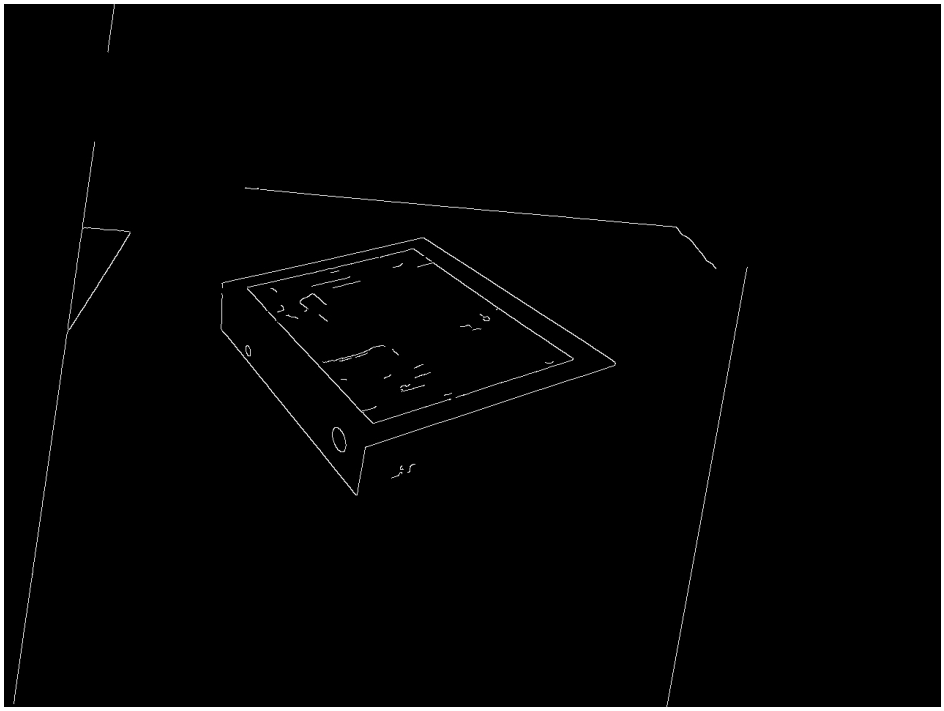
(b) Right

Figure 4: Rectified Images

The Canny edge detector outputs:



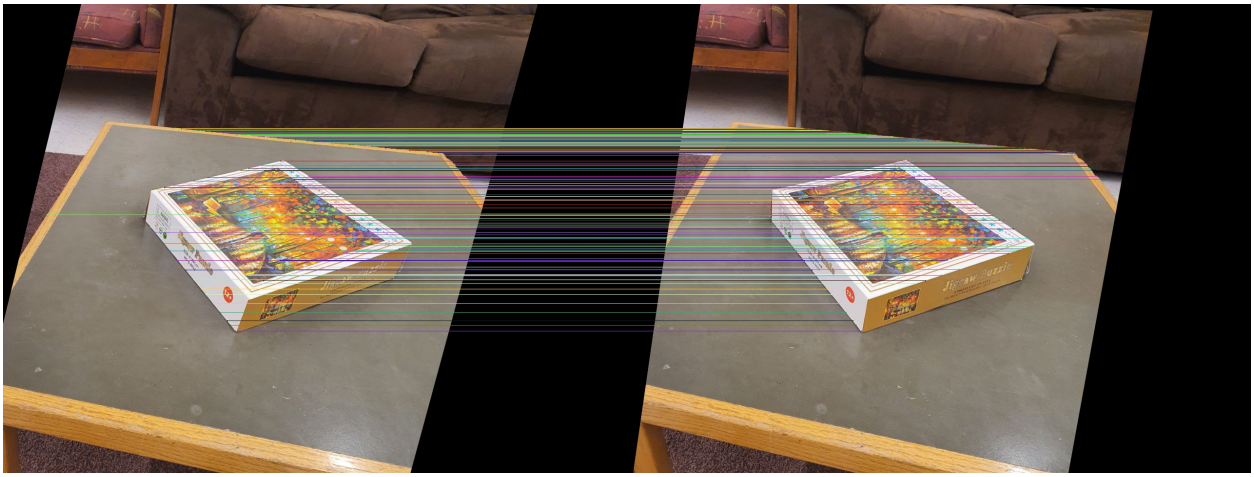
(a) Left



(b) Right

Figure 5: Canny Edges

Automatically generated Correspondences:



(a) Left

Figure 6: Correspondences

Projective reconstruction in 3D plot

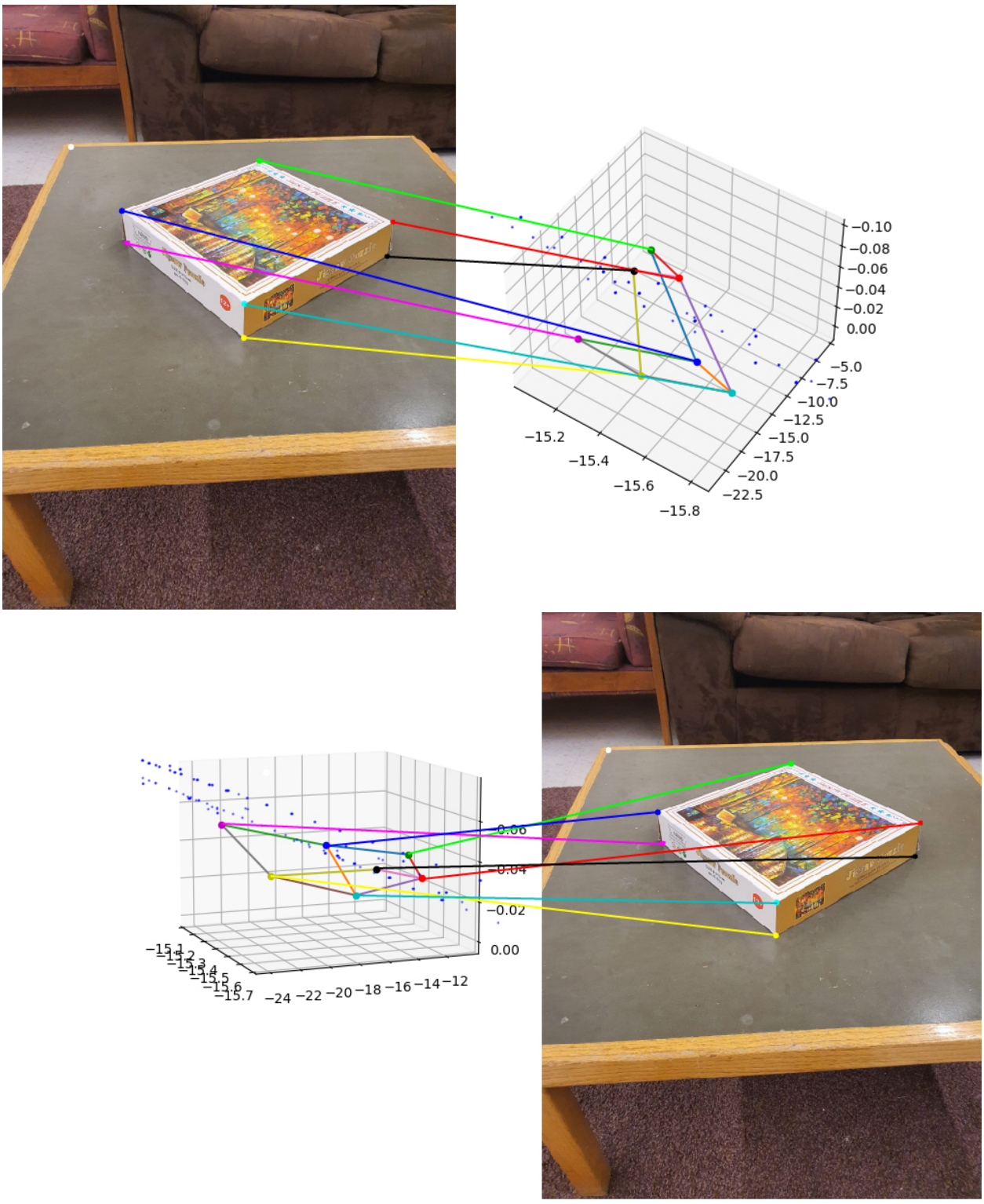


Figure 7: 3D plots of projective reconstruction

1.3 Observations

The image rectification worked well, however projective reconstruction did not turn out as well as expected. As can be seen from the projective reconstruction plots, the structure has projective distortion since the parallel edges of the box in the original scene are converging and diverging. For example in the top plot, the edges that connect blue and cyan points and pink and yellow points, parallel in the original scene, are converging in the plot (brown and orange lines).

Task 2: The Loop and Zhang Algorithm

2.1 Overview

The Loop and Zhang algorithm computes the rectifying homographies as:

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

where

- $\mathrm{H_p}$ and $\mathrm{H'_p}$ are purely projective homographies that send the epipoles to infinity in the image plane in a way such that the direction causes the least distortion to the images as possible
- $\mathrm{H_{sim}}$ and $\mathrm{H'_{sim}}$ are similarity homographies that can only rotate, translate and uniformly scale the image. They rotate the epipoles at infinity onto the X-axis without causing any distortion.
- $\mathrm{H_{sh}}$ and $\mathrm{H'_{sh}}$ are shearing homographies. They help reduce any nonlinear distortion that are caused by the $\mathrm{H_p}$ and $\mathrm{H'_p}$ by introducing additional degrees of freedom into the overall rectification transformation.

2.2 Observations

As can be seen from the outputs of Loop and Zhang, the rectified images are very similar to the ones produced in task 1. The correspondence matching is far better in this algorithm than task 1 and as can be seen from the images, the correspondences match perfectly.

