Purdue University

ECE 661 Computer Vision

Homework 10

Submission: Arjun Kramadhati Gopi

Email: akramadh@purdue.edu

Tasks for this homework

We have two broad tasks for this homework assignment:

1. Create 3D reconstruction from a pair a pair of images recorded with uncalibrated camera.
2. Dense stereo matching

Let us explore these two tasks in detail. For the task 1 we have the following sub tasks that we need to achieve:

1. Image Rectification
2. Projective reconstruction
3. Visual representation of the projective distortion

Image Rectification

Before we talk about stereo rectification, we need to first talk about the method to determine or estimate the fundamental matrix F which defines the relationship between the corresponding pixels in each of the camera and also the world point which corresponds to them.
Given a pair of corresponding points $\vec{x}$ and $\vec{x'}$ we have the following constraint:

$$\vec{x'}^T F\vec{x} = 0 \tag{1}$$

Where we have the following representations for each of the variables in the equation:

$$\vec{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\vec{x'} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

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

Therefore, equation 1 becomes:

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0$$

In other words, we have:

$$A\vec{f} = 0 \tag{2}$$

Where,

$$A = \begin{bmatrix} x'x & x'y & x' & y'x & y'y & y' & x & y & 1 \end{bmatrix}$$

and

$$\vec{f} = \begin{pmatrix} f_{11} & f_{12} & f_{13} & f_{21} & f_{22} & f_{23} & f_{31} & f_{32} & f_{33} \end{pmatrix}^T$$

Knowing all this, we then select at least 8 such pairs of corresponding points. Stacking them together in the same form we get:

$$A\vec{f} = \vec{0} \tag{3}$$

We solve the above equation using the least squares method. By doing that, we obtain an initial rough estimate of the fundamental matrix F. But we still have some ways to go before we have an acceptable F matrix. For the initial rough estimate we solve by a SVD of A wherein the eigenvector with the smallest eigen value is the solution. To have a strict 1-to-1 epipole correspondence, we need to make sure the rank of the fundamental matrix F is two. To do this, we perform a SVD on F to get the u,d and vT values. We then modify the d value by making the smallest eigen value as 0. We compute the value of F again by multiplying the updated decomposed values. We then denormalize the value of the fundmental matrix.
We then refine the fundamental matrix using the LM algorithm. It is the same non-linear least squares refinement that we have used in the previous homework.
The cost function for the optimization is:

$$D^2_{geometeric} = \sum_i (||x_i - x_i^{projected}||^2 + ||x'_i - x_i'^{projected}||^2) \tag{4}$$

The question is how do we get the projected x and x' values. For that we will first need to triangulate the world point for each pair of corresponding points. The procedure to find the world point is simple:

- Estimate the two epipole locations by using:

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

and

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

- Once we have the epipole loctions, we then set the camera matrix values as:

$$P = [I|\vec{0}]$$

$$P' = [[\vec{e'}]_x F|\vec{e'}]$$

Further, we can express these two matrices as:

$$P = \begin{bmatrix} \vec{p}_1 & \vec{p}_2 & \vec{p}_3 \end{bmatrix}$$

$$P' = \begin{bmatrix} \vec{p'}_1 & \vec{p'}_2 & \vec{p'}_3 \end{bmatrix}$$

- Next, we make use of the contraints:

$$\vec{x} = P\vec{X}$$

and

$$\vec{x'} = P'\vec{X}$$

Using their HC representations and taking the cross products:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} X \begin{pmatrix} \vec{p}_1\vec{X} \\ \vec{p}_2\vec{X} \\ \vec{p}_3\vec{X} \end{pmatrix} = 0$$

and

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} X \begin{pmatrix} \vec{p'}_1\vec{X} \\ \vec{p'}_2\vec{X} \\ \vec{p'}_3\vec{X} \end{pmatrix} = 0$$

From these two cross products, we obtain four unique equations which can be written as:

$$A = \begin{bmatrix} x\vec{p_3^T} - \vec{p_1^T} \\ y\vec{p_3^T} - \vec{p_2^T} \\ x'\vec{p_3^T} - \vec{p_1^T} \\ y'\vec{p_3^T} - \vec{p_2^T} \end{bmatrix}$$

- We have the final equation in the form:

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

We estimate the world point after a SVD of A.

Once we have the world point, we then project them back onto the individual camera planes by using the camera matrix and the camera projection equation which will yeild us the two projected points which is needed for the cost function in the optimization process. This is how we estimate an acceptable fundamental matrix.
The next task is to perform stereo rectification. To do this, we send the epipoles of each image to infinity. The procedure to send the epipole of the second image is as follows:

- Estimate the value of T which send the image center to the origin. Where:

$$T = \begin{bmatrix} 1 & 0 & -0.5width \\ 0 & 1 & -0.5height \\ 0 & 0 & 1 \end{bmatrix}$$

- Estimate the roation matrix:

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

where theta is the angle with respect to x-axis which is the rotation angle needed.

- Calculate the G matrix to send the epipole to infinity along the x-axis:

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -0.5f & 0 & 1 \end{bmatrix}$$

- We then estimate another translational to preserve the center.
- The final homography which is used is:

$$H' = T_2GRT_1$$

Next, we send the epipole of the first image to inifinity by doing:

- Estimate $homographyH_0$

$$H_0 = H'M$$

- We then obtain the abc values by minimizing the following:

$$\sum_i (ax_i^{hat} + by_i^{hat} + c - x_i'^{hat})^2$$

- We make use of the abc values to set the values of the homography H_a
- The final homography which is used is;

$$H = H_a H_0$$

Projective reconstruction

Once we have the rectified images, the task of recreating the scene becomes easier. For a given coordinate image pixel on the left image, all we have to do is look in the same corresponding row in the second image to find the corresponding image point in the second image. Once we have the corresponding image points, we then triangulate the world point using the same technique which was described in the above section. We then label and draw lines between the points so that we can see the scene in the 3D perspective.