2.3 Outputs and Results

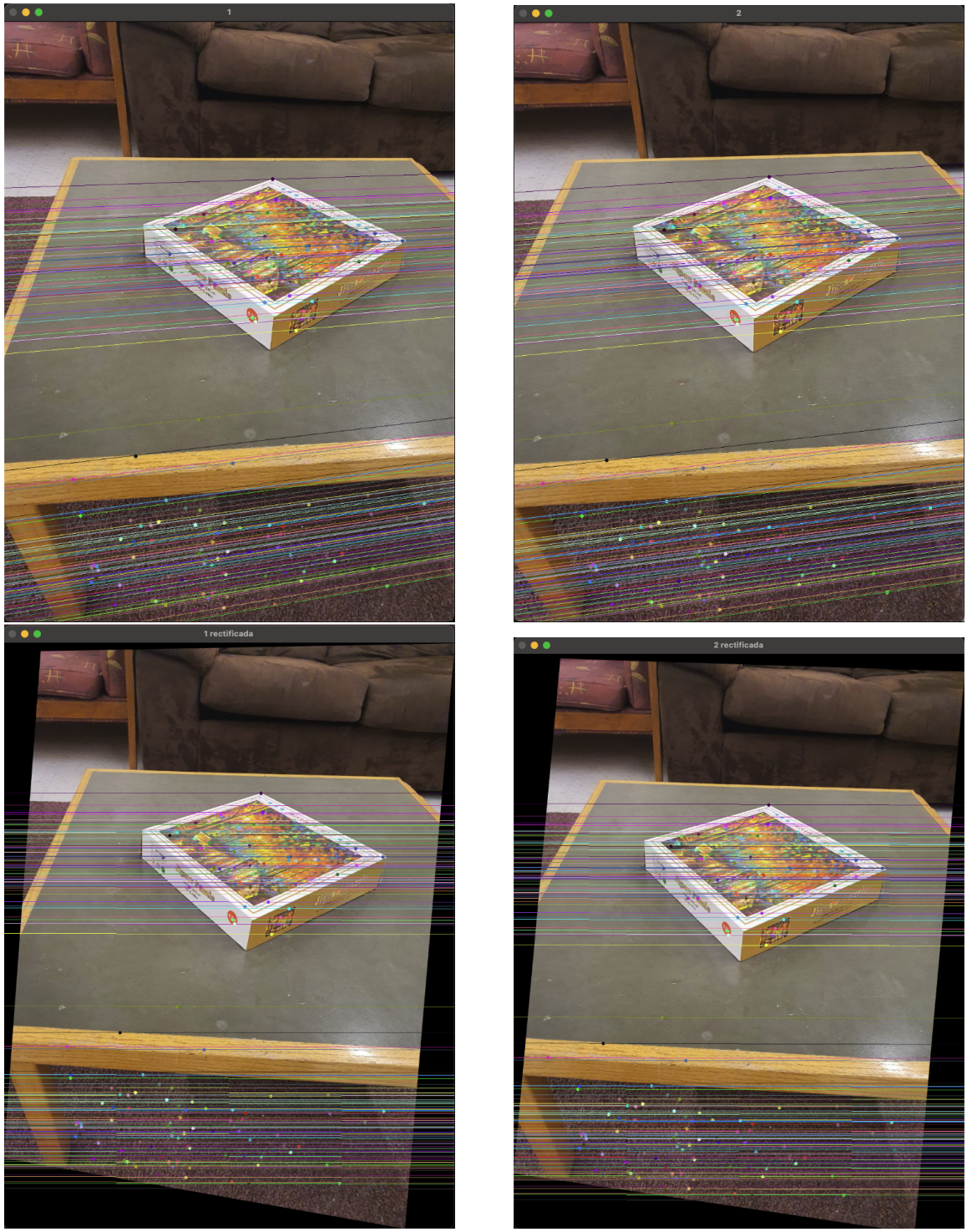


Figure 8: Loop and Zhang Algorithm Outputs

Task 3: Dense Stereo Matching

3.1 Background and Methodology

Dense Stereo Matching involves finding accurate pixel correspondences given stereo rectified images. This is done using the Census Transform. **Brief Summary of the process:**

1. For a pixel in the left image, the local intensity distribution in a M $\times$ N neighborhood around the pixel is formed and passed through a binary threshold to form a bit vector that is computed by setting a bit one wherever the pixel value is strictly greater than the center pixel value.

2. Similarly, for a pixel in the right image, at the same coordinates as the first pixel, the local intensity distribution in a M $\times$ N neighborhood around the pixel is formed and passed through a binary threshold to form a bit vector that is computed by setting a bit one wherever the pixel value is strictly greater than the center pixel value.

3. A bitwise XOR is performed between the two bit vectors of the two pixels from each image. The number of 1s in the result gives us the data cost between the two pixels.

4. In the same row, a pixel in the right image at a distance d from the original pixel is selected and the steps above are repeated. The data cost is computed for each pixel for $d \in (0, \ldots, d_{max})$.

5. The disparity value d is picked such the data cost is minimized. In the case of multiple minima, the first disparity value is picked that results in the minimum data cost while iterating from 0 to d_{max}. This disparity value is then associated with the original pixel in the left image.

6. Repeat this process for all the pixels in the left image. This forms the disparity map for the left image.

7. With the given ground truth disparity map D and the above estimated disparity map $\hat{\text{D}}$, the error $|\text{D} - \hat{\text{D}}|$ is calculated and the accuracy is computed as the percentage of N (valid pixels) in D such that the error is $\leq \delta$. For this assignment $\delta \leq 2$.

3.2 Outputs and Results

The left and right input images:

(a) Left Image (b) Right Image

Figure 9: Left and Right images

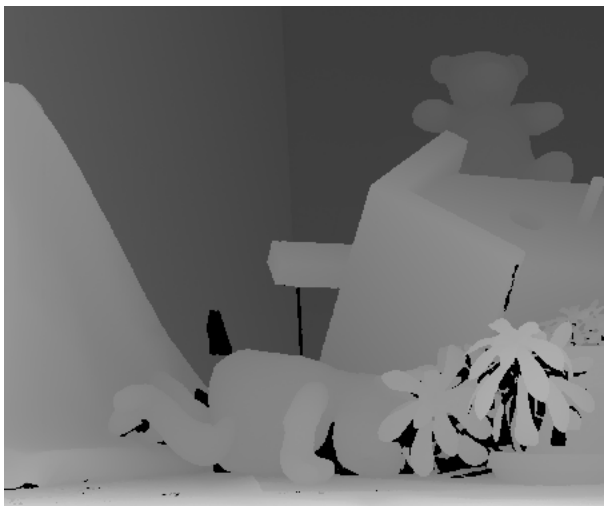
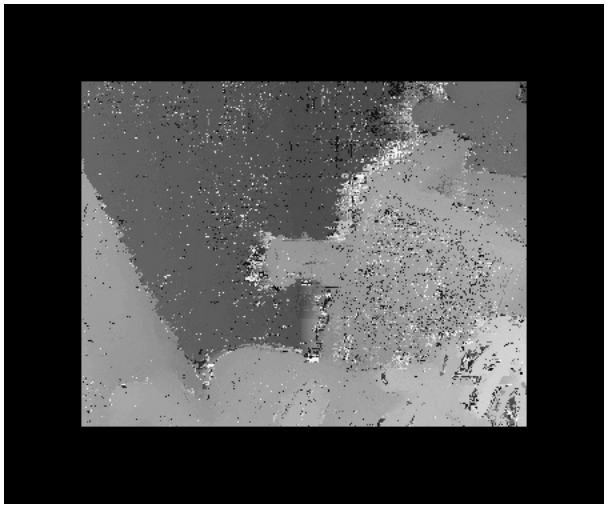
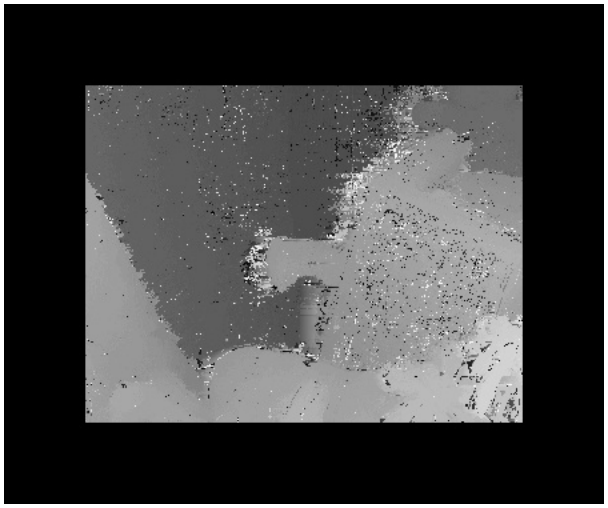


Figure 10: Left Disparity map Ground Truth

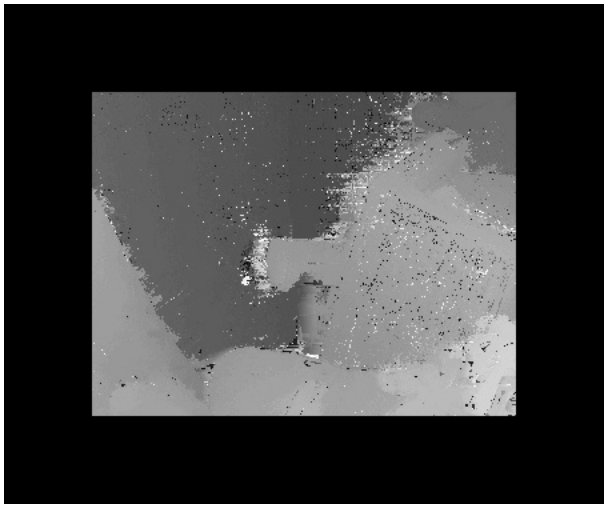
Disparity Maps:



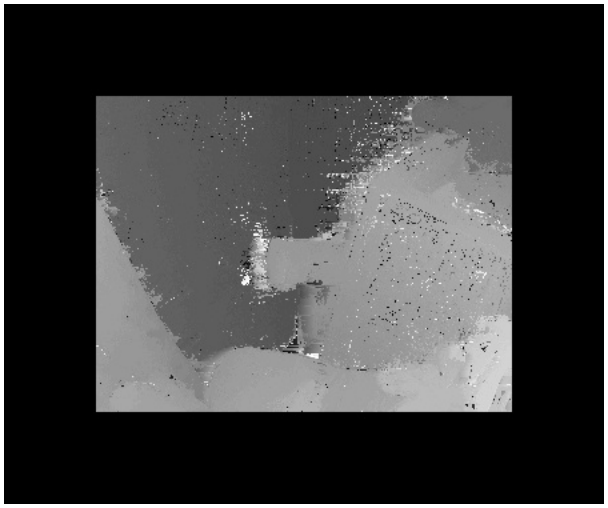
(a) Disparity Map for M = 13



(b) Disparity Map for M = 18



(c) Disparity Map for M = 29



(d) Disparity Map for M = 35

Figure 11: Disparity Maps for different M

Error Maps for $\delta \leq 2$:

(a) Accuracy: 80.95% for M = 13

(b) Accuracy: 83.54% for M = 18

(c) Accuracy: 84.13% for M = 29

(d) Accuracy: 83.75% for M = 35

Figure 12: Error Maps for different M

3.3 Observations

The disparity maps were reduced to valid sub areas to get the accuracies. As can be seen from the outputs, the accuracy plateaus around M = 35. The accuracy may improve if a window of size M × N is used rather than M × M (square).