Dense stereo matching - Census transform

Now we move on to the second major task which is dense stereo matching. The main challenge is to figure out how to identify occluded regions in the images using the information in the two images of the same scene.
The procedure for this is:

- We first construct a MXM window.
- We perform a roster scan of this window over the pixels of the left image which also the first image.
- For each pixel scanned, we find out how far the corresponding pixel on the right image is.
- To do this, we first find the corresponding pixel on the right image (second image).
- Luckily for us, we only need to search in one row of the second image to find the corresponding pixel as the images are rectified.
- For each corresponding pixel distance (d value) we estimate the data cost to determine pixel similarity.
- The data cost is obtained using census transform.
- To elaborate:
  1. We build a M2 size element bitvector for each pixel pair.
  2. The bitvector value is 1 if the pixel value is greater than the center pixel value. It is zero otherwise.
  3. The number of 1s in the XOR operation of the bitvectors in a given window scan of the left and right images dictate the data cost (d).

4. The d value which has the lowest data cost is the value we need to build the disparity map.

Result and analysis

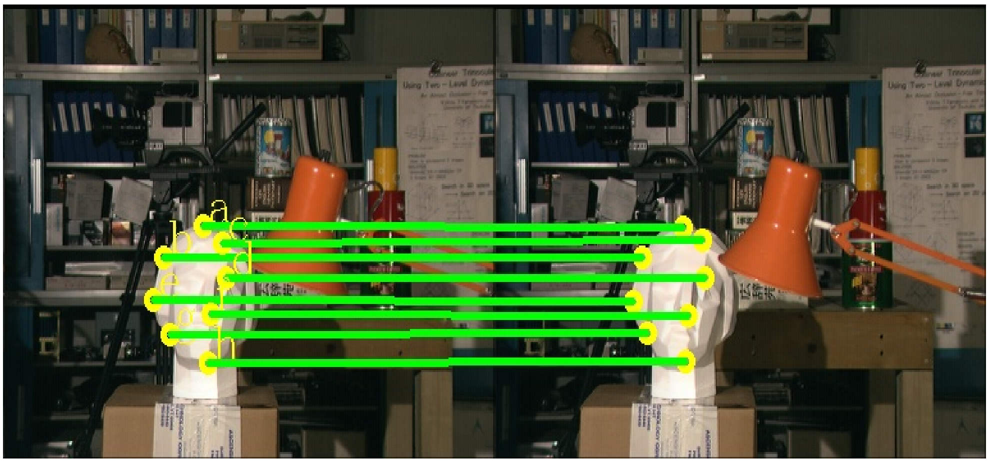

Figure 1: **Interest point - Manually picked**

Figure 2: **Left rectified image**

Figure 3: **Right rectified image**

Figure 4: **Left rectified image with labels**

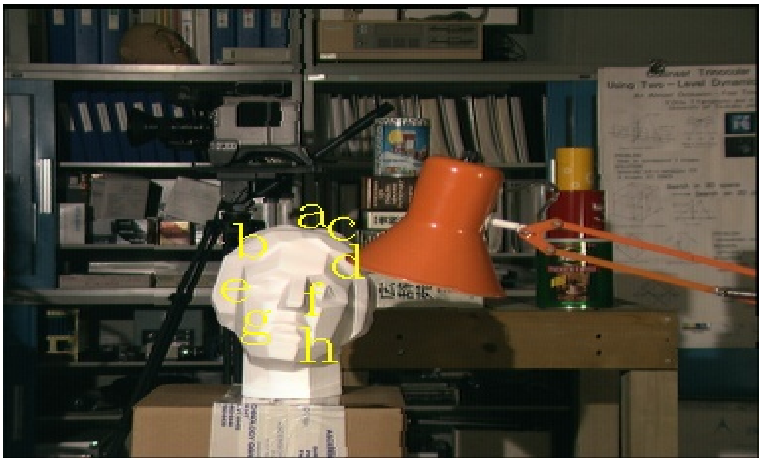

Figure 5: **Right rectified image with labels**

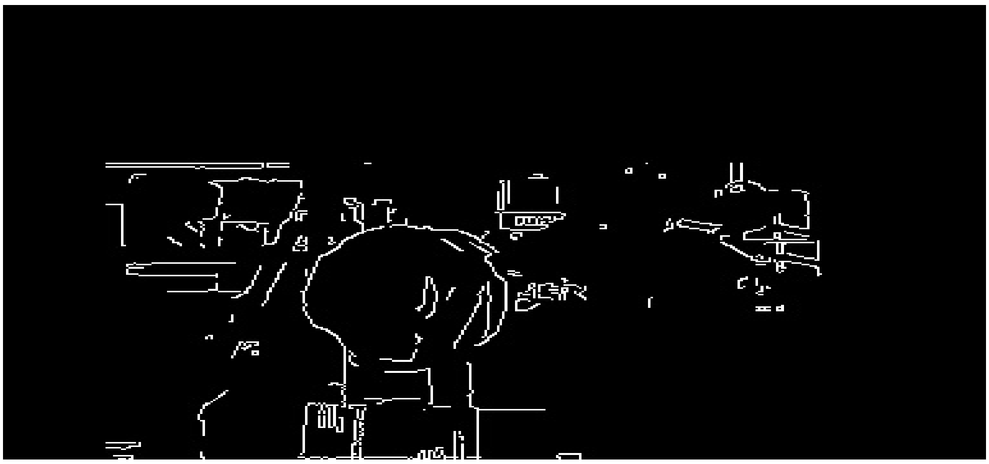

Figure 6: **Canny edge detection Left**

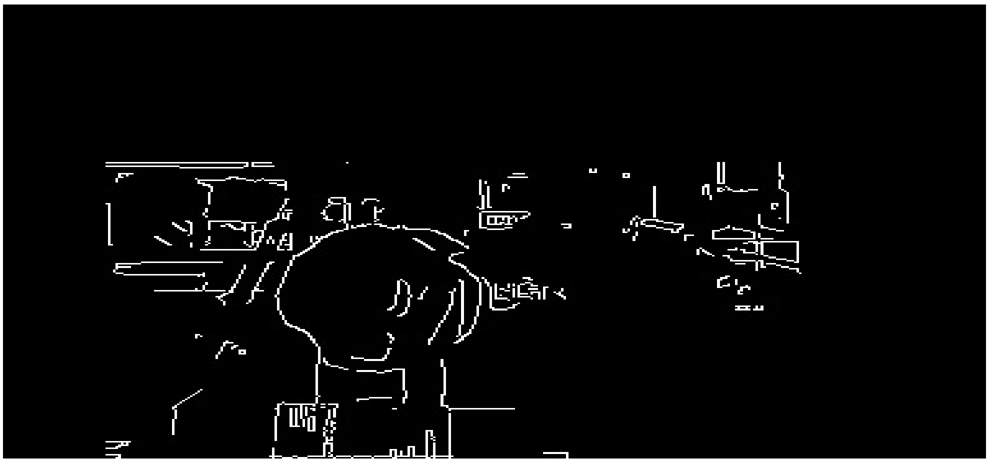

Figure 7: **Canny edge detection Right**

Figure 8: **Corner correspondence**

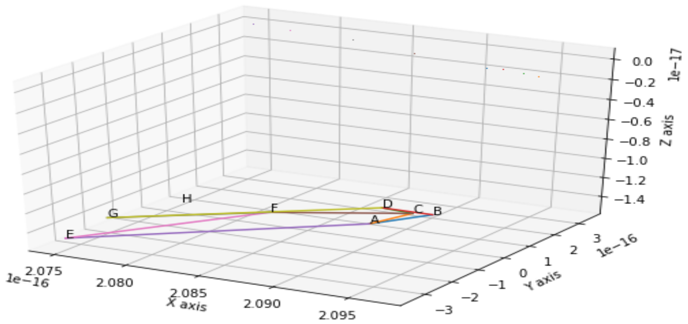

Figure 9: **3D reconstruction**

```
F= [[-2.256e-16 -1.152e+12 -5.649e+13]
 [ 1.152e+12  2.164e-02  4.743e+15]
 [ 5.649e+13 -4.743e+15  1.202e-01]]
Before LM Initial Cost = 1592.49300
After LM Initial Cost = 0.57216
```

Figure 10: **Cost before and after LM refinement**

Task 2

M values 3, 5 and accuracy = 42.8, 59.6 respectively

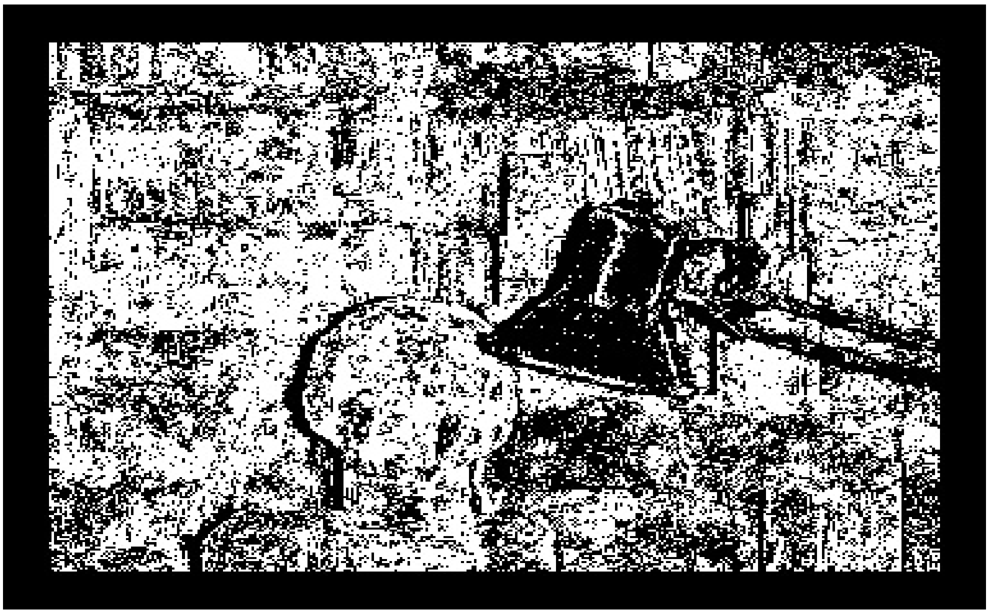

Figure 11:

Figure 12:

Figure 13:

Figure 14:

Source code

The code for task 1 is not entirely mine, many of the functions have been written by referencing from previous year solution: Link

```
"""
Computer Vision - Purdue University - Homework 10

Author : Arjun Kramadhati Gopi, MS-Computer & Information Technology, Purdue University.
Date: Oct 19, 2020

Reference : https://github.com/rmahfuz/Computer-Vision/tree/master/HW09
[TO RUN CODE]: python3 scene_reconstruction.py
"""


import pickle
import cv2 as cv
import numpy as np
from tqdm import tqdm
from scipy.linalg import null_space
from scipy import optimize
import matplotlib.pyplot as plt


class Reconstruct:

    def __init__(self, image_paths, second_task_path):
        """
        Initialize the object
```

```
        :param image_paths: Path to the two images
        """
        self.image_pair = list()
        self.occ_images = list()
        self.occ_grey = list()
        self.grey_image_pair = list()
        self.roiList = list()
        self.roiCoordinates = list()
        self.roiCoordinates_3d = list()
        self.image_specs = list()
        self.left_manual_points = list()
        self.right_manual_points = list()
        self.count = 0
        self.padding = int(31 / 2)
        self.parameter_dict = dict()
        self.parameter_dict['P_triangulation']=np.array([[1, 0,
            0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
        for image_path_index in tqdm(range(len(image_paths)),
            desc='Image Load'):
            file = cv.imread(image_paths[image_path_index])
            self.image_pair.append(file)
            self.grey_image_pair.append(cv.cvtColor(file, cv.
                COLOR_BGR2GRAY))
            self.image_specs.append(file.shape)
        self.reference_center = np.array([[self.image_specs[0][1]
            / 2.0, self.image_specs[0][0] / 2.0, 1]])
        for path in second_task_path:
            file = cv.imread(path)
            self.occ_images.append(file)
            self.occ_grey.append(cv.cvtColor(file,cv.
                COLOR_BGR2GRAY))
        print("----------------------------")
        print("Initialization complete")
        print("----------------------------")

    def schedule(self):
        """
        This function runs all the required processes in sequence
            .
        :return:
        """
        #1Get interest points from user
        self.getROIFromUser(type='No',type2='No')
        #2Process the points: Segregate them based on the left
            and right images
        self.process_points()
        #3Perform stereo rectification
        self.process_rectification()
        #4Optimize
        self.levenberg_marquardt_optimization()
        #5Extract corners
        self.extract_corners()
        #6Corresponding matching
```

```
        self.get_closest_match()
        #73D reconstruction
        self.plot_3d()

    def get_world_individual(self, world_points):
        temp = []
        for i in range(3):
            temp.append(np.array(list(map(lambda val: val[i], world_points))))
        return temp[0],temp[1],temp[2]

    def get_plot(self, worldX,worldY,worldZ):
        d_plot = plt.figure()
        plot_world = d_plot.add_subplot(111, projection='3d')
        plot_world = d_plot.add_subplot(111, projection='3d')
        plot_world.scatter(worldX[:7], worldY[:7], worldZ[:7], color='k', zdir='y')
        for i in range(7):
            plot_world.text(worldX[i], worldZ[i], worldY[i], str(i + 1), zdir='y')
        plot_world.plot([worldX[0], worldX[1]], [worldY[0], worldY[1]], [worldZ[0], worldZ[1]], color='b', zdir='y')
        plot_world.plot([worldX[6], worldX[2]], [worldY[6], worldY[2]], [worldZ[6], worldZ[2]], color='b', zdir='y')
        plot_world.plot([worldX[4], worldX[3]], [worldY[4], worldY[3]], [worldZ[4], worldZ[3]], color='g', zdir='y')
        plot_world.plot([worldX[0], worldX[6]], [worldY[0], worldY[6]], [worldZ[0], worldZ[6]], color='b', zdir='y')
        plot_world.plot([worldX[1], worldX[2]], [worldY[1], worldY[2]], [worldZ[1], worldZ[2]], color='b', zdir='y')
        plot_world.plot([worldX[2], worldX[3]], [worldY[2], worldY[3]], [worldZ[2], worldZ[3]], color='r', zdir='y')
        plot_world.plot([worldX[6], worldX[4]], [worldY[6], worldY[4]], [worldZ[6], worldZ[4]], color='r', zdir='y')
        plot_world.plot([worldX[0], worldX[5]], [worldY[0], worldY[5]], [worldZ[0], worldZ[5]], color='r', zdir='y')
        plot_world.plot([worldX[5], worldX[4]], [worldY[5], worldY[4]], [worldZ[5], worldZ[4]], color='g', zdir='y')
        plot_world.view_init(None, 30)
        return plot_world

    def plot_3d(self):
        world_points = self.triangulate_world_points(self.parameter_dict['corners_all'], self.parameter_dict['
```

```
            P_value_two'])
        worldX, worldY, worldZ = self.get_world_individual(world_points=world_points)
        plot_world = self.get_plot(worldX,worldY,worldZ)
        plot_world.set_xlabel('x')
        plot_world.set_ylabel('y')
        plot_world.set_zlabel('z')
        plt.savefig('3dplot.png')

    def process_rectification(self):
        """
        Perform stereo rectification
        :return:
        """
        x1,x2 = list(map(lambda  x: x[0], self.left_manual_points)),list(map(lambda  x: x[0], self.right_manual_points))
        y1,y2 = list(map(lambda  x: x[1], self.left_manual_points)),list(map(lambda  x: x[1], self.right_manual_points))
        mux_1,mux_2 = np.mean(x1),np.mean(x2)
        muy_1,muy_2 = np.mean(y1), np.mean(y2)
        tx1,tx2 = np.square(x1 - mux_1),np.square(x2 - mux_2)
        ty1,ty2 = np.square(y1 - muy_1),np.square(y2 - muy_2)
        m1,m2 = (1.0 / len(self.left_manual_points)) * np.sum(np.sqrt(np.add(tx1,tx2))),(1.0 / len(self.right_manual_points)) * np.sum(np.sqrt(np.add(ty1,ty2)))
        s1,s2 = np.sqrt(2)/m1,np.sqrt(2)/m2
        x1,x2 = -1 * s1 * mux_1,-1 * s2 * mux_2
        y1,y2 = -1 * s1 * muy_1,-1 * s2 * muy_2
        T1,T2 = np.array([[s1, 0, x1], [0, s1, y1], [0, 0, 1]]), np.array([[s2, 0, x2], [0, s2, y2], [0, 0, 1]])