Source Code

Task 1 Source Code

```
#####==========Task 1: Projective Stereo Reconstruction==========#####

import numpy as np
import cv2 as cv
import math
import matplotlib.pyplot as plt
from scipy import optimize
from matplotlib.patches import Circle, ConnectionPatch
from mpl_toolkits.mplot3d import Axes3D

###============================Functions============================###
# Image Rectification
def drawPoints(image, pts, r, color, t):
    img = image.copy()
    for i in range(pts.shape[0]):
        pt1 = int(pts[i,0])
        pt2 = int(pts[i,1])
        cv.circle(img, (pt1,pt2), r, color, t)
    return img

def drawPts(image, pts, r, t):
    img = image.copy()
    for i,c in enumerate([(255,0,0), (0,255,0), (0,0,255), (255,255,0), (
                                        255,0,255), (0,255,255), (0,0,0),
                                         (255,255,255)]):
        pt1 = int(pts[i,0])
        pt2 = int(pts[i,1])
        cv.circle(img, (pt1,pt2), r, c, t)
    return img

def normalize(coords):
    # x' = Tx
    pts_mean = np.mean(coords,axis=0)
    dists = np.linalg.norm(coords - pts_mean,axis=1)
    dist_mean = np.mean(dists)
    s = np.sqrt(2) / dist_mean

    # T matrix
    Tmat = np.array([[s, 0, -s*pts_mean[0]], [0, s, -s*pts_mean[1]], [0, 0
                                        , 1]])

    # x'
    coords_hc = np.array([[x, y, 1.] for x,y in coords])
    coords_norm = np.dot(Tmat, coords_hc.T).T

    return coords_norm, Tmat

def calcF(coords1, coords2):
    # Linear solution Af = 0
```

```
    A = np.zeros((8,9))
    for i in range(8):
        x1, y1 = coords1[i,0], coords1[i,1]
        x2, y2 = coords2[i,0], coords2[i,1]
        A[i] = np.array([x2*x1, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1])
    # Linear least squares
    U,D,V = np.linalg.svd(A)
    f = V[-1]
    Fmat = np.reshape(f, (3,3))

    # Condition F to be rank=2
    U,D,V = np.linalg.svd(Fmat)
    D_p = np.array([[D[0], 0, 0], [0, D[1], 0], [0, 0, 0]])
    Fmat = np.dot(U, np.dot(D_p, V))
    return Fmat

def calcEpipole(Fmat):
    # e'F = 0 and Fe = 0
    U,D,V = np.linalg.svd(Fmat)
    e_1 = V[-1].T
    e_2 = U[:,-1]

    e_1 = e_1 / e_1[2]
    e_2 = e_2 / e_2[2]

    return np.array(e_1), np.array(e_2)

def getPmat(e_2, Fmat):
    # P = [I | 0] and P' = [e_2_x * Fmat | e_2]
    P_1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
    e_2_x = np.array([[0, -e_2[2], e_2[1]], [e_2[2], 0, -e_2[0]], [-e_2[1]
                                            , e_2[0], 0]])
    P_2 = np.hstack((np.dot(e_2_x, Fmat), np.transpose([e_2])))
    return P_1, P_2

def costFunc(fVec, coords1, coords2):
    d_geom = []
    Fmat = np.reshape(fVec, (3,3))
    e_1, e_2 = calcEpipole(Fmat)
    P_1, P_2 = getPmat(e_2, Fmat)
    P1_1, P1_2, P1_3 = P_1[0,:], P_1[1,:], P_1[2,:]
    P2_1, P2_2, P2_3 = P_2[0,:], P_2[1,:], P_2[2,:]
    coords1_hc = np.array([[x, y, 1.] for x,y in coords1])
    coords2_hc = np.array([[x, y, 1.] for x,y in coords2])

    # Iterate through each correspondence pair and find error dist
    for i in range(len(coords1)):
        x1, y1 = coords1[i,0], coords1[i,1]
        x2, y2 = coords2[i,0], coords2[i,1]

        A = np.zeros((4,4))
        A[0] = x1*P1_3 - P1_1
        A[1] = y1*P1_3 - P1_2
        A[2] = x2*P2_3 - P2_1
```

```
        A[3] = y2*P2_3 - P2_2

        # World Pt X
        U,D,V = np.linalg.svd(A.T @ A)
        world_X = V[-1,:].T
        world_X = world_X / world_X[3]

        # Estimate the corresponding x and x'
        # x = PX and x' = P'X
        x1_esti = np.dot(P_1, world_X)
        x1_esti = x1_esti/x1_esti[2]

        x2_esti = np.dot(P_2, world_X)
        x2_esti = x2_esti / x2_esti[2]

        # Euclidean dist error
        d_geom.append(np.linalg.norm(coords1_hc[i] - x1_esti)**2)
        d_geom.append(np.linalg.norm(coords2_hc[i] - x2_esti)**2)
    d_geom = np.ravel(d_geom)
    return np.array(d_geom)

def drawlines(image1, image2, lines, pts1src, pts2src):
    ''' Referenced from
    https://docs.opencv.org/master/da/de9/tutorial_py_epipolar_geometry.
                                            html '''
    img1 = image1.copy()
    img2 = image2.copy()
    r, c = img1.shape[1], img1.shape[0]
    np.random.seed(0)
    for r, pt1, pt2 in zip(lines, pts1src, pts2src):
        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [c, -(r[2]+r[0]*c)/r[1]])
        img1 = cv.line(img1, (x0, y0), (x1, y1), color, 2)
        img1 = cv.circle(img1, tuple(pt1), 6, color, -1)
        img2 = cv.circle(img2, tuple(pt2), 6, color, -1)
    return img1, img2

def drawEpipoleLines(img1, img2, Fmat, coords1, coords2):
    pts1 = np.array([[x, y, 1.] for x,y in coords1])
    pts2 = np.array([[x, y, 1.] for x,y in coords2])
    elines1 = np.dot(Fmat.T, pts2.T).T
    elines2 = np.dot(Fmat, pts1.T).T
    img1_el1, img2_el1 = drawlines(img1, img2, elines1, coords1, coords2)
    img2_el2, img1_el2 = drawlines(img2, img1, elines2, coords2, coords1)
    return img1_el1, img2_el1, img2_el2, img1_el2

def calcHomography(img2, coords1, coords2, Fmat_lm):
    '''Referenced from Hw10 2020 sol1'''
    w = img2.shape[1]
    h = img2.shape[0]

    # Get refined epipoles and P matrices
    e_1, e_2 = calcEpipole(Fmat_lm)
```

```
P_1, P_2 = getPmat(e_2, Fmat_lm)

## Compute H_2
# H' = T_2 * GRT
# T matrix translates image center to origin
Tmat = np.array([[1, 0, -w/2], [0, 1, -h/2], [0 , 0, 1]])

# R matrix rotates epipole onto x-axis
phi = np.arctan((e_1[1] - h/2.)/(e_1[0] - w/2.)) # angle between e'
                                                 and x-axis
phi_c, phi_s = math.cos(-phi), math.sin(-phi)
Rmat = np.array([[phi_c, -phi_s, 0], [phi_s, phi_c, 0], [0, 0, 1]])

# G matrix translates epipole to infinity
f = (e_2[0] - w/2)*phi_c - (e_2[1] - h/2)*phi_s
G = np.array([[1, 0, 0], [0, 1, 0], [-1/f, 0, 1]])

# Rectify the image center
H_center = np.dot(G, np.dot(Rmat, Tmat))
centerPts = np.array([w/2, h/2, 1])
centerPts_rec = np.dot(H_center, centerPts)
centerPts_rec = centerPts_rec / centerPts_rec[2]

# T_2 moves the center of the image back to its original center
T_2 = np.array([[1, 0, w/2-centerPts_rec[0]], [0, 1, h/2 -
                                                 centerPts_rec[1]], [0, 0, 1]])

# H_2 final homography
H_2 = np.dot(T_2, H_center)
H_2 = H_2 / H_2[2,2]

## Compute H_1
# H_1 = H_a * H_0

# H_0
# R matrix rotates epipole onto x-axis
phi = np.arctan((e_1[1] - h/2.)/(e_1[0] - w/2.)) # angle between e'
                                                 and x-axis
phi_c, phi_s = math.cos(-phi), math.sin(-phi)
Rmat = np.array([[phi_c, -phi_s, 0], [phi_s, phi_c, 0], [0, 0, 1]])
# G matrix translates epipole to infinity
f = (e_1[0] - w/2)*phi_c - (e_1[1] - h/2)*phi_s
G = np.array([[1, 0, 0], [0, 1, 0], [-1/f, 0, 1]])
H_0 = np.dot(G, np.dot(Rmat, Tmat))

# x_hat_1 = H_0*x_1 and x_hat_2 = H_2 * x_2
coords1_hc = np.array([[x, y, 1.] for x,y in coords1])
coords2_hc = np.array([[x, y, 1.] for x,y in coords2])
A = np.zeros((len(coords1_hc), 3))
b = np.zeros((len(coords1_hc), 1))
for i, pts in enumerate(zip(coords1_hc, coords2_hc)):
    x_hat_1 = np.dot(H_0, pts[0])
    x_hat_1 = x_hat_1 / x_hat_1[2]
```

```
        x_hat_2 = np.dot(H_2, pts[1])
        x_hat_2 = x_hat_2 / x_hat_2[2]

        A[i] = [x_hat_1[0], x_hat_1[1], 1]
        b[i] = x_hat_2[0]
    h_abc = np.dot(np.linalg.pinv(A), b)
    h_abc = h_abc.flatten()
    H_A = np.array([[h_abc[0], h_abc[1], h_abc[2]], [0, 1, 0], [0, 0, 1]])
    H_center = np.dot(H_A,H_0)
    centerPts = np.array([w/2, h/2, 1])
    centerPts_rec = np.dot(H_center, centerPts)
    centerPts_rec = centerPts_rec / centerPts_rec[2]

    # T_2 moves the center of the image back to its original center
    T_1 = np.array([[1, 0, w/2-centerPts_rec[0]], [0, 1, h/2 -
                                        centerPts_rec[1]], [0, 0, 1]])

    # H_2 final homography
    H_1 = np.dot(T_1, H_center)
    H_1 = H_1 / H_1[2,2]
    return H_1, H_2

def calcHomography1(img2, coords1, coords2, Fmat_lm):
    '''This method also gave very similar homography matrices'''
    w = img2.shape[1]
    h = img2.shape[0]

    # Get refined epipoles and P matrices
    e_1, e_2 = calcEpipole(Fmat_lm)
    P_1, P_2 = getPmat(e_2, Fmat_lm)

    ## Compute H_2
    # H' = T_2 * GRT
    # T matrix translates image center to origin
    Tmat = np.array([[1, 0, -w/2], [0, 1, -h/2], [0 , 0, 1]])

    # R matrix rotates epipole onto x-axis
    phi = np.arctan((e_2[1] - h/2.)/(e_2[0] - w/2.)) # angle between e'
                                        and x-axis
    phi = -phi
    phi_c, phi_s = np.cos(phi), np.sin(phi)
    Rmat = np.array([[phi_c, -phi_s, 0], [phi_s, phi_c, 0], [0, 0, 1]])

    # G matrix translates epipole to infinity
    f = (e_1[0] - w/2)*phi_c - (e_1[1] - h/2)*phi_s
    G = np.array([[1, 0, 0], [0, 1, 0], [-1./f, 0, 1]])
    # Rectify the image center
    H_center = np.dot(G, np.dot(Rmat, Tmat))
    centerPts = np.array([w/2, h/2, 1])
    centerPts_rec = np.dot(H_center, centerPts)
    centerPts_rec = centerPts_rec / centerPts_rec[2]
    # T_2 moves the center of the image back to its original center
    T_2 = np.array([[1, 0, w/2-centerPts_rec[0]], [0, 1, h/2 -
                                        centerPts_rec[1]], [0, 0, 1]])
```

```
    # H_2 final homography
    H_2 = np.dot(T_2, H_center)

    ## Compute H_1
    # H_1 = H_a * H_0

    # H_0
    M = np.dot(P_2, np.linalg.pinv(P_1))
    H_0 = np.dot(H_2, M)
    # x_hat_1 = H_0*x_1 and x_hat_2 = H_2 * x_2
    coords1_hc = np.array([[x, y, 1.] for x,y in coords1])
    coords2_hc = np.array([[x, y, 1.] for x,y in coords2])
    A = np.zeros((len(coords1_hc), 3))
    b = np.zeros((len(coords1_hc), 1))
    for i, pts in enumerate(zip(coords1_hc, coords2_hc)):
        x_hat_1 = np.dot(H_0, pts[0])
        x_hat_1 = x_hat_1 / x_hat_1[2]

        x_hat_2 = np.dot(H_2, pts[1])
        x_hat_2 = x_hat_2 / x_hat_2[2]

        A[i] = [x_hat_1[0], x_hat_1[1], 1]
        b[i] = x_hat_2[0]
    h_abc = np.dot(np.linalg.pinv(A), b)
    h_abc = h_abc.flatten()
    H_A = np.array([[h_abc[0], h_abc[1], h_abc[2]], [0, 1, 0], [0, 0, 1]])
    H_center = np.dot(H_A,H_0)
    centerPts = np.array([w/2, h/2, 1])
    centerPts_rec = np.dot(H_center, centerPts)
    centerPts_rec = centerPts_rec / centerPts_rec[2]

    # T_2 moves the center of the image back to its original center
    T_1 = np.array([[1, 0, w/2-centerPts_rec[0]], [0, 1, h/2 -
                                        centerPts_rec[1]], [0, 0, 1]])

    # H_2 final homography
    H_1 = np.dot(T_1, H_center)

    H_2 = H_2 / H_2[2,2]
    H_1 = H_1 / H_1[2,2]
    return H_1, H_2

# Interest Point Detection

def getEdges(img, sigma, lt, ht, apetsize):
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    gray = cv.GaussianBlur(gray,(3,3), sigma)
    edges = cv.Canny(gray, lt, ht, apetsize)
    return edges

def getDistance(img1, img2, coord1, coord2, windowSize, mode):
    x1,y1 = coord1
    x2,y2 = coord2
```

```
    # Create window
    r = int(windowSize/2)
    u1, u2 = max(0, y1-r), max(0, y2-r)
    d1, d2 = min(img1.shape[0], y1+r+1), min(img2.shape[0], y2+r+1)
    l1, l2 = max(0, x1-r), max(0, x2-r)
    r1, r2 = min(img1.shape[1], x1+r+1), min(img2.shape[1], x2+r+1)
    reg1 = img1[u1 : d1, l1 : r1]
    reg2 = img2[u2 : d2, l2 : r2]
    # get distance
    if mode == 'NCC':
        m1 = np.mean(reg1)
        m2 = np.mean(reg2)
        d = 1 - np.sum((reg1-m1)*(reg2-m2))/(np.sqrt(np.sum(np.square(reg1
                                                -m1))*np.sum(np.square(reg2-
                                                m2)))+1e-5)
    elif mode =='SSD':
        d = np.sum(np.square(reg1-reg2))
    return d

def getCorres(edges1, edges2, searchDist, windowSize, maxCorrs, mode):
    corrs = []
    dist = []
    sd = int(searchDist/2)
    for row in range(edges1.shape[0]):
        colsL_valid = np.where(edges1[row]>0)[0]
        if len(colsL_valid) == 0:
            continue
        for colL in colsL_valid:
            regL = min(colL-sd, colL)
            regR = max(colL+sd+1, edges2.shape[1])
            colsR = edges2[row, regL:regR]
            colsR_valid = np.where(colsR>0)[0]
            if len(colsR_valid)==0:
                continue
            colR = colsR_valid[0]+colL
            corrs.append([[colL, row],[colR, row]])
            d = getDistance(edges1, edges2, [colL, row], [colR, row],
                                                    windowSize, mode)
            dist.append(d)
    sorted_corrs = [pt for _, pt in sorted(zip(dist, corrs), key=lambda
                                        dist: dist[0])]