        self.parameter_dict['T1'] = T1
        self.parameter_dict['T2'] = T2
        self.parameter_dict['NLeft'] = np.matmul(T1, np.array(list(map(lambda x: [x[0], x[1], 1], self.left_manual_points))).T)
        self.parameter_dict['NRight'] = np.matmul(T2, np.array(
         list(map(lambda x: [x[0], x[1], 1], self.right_manual_points))).T)
        NLeftT= self.parameter_dict['NLeft'].T
        NRightT = self.parameter_dict['NRight'].T
        matrixA = self.obtain_matrix_A(NLeftT,NRightT)
        u,d,vt = np.linalg.svd(matrixA)
        v=vt.T
        initial_F_estimate = v[:,v.shape[1]-1]
        assert len(initial_F_estimate)==9
        initial_F_estimate=initial_F_estimate.reshape(3,3)
        initial_F_estimate = self.reinforce_F_estimate(initial_F_estimate,T1,T2)
        self.parameter_dict['F_beta']=initial_F_estimate
        print("----------------------------")
```

```
        print("Initial F estimate complete")
        print("----------------------------")
        e_one,e_two = self.get_nulls(self.parameter_dict['F_beta'])
        H2 = self.get_homography(e_one=e_one,e_two=e_two,type='H2')
        center_value = self.get_updated_center(homography=H2)
        second_T_value = np.array([[1, 0, (self.image_specs[0][1] / 2.0) - center_value[0]], [0, 1, (self.image_specs[0][0] / 2.0) - center_value[1]], [0, 0, 1]])
        H2 = np.matmul(second_T_value,H2)
        H1 = self.get_homography(e_one=e_one, e_two=e_two, type='H1')
        center_value_2 = self.get_updated_center(homography=H1)
        first_T_value = np.array([[1, 0, (self.image_specs[0][1] / 2.0) - center_value_2[0]], [0, 1, (self.image_specs[0][0] / 2.0) - center_value_2[1]], [0, 0, 1]])
        H1 = np.matmul(first_T_value,H1)
        self.parameter_dict['H1&H2']=[H1,H2]
        P_dash_value = self.get_P_values(e_two=e_two,F=self.parameter_dict['F_beta'])
        print("----------------------------")
        print("H1 & H2 estimation complete")
        print("----------------------------")
        self.rectify_image()
        print("----------------------------")
        print("Individual rectification complete")
        print("----------------------------")
        F = np.matmul(np.linalg.pinv(self.parameter_dict['Rectified_Params1'][1].T), self.parameter_dict['F_beta'])
        F = np.matmul(F, np.linalg.inv(self.parameter_dict['Rectified_Params0'][1]))
        tp_1 = list(map(lambda x: [x[1], x[0], 1], self.left_manual_points))
        point_one = np.matmul(H1, np.array(tp_1).T)
        point_one /= point_one[2]
        point_one = point_one.T
        point_one = np.array(list(map(lambda x: [x[1], x[0], x[2]], point_one)))
        tp_2 = list(map(lambda x: [x[1], x[0], 1], self.right_manual_points))
        point_two = np.matmul(H2, np.array(tp_2).T)
        point_two /= point_two[2]
        point_two = point_two.T
        point_two = np.array(list(map(lambda x: [x[1], x[0], x[2]], point_two)))
        rectification = np.hstack((self.parameter_dict['Rectified_Params0'][0], self.parameter_dict['Rectified_Params1'][0]))
        for i in range(len(point_one)):
            cv.line(rectification, (int(point_one[i, 0]), int(point_one[i, 1])),
```

```python
                    (int(point_two[i, 0]) + self.image_specs[0][1], int(point_two[i, 1])),
                    color=(0, 0, 255), thickness=2)
        cv.imwrite('Rectification.jpg', rectification)
        self.parameter_dict['P_dash_value'] = P_dash_value
        print("Rectification complete")

    def rectify_image(self):
        """
        This function specifically applies the two homographies
        to the respective images which will effectively send their
        epipoles to infinity along the x-axis
        :return:
        """
        for index, element in enumerate(self.parameter_dict['H1&H2']):
            correlation = self.get_correlation(index,element)
            values = [[min(correlation[0]), min(correlation[1])],[max(correlation[0]), max(correlation[1])]]
            d_im = np.array(values[1]) - np.array(values[0])
            d_im = [int(d_im[0]), int(d_im[1])]
            scale = np.array([[self.image_specs[index][1] / d_im[0], 0, 0], [0, self.image_specs[index][0] / d_im[1], 0], [0, 0, 1]])
            print(element.shape)
            H = np.matmul(scale, element)
            correlation = self.get_correlation(index,H)
            values_2 = [min(correlation[0]), min(correlation[1])]
            d_im = values_2
            d_im = [int(d_im[0]), int(d_im[1])]
            T_value = np.array([[1, 0, -1 * d_im[0] + 1], [0, 1, -1 * d_im[1] + 1], [0, 0, 1]], dtype=float)
            homography_n = np.matmul(T_value, H)
            inverse_homography = np.linalg.pinv(homography_n)
            result_image = self.create_image(index=index,H=inverse_homography)
            self.parameter_dict['Rectified_Params'+str(index)] = [result_image, homography_n]

    def get_closest_match(self):
        """
        Function to find the nearest neighbor match to find the
        corresponding pixel in the second image.
        :return: rectified corners and the list of the nearest neighbors
        """
        corners_left = self.parameter_dict['corners_list_all'][0]
        corners_right = self.parameter_dict['corners_list_all'][1]
        image_left = self.image_pair[0]
        image_right = self.image_pair[1]
        ncc = np.zeros((len(corners_left), len(corners_right)))
```

```
        ncc = ncc - 2
        for row in range(len(corners_left)):
            print(str(row) + " out of "+str(len(corners_left)))
            for column in range(len(corners_right)):
                cor1 = corners_left[row]
                cor2 = corners_right[column]
                x_left_one = max(0, cor1[0] - self.padding)
                x_right_one = min(cor1[0] + self.padding + 1, image_left.shape[0])
                y_left_one = max(0, cor1[1] - self.padding)
                y_right_one = min(cor1[1] + self.padding + 1, image_left.shape[1])
                x_left_two = max(0, cor2[0] - self.padding)
                x_right_two = min(cor2[0] + self.padding + 1, image_right.shape[0])
                y_left_one = max(0, cor2[1] - self.padding)
                y_right_one = min(cor2[1] + self.padding + 1, image_right.shape[1])
                if x_right_one - x_left_one == x_right_two - x_left_two and y_right_one - y_left_one == y_right_one - y_left_one:
                    mean_value_one = np.mean(image_left[x_left_one:x_right_one, y_left_one:y_right_one])
                    mean_value_two = np.mean(image_right[x_left_two:x_right_two, y_left_one:y_right_one])
                    term_value_one = np.subtract(image_left[x_left_one:x_right_one, y_left_one:y_right_one], mean_value_one)
                    term_value_right = np.subtract(image_right[x_left_two:x_right_two, y_left_one:y_right_one], mean_value_two)
                    ncc[row, column] = np.divide(np.sum(np.multiply(term_value_one, term_value_right)),
                                                np.sqrt(
                                                    np.multiply(np.sum(np.square(term_value_one)), np.sum(np.square(term_value_right)))))

        rectify = list()
        neighbors = list()
        index_list = np.ones(len(corners_right))
        for row in range(len(ncc)):
            current_value = ncc[row]
            value = current_value[current_value >= -1]
```

```
                if len(value) > 0:
                    column = np.argmax(value)
                    if abs(corners_left[row][0] - corners_right[
                        column][0]) < 30 and abs(corners_left[row][1]
                        - corners_right[column][1]) < 60 and
                        index_list[
                         column] == 1:
                         if index_list[column] == 1 and max(value) >
                            0.6:
                             neighbors.append(max(value))
                             rectify.append([corners_left[row],
                                corners_right[column]])
                             index_list[column] = 0
        print('Returning closest match')
        self.parameter_dict['Correspondence']=rectify
        self.parameter_dict['neighbors'] = neighbors
        corresp_img = np.hstack((cv.imread('rectified_0.jpg'), cv
           .imread('rectified_1.jpg')))
        for i in range(len(rectify)):
            cv.line(corresp_img, (rectify[i][0][1], rectify[i
               ][0][0]),
              (rectify[i][1][1]+self.image_specs[0][1], rectify
                 [i][1][0]), color = (0,0,255), thickness = 1)
        cv.imwrite('corresps.jpg', corresp_img)

    def extract_corners(self):
        """
        Extract the corners from the images using Canny edge
           detector
        :return: Stores list of the corner coordinates.
        """
        edge_left = cv.Canny(self.grey_image_pair[0],255*1.5,255)
        edge_right = cv.Canny(self.grey_image_pair
           [1],255*1.5,255)
        edge_list = [edge_left,edge_right]
        temp_list = []
        for element in edge_list:
            for row in range(element.shape[0]):
                for column in range(element.shape[1]):
                    if column < 40 or column > 350 or row < 100:
                        element[row, column] = 0
            temp_list.append(element)
        edge_left = temp_list[0]
        edge_right =temp_list[1]
        cv.imwrite('edges_left_image.jpg',edge_left)
        cv.imwrite('edges_right_image.jpg', edge_right)
        corners = list()
        for image in self.grey_image_pair:
            corner_list = list()
            count=0
            for row in range(image.shape[0]):
                for column in range(image.shape[1]):
                    if image[row, column] != 0:
```

```
                        count+=1
                        if count % 12 == 0:
                            corner_list.append([row,column])
            corners.append(corner_list)
        self.parameter_dict['corners_list_all'] = corners
        self.image_res = np.hstack((self.image_pair[0], self.image_pair[1]))
        # for index,point in enumerate(corners[0]):
        #     ptx=(point[0],point[1])
        #     ptx2=(corners[1][index][0],corners[1][index][1])
        #     cv.line(self.image_res,ptx,ptx2,[255,255,0],3)

        print('Corner extraction complete')

    def create_image(self, index, H):
        """
        Create and store each of the rectified images
        :param index: Index of the image being considered
        :param H: Homography to project the new image
        :return: Rectified image
        """
        result_image = np.zeros((self.image_specs[index][0], self.image_specs[index][1], 3))
        for row in range(self.image_specs[index][0]):
            for column in range(self.image_specs[index][1]):
                temp_variable = np.matmul(H, np.array([[row],[column],[1]]))
                temp_variable = temp_variable/temp_variable[2]
                if temp_variable[0] >= 0 and temp_variable[0] < self.image_specs[index][0] and int(temp_variable[1]) >= 0 and int(temp_variable[1]) < self.image_specs[index][1]:
                    result_image[row, column] = self.image_pair[index][int(temp_variable[0]), int(temp_variable[1])]
        cv.imwrite('rectified_'+str(index)+'.jpg',result_image)
        return result_image

    def get_correlation(self, index, element):
        """
        Return the correlation needed to rectify the image
        :param index: Index of the image being considered
        :param element: Homography estimation
        :return: Return the correlation needed to rectify the image
        """
        correlation = np.matmul(element, np.array(
            [[0, self.image_specs[index][1], 0, self.image_specs[index][1]],
             [0, 0, self.image_specs[index][0], self.image_specs[index][0]], [1, 1, 1, 1]]))
        return (correlation / correlation[2])

```

```
    def triangulate_world_points(self, point_values, P_dash_value):
        """
        Function to estimate the triangulated world point coordinate
        using the corresponding pixel pairs from the left and right images
        :param point_values: Point pair
        :return: Triangulated world point coordinate
        """
        triangularted_world_points = list()
        P = self.parameter_dict['P_triangulation']
        for (point_one, point_two) in point_values:
            matrixA = np.array([point_one[0] * P[2] - P[0],
                                point_one[1] * P[2] - P[1],
                                point_two[0] * P_dash_value[2] - P_dash_value[0],
                                point_two[1] * P_dash_value[2] - P_dash_value[1]])
            u, d, v_t = np.linalg.svd(matrixA)
            v = v_t.T
            variable = v[:, -1]
            variable = variable/ variable[3]
            triangularted_world_points.append(variable)
        return triangularted_world_points

    def get_P_values(self, e_two, F ):
        """
        Function to estimate the metric needed to triangulate the
        world point
        """
        return np.append(np.matmul(
            np.array(
                [[0, -1 * e_two[2], e_two[1]], [e_two[2], 0, -1 * e_two[0]], [-1 * e_two[1], e_two[0], 0]]),
            F
        ),
        np.array([e_two[0], e_two[1], e_two[2]])
        ,
        axis=1)

    def get_updated_center(self, homography):
        """
        Get the updated centers of the images.
        :param homography: Homography
        :return: updated center
        """
        center = np.matmul(homography,self.reference_center.T)
        return center/center[2]

    def get_homography(self,e_one, e_two, type):
        """
        Calculate the homography to tranform or rectify the
```

```
            images.
        :param type: H1 for image one, H2 for image two
        :return: Homography estimation
        """
        if type == 'H2':
            theta_value = self.get_theta(e_one=e_one,e_two=e_two,image_index=0, type='2')
            F_value = (np.cos(theta_value)*(e_two[0]-self.image_specs[0][1]/2.0)-np.sin(theta_value)*(e_two[1]-self.image_specs[0][0]/2.0))[0]
            R_value = np.array([[np.cos(theta_value)[0], -1 * np.sin(theta_value)[0], 0], [np.sin(theta_value)[0], np.cos(theta_value)[0], 0], [0, 0, 1]])
            T_value = np.array([[1, 0, -1 * self.image_specs[0][1] / 2.0], [0, 1, -1 * self.image_specs[0][0] / 2.0], [0, 0, 1]])
            G_value = np.array([[1, 0, 0], [0, 1, 0], [-1.0 / F_value, 0, 1]])
            H = np.matmul(np.matmul(G_value,R_value),T_value)
            print('Returning H2')
            return H
        elif type == 'H1':
            theta_value = self.get_theta(e_one=e_one,e_two=e_two,image_index=0, type='1')
            F_value = (np.cos(theta_value) * (e_one[0] - self.image_specs[0][1] / 2.0) - np.sin(theta_value) * (e_one[1] - self.image_specs[0][0] / 2.0))[0]
            R_value = np.array([[np.cos(theta_value)[0], -1 * np.sin(theta_value)[0], 0], [np.sin(theta_value)[0], np.cos(theta_value)[0], 0], [0, 0, 1]])
            T_value = np.array([[1, 0, -1 * self.image_specs[0][1] / 2.0], [0, 1, -1 * self.image_specs[0][0] / 2.0], [0, 0, 1]])
            G_value = np.array([[1, 0, 0], [0, 1, 0], [-1.0 / F_value, 0, 1]])
            H = np.matmul(np.matmul(G_value,R_value),T_value)
            print('Returning H1')
            return H

    def get_theta(self, e_one, e_two, image_index, type):
        """
        Theta value to calculate the homography for stereo rectification
        :param image_index: Image index for the image under consideration
        :param type: 2 for second image and 1 for the first image
        :return: Theta value
        """
        if type == '2':
            return np.arctan(-1*(e_two[1]-self.image_specs[image_index][0]/2.0)/(e_two[0]-self.image_specs[image_index][1]/2.0))
        elif type == '1':
```

```
            return np.arctan(-1 * (e_one[1] - self.image_specs[image_index][0] / 2.0) / (
                        e_one[0] - self.image_specs[image_index][1] / 2.0))

    def get_nulls(self, F):
        e_one = null_space(F)
        e_two = null_space(F.T)
        e_one = e_one/e_one[2]
        e_two = e_two/e_two[2]
        return e_one, e_two

    def reinforce_F_estimate(self, F, T1,T2):
        """
        Make sure the F estimation has a rank of two.
        :param F: Initial F estimation
        :return: Reinforced F estimation
        """
        u,d,vt = np.linalg.svd(F)
        d=np.array([[d[0],0,0],[0,d[1],0],[0,0,0]])
        return np.matmul(np.matmul(T2.T, np.matmul(np.matmul(u, d), vt)), T1)

    def obtain_matrix_A(self,point_left, point_right):
        """
        Matrix to determine the first estimation of F
        :return: Matrix A
        """
        matrixA = list()
        for index in range(len(point_left)):
            value = [self.right_manual_points[index][0]*self.left_manual_points[index][0],
                     self.right_manual_points[index][0]*self.left_manual_points[index][1],
                     self.right_manual_points[index][0],self.right_manual_points[index][1]*self.left_manual_points[index][0],
                     self.right_manual_points[index][1]*self.left_manual_points[index][1],
                     self.right_manual_points[index][1],
                     self.left_manual_points[index][0],
                     self.right_manual_points[index][1],
                     1.0]
            matrixA.append(value)
        return matrixA

    def process_points(self):
        """
        Segregate the interest points into two unique list each for one of
        the two images. The coordinates of the second image are updated to
        account for the offset along the width.
```

```
        :return: Store the left and right interest points
        """
        for element_index in tqdm(range(len(self.roiCoordinates)),desc='Point process'):
            if self.roiCoordinates[element_index][0]>self.image_specs[0][1]:
                point = (self.roiCoordinates[element_index][0]-self.image_specs[0][1],self.roiCoordinates[element_index][1])
                self.right_manual_points.append(point)
            else:
                self.left_manual_points.append(self.roiCoordinates[element_index])
        pickle.dump(self.left_manual_points,open('left_manual_points.obj','wb'))
        pickle.dump(self.right_manual_points, open('right_manual_points.obj', 'wb'))
        print('Selected points are:')
        left_points = list(map(lambda x: [x[1], x[0]], self.left_manual_points))
        right_points = list(map(lambda x: [x[1], x[0]], self.right_manual_points))
        print(left_points)
        print(right_points)

    def append_points(self, event, x, y, flags, param):
        """
        [This function is called every time the mouse left button is clicked - It records the (x,y) coordinates of the click location]

        """
        lit = ['a','a','b','a','c','a','d','a','e','a','f','a','g','a','h','a','i','a']
        if event == cv.EVENT_LBUTTONDOWN:
            self.roiCoordinates.append((int(x), int(y)))
            if self.count % 2 == 0:
                cv.circle(self.image,(int(x), int(y)),5,[0,255,255],3)
                cv.putText(self.image,lit[self.count],(int(x+3), int(y-2)),cv.FONT_HERSHEY_COMPLEX,
                           1,[0,255,255])
                # cv.putText(self.image, str(self.count), cv.Point(30, 30),
                #         cv.FONT_HERSHEY_COMPLEX_SMALL, 0.8, cv.Scalar(200, 200, 250), 1)
            else:
                cv.circle(self.image, (int(x), int(y)), 5, [0, 255, 255], 3)
                # cv.putText(self.image,lit[self.count],(int(x+3), int(y-2)),cv.FONT_HERSHEY_COMPLEX,
                #            1,[0,255,255])
                # cv.putText(self.image, str(self.count), cv.
```

```
                    Point(30, 30),
                #               cv.FONT_HERSHEY_COMPLEX_SMALL, 0.8, cv.Scalar(200, 200, 250), 1)
                cv.line(self.image,self.roiCoordinates[self.count-1],(int(x), int(y)),[0,255,0],3)
            self.count +=1

    def append_points_3d(self, event, x, y, flags, param):
        """
        [This function is called every time the mouse left button is clicked - It records the (x,y) coordinates of the click location]

        """
        print('3D point')
        if event == cv.EVENT_LBUTTONDOWN:
            self.roiCoordinates_3d.append((int(x), int(y)))
            if self.count % 2 == 0:
                cv.circle(self.image,(int(x), int(y)),5,[0,255,255],3)
            else:
                cv.circle(self.image, (int(x), int(y)), 5, [0, 255, 255], 3)
                cv.line(self.image,self.roiCoordinates_3d[self.count-1],(int(x), int(y)),[0,255,0],3)
            self.count +=1

    def getROIFromUser(self, type = 'No', type2='No'):
        """
        [This function is responsible for taking the regions of interests from the user]

        """
        if type == 'yes':
            self.roiCoordinates = pickle.load(open('points.obj','rb'))
        elif type =='No':
            self.roiList = []
            cv.namedWindow('Select ROI')
            cv.setMouseCallback('Select ROI', self.append_points)
            self.image = np.hstack((self.image_pair[0],self.image_pair[1]))
            while(True):
                cv.imshow('Select ROI', self.image)
                k = cv.waitKey(1) & 0xFF
                if cv.waitKey(1) & 0xFF == ord('q'):
                    break
            pickle.dump(self.roiCoordinates,open('points.obj','wb'))
            cv.imwrite('result_1.jpg',self.image)

        # if type2 == 'yes':
        #     self.roiCoordinates_3d = pickle.load(open('points3d
```

```
            .obj','rb'))
        # elif type2 == 'No':
        #     print('here')
        #     cv.namedWindow('3D ROI')
        #     cv.setMouseCallback('3D ROI', self.append_points_3d)
        #     self.image_3d = np.hstack((self.image_pair[0],self.image_pair[1]))
        #     while(True):
        #         cv.imshow('Select ROI', self.image_3d)
        #         k = cv.waitKey(1) & 0xFF
        #         if cv.waitKey(1) & 0xFF == ord('q'):
        #             break
        #     pickle.dump(self.roiCoordinates_3d, open('points3d.obj', 'wb'))
        #     cv.imwrite('3d_points.jpg', self.image)

    def get_sliding_window(self, image, column, row, left = 2, right = 3):
        """
        Get the sliding window to estimate the bitvector
        for each pixel.
        :param image: Image under consideration
        :param column: column value of the pixel coordinate
        :param row: row value of the pixel coordinate
        :param left: Left padding
        :param right: Right padding
        :return: Window image
        """
        return image[column-left:column+right, row-left:row+right]

    def get_bit_vector(self,M_value, image, column, row):
        """
        Function to estimate the bit vector for the given window over
        the given center pixel
        :param image: Image under consideration
        :param column: Column value of the pixel coordinate
        :param row: Row value of the pixel coordinate
        :return: Bitvector in list type
        """
        bitvector = list()
        slidingwindow = self.get_sliding_window(image=image, column=column, row=row)
        for row_slide in range(M_value):
            for column_slide in range(M_value):
                if slidingwindow[row_slide, column_slide] > image[column,row]:
                    bitvector.append(1)
                elif slidingwindow[row_slide, column_slide] <= image[column,row]:
                    bitvector.append(0)
```

```
        bitvector = np.asarray(bitvector)
        return bitvector

    def estimate_error_mask(self):
        """
        Estimate the error mask
        :return:
        """
        result_disparity_map = self.parameter_dict['Disparity Map
            ']
        mask = self.occ_grey[1]
        mask_E = np.zeros((result_disparity_map.shape))
        total = 0
        count_correct = 0
        for row in range(result_disparity_map.shape[1]):
            for column in range(result_disparity_map.shape[0]):
                if mask[column, row] == 0:
                    pass
                total = total + 1
                if np.absolute(result_disparity_map[column, row]
                    - self.occ_grey[0][column, row]) <= 1.5:
                    count_correct = count_correct + 1
                    mask_E[column, row] = 255
        cv.imwrite("error_mask.jpg", mask_E)
        print('Accuracy is', count_correct / total * 100)
        print('Completed error mask estimation')

    def estimate_disparity_maps(self, M_value, Max_D):
        """
        Create and save the disparity map.
        :param M_value: M value
        :param Max_D: Dmax value
        :return:
        """
        result_disparity_map = np.zeros((self.image_specs[0]))
        for row in range(self.image_specs[0][1]):
            for column in range(self.image_specs[0][0]):
                bitvector_left = self.get_bit_vector(M_value=
                    M_value,image = self.grey_image_pair[0],
                    column=column, row=row)
                lower_cutoff = 255
                for d_value in range(Max_D):
                    if row - d_value < 0:
                        break
                    right_bv = self.get_bit_vector(M_value=
                        M_value,image = self.grey_image_pair[1],
                        column=column, row = row - d_value)
                    value = (np.bitwise_xor(bitvector_left,
                        right_bv)).sum(-1)
                    if value < lower_cutoff:
                        final_d = d_value
                        lower_cutoff = value
                result_disparity_map[column, row] = final_d
```

```
        cv.imwrite("disparity_map.jpg", result_disparity_map)
        self.parameter_dict['Disparity Map'] =
           result_disparity_map
        print('Disparity map created and saved.')

    def cost(self, F, c_one, c_two):
        """
        Cost function for the LM optimization
        :param F: Initial F estimation
        :param c_one: Point list one
        :param c_two: Point list two
        :return: Cost estimate
        """
        F = F.flatten()
        matches = list()
        F = np.append(F, 1).reshape(3, 3)
        e_two = null_space(F.T)
        e_two =e_two/ e_two[2]
        e_two_1 = np.array([[0, -1 * e_two[2], e_two[1]], [e_two
           [2], 0, -1 * e_two[0]],
                          [-1 * e