    if maxCorrs > 0:
        sorted_corrs = sorted_corrs[0:maxCorrs]
    return sorted_corrs

def drawCorrs(img1, img2, corrs, showCorrs):
    # Concatenate images
    comb_img = np.concatenate((img1, img2), 1)
    r = img1.shape[1]
    np.random.seed(1)
    for i in range(len(corrs)):
        if (not i % showCorrs == 0):
            continue
        color = tuple(np.random.randint(0, 255, 3).tolist())
```

```
        pt1 = tuple(corrs[i][0])
        pt2 = tuple(np.array(corrs[i][1]) + [r, 0])
        cv.circle(comb_img, pt1, 2, color, -1)
        cv.circle(comb_img, pt2, 2, color, -1)
        cv.line(comb_img, pt1, pt2, color, 1)
    return comb_img

# Projective Reconstruction
def backProj(corrs, Fmat_lm):
    world_coords = []
    e_1, e_2 = calcEpipole(Fmat_lm)
    P_1, P_2 = getPmat(e_2, Fmat_lm)
    P1_1, P1_2, P1_3 = P_1[0,:], P_1[1,:], P_1[2,:]
    P2_1, P2_2, P2_3 = P_2[0,:], P_2[1,:], P_2[2,:]

    # Iterate through each correspondence pair and back project
    for i in range(len(corrs)):
        coord1 = corrs[i][0]
        coord2 = corrs[i][1]
        x1,y1 = coord1[0],coord1[1]
        x2,y2 = coord2[0],coord2[1]

        A = np.zeros((4,4))
        A[0] = x1*P1_3 - P1_1
        A[1] = y1*P1_3 - P1_2
        A[2] = x2*P2_3 - P2_1
        A[3] = y2*P2_3 - P2_2

        # World Pt X
        U,D,V = np.linalg.svd(A.T @ A)
        world_X = V[-1,:].T
        world_X = world_X / world_X[3]
        world_coords.append(world_X)
    world_coords= np.reshape(world_coords, (len(corrs), 4))
    return world_coords

def applyTransform(coords, H):
    pts = []
    for coord in coords:
        pt = np.array([coord[0], coord[1], 1.])
        pt = np.dot(H, pt)
        pt = pt / pt[-1]
        pts.append([pt[0], pt[1]])
    return np.array(pts)


###=========================Main Function==========================###

# Initial Coordinates
coords1 = np.array([[296,433],[572,335],[857, 468],[574,629],[301,501],[
                                        567,701],[841,537],[160,296]])  # x
coords2 = np.array([[253, 436],[544,330],[826,459],[512,632],[263,503,],[
                                        511,704],[814,532],[145,300]])   # x'
```

```
# Input images
img1 = cv.imread('HW9/task1/1.jpg')
img2 = cv.imread('HW9/task1/2.jpg')

img1_pts = drawPts(img1, coords1, 6, -1)
img2_pts = drawPts(img2, coords2, 6, -1)
cv.imwrite('HW9/task1/img1_pts.jpg',img1_pts)
cv.imwrite('HW9/task1/img2_pts.jpg',img2_pts)


##==============Image Rectification============##

## Step 1: Normalized 8-point algo for F
# Normalize
coords1_hat, Tmat_1 = normalize(coords1)    # x_hat, T
coords2_hat, Tmat_2 = normalize(coords2)    # x'_hat, T'

# Estimate initial F_hat with x_hat and x'_hat
F_hat = calcF(coords1_hat, coords2_hat)

# Denormalize
# F = T'.T * F_hat * T
Fmat = np.dot(Tmat_2.T, np.dot(F_hat, Tmat_1))
Fmat = Fmat / Fmat[2,2]
print("Initial Guess:\n",Fmat)


## Step 2: Estimate left and right epipole e and e'
e_1, e_2 = calcEpipole(Fmat)
img1_el1, img2_el1, img2_el2, img1_el2 = drawEpipoleLines(img1, img2, Fmat
                                        , coords1, coords2)
cv.imwrite('HW9/task1/img1_elines1.jpg', img1_el1)
cv.imwrite('HW9/task1/img2_elines1.jpg', img2_el1)
cv.imwrite('HW9/task1/img1_elines2.jpg', img2_el2)
cv.imwrite('HW9/task1/img2_elines2.jpg', img1_el2)

## Step 3: Estimate P and P' in canonical form
P_1, P_2 = getPmat(e_2, Fmat)


## Step 4 and 5: Refine the right projection matrix  P   with Nonlinear
                                        optimization
fVec = Fmat.flatten()
Fmat_lm_sol = optimize.least_squares(costFunc, fVec, args=[coords1,
                                        coords2], method='lm')
Fmat_lm = Fmat_lm_sol.x
Fmat_lm = np.reshape(Fmat_lm, (3,3))
# Condition refined F matrix
U,D,V = np.linalg.svd(Fmat_lm)
D_p = np.array([[D[0], 0, 0], [0, D[1], 0], [0, 0, 0]])
Fmat_lm = np.dot(U, np.dot(D_p, V))
Fmat_lm = Fmat_lm / Fmat_lm[2,2]
print("Refined:\n",Fmat_lm)
```

```
## Step 6 and 7: Estimate the right homography matrix  H
H1, H2 = calcHomography(img2, coords1, coords2, Fmat_lm)

## Step 8: Apply homographies
w1, h1 = img1.shape[0], img1.shape[1]
w2, h2 = img2.shape[0], img2.shape[1]


img1_rec = cv.warpPerspective(img1, H1, (w1, h1))
img2_rec = cv.warpPerspective(img2, H2, (w2, h2))
cv.imwrite('HW9/task1/recImg1.jpg', img1_rec)
cv.imwrite('HW9/task1/recImg2.jpg', img2_rec)


##===============Interest Point Detection============##
edges1 = getEdges(img1_rec, 6, 200, 350, 7)
edges2 = getEdges(img2_rec, 6, 200, 350, 7)
cv.imwrite('HW9/task1/img1_edges.jpg', edges1)
cv.imwrite('HW9/task1/img2_edges.jpg', edges2)


corrs = getCorres(edges1, edges2, searchDist = 3, windowSize = 3, maxCorrs
                                        = 1500, mode = 'NCC')
print(len(corrs))
img_corrs = drawCorrs(img1_rec, img2_rec, corrs, 10)
cv.imwrite('HW9/task1/img_corrs.jpg', img_corrs)

##===============Projective Reconstruction============##

# Get world coordinates for the correspondences and plot them
worldCoords = backProj(corrs, Fmat_lm)
test = worldCoords[worldCoords[:,0]>-50]
test = test[test[:,1]>-50]
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(test[:,1], test[:,0], test[:,2], c='b', s = 1, depthshade=True)
# plt.show()

# Get world coordinates for the manually picked points and plot them
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
coords1_rect = applyTransform(coords1, H1)
coords2_rect = applyTransform(coords2, H2)
coords_rect = [[[x1[0],x1[1]],[x2[0],x2[1]]] for x1,x2 in zip(coords1_rect
                                        , coords2_rect)]
coords_rect_world = backProj(coords_rect, Fmat_lm)
# pairs to draw lines making the edges of the box
pairs = [[0, 1], [0,3], [0,4], [1, 2], [2,3], [3,5], [2,6], [4,5], [5,6]]
for pair in pairs:
    ax.plot([coords_rect_world[pair[0]][0],coords_rect_world[pair[1]][0]],
             [coords_rect_world[pair[0]][1],coords_rect_world[pair[1]][1]]
                                                    ,
```

```
            [coords_rect_world[pair[0]][2],coords_rect_world[pair[1]][2]]
                                                        )
for i,c in enumerate(['b', 'g', 'r', 'c', 'm', 'y', 'k', 'w']):
    pt = coords_rect_world[i]
    ax.scatter(pt[0], pt[1], pt[2], c=c, s=20, depthshade=False)
plt.show()
```

Task 2 Source Code

```
#####============Task 3: Dense Stereo Matching==========#####

import numpy as np
import cv2 as cv

###============================Functions============================###
def getDispMap(imageL, imageR, M, dMax):
    imgL = cv.cvtColor(imageL, cv.COLOR_BGR2GRAY) if len(imageL.shape)==3
                                                  else imageL
    imgR = cv.cvtColor(imageR, cv.COLOR_BGR2GRAY) if len(imageR.shape)==3
                                                  else imageR

    win = int(M/2)
    rg = dMax+win
    w, h = imgL.shape[1], imgL.shape[0]
    dMap = np.zeros((h,w), dtype=np.uint8)
    for rowL in range(rg, h-rg):
        print(f'Row: {rowL+1}/{h-rg}')
        for colL in range(w-rg-1, rg-1, -1): #go right to left
            dataCost = []
            winL = imgL[rowL-win:rowL+win+1, colL-win:colL+win+1]
            binL = np.ravel(np.where(winL>imgL[rowL, colL], 1, 0))
            for d in range(dMax+1):
                colR = colL-d
                winR = imgR[rowL-win:rowL+win+1, colR-win:colR+win+1]
                binR = np.ravel(np.where(winR>imgR[rowL, colR], 1, 0))
                cost = np.bitwise_xor(binL, binR)
                cost = np.sum(cost)
                dataCost.append(cost)
            dMap[rowL,colL] = np.argmin(dataCost) # d for which cost in
                                                  min
    dMap = dMap.astype(np.uint8)
    print("Max dMap value:", np.max(dMap))
    # get Mask
    dMapMask = cv.normalize(dMap, dst=None, alpha=0, beta=255, norm_type =
                                                  cv.NORM_MINMAX).astype(np.uint8)
    return dMap, dMapMask, rg

def getAccuracy(gtDispMap, dMap, rg, sigma):
    # Create subarea to remove black borders from disparity maps
    subarea_gt = gtDispMap[rg:gtDispMap.shape[0]-rg, rg:gtDispMap.shape[1]
                                                  -rg]
    subarea_dMap =  dMap[rg:dMap.shape[0]-rg, rg:dMap.shape[1]-rg]
    # Find the error difference
    diff = np.abs(subarea_dMap.astype(np.uint16) - subarea_gt.astype(np.
                                                  uint16)).astype(np.uint8)
    N = cv.countNonZero(subarea_gt[:,:])
    Ntotal = subarea_gt.shape[1]*subarea_gt.shape[0]
    print(f'Valid Pixels N: {N}/{Ntotal}')
    # accuracy
    acc = np.sum(diff<=sigma)/N
    # Error Mask
```

```
    errMask = np.where(diff<=sigma, 255, 0)

    return acc, errMask
###=========================Main Function=========================###
#Input images
imgL = cv.imread('HW9/task3/im2.ppm', 0)
imgR = cv.imread('HW9/task3/im6.ppm', 0)

# Input the left disparity map
gtDispMap = cv.imread('HW9/task3/disp2.pgm',0)
gtDispMap = gtDispMap.astype(np.float32) / 4.0
gtDispMap = gtDispMap.astype(np.uint8)
dMax = np.max(gtDispMap)
print(dMax)

# Use Census transform on left image to make its disparity map for
                                        different M
for M in [13, 18, 29, 35]:
    print(f'M: {M}')
    dMapL, dMapMaskL, rg = getDispMap(imgL, imgR, M, dMax)
    cv.imwrite('HW9/task3/dMapL'+str(M)+'.jpg', dMapMaskL)
    percL, errMaskL = getAccuracy(gtDispMap, dMapL, rg, 2)
    print(f'Accuracy percentage for M {M}: {percL}')
    cv.imwrite('HW9/task3/errMaskL'+str(M)+'.jpg', errMaskL)
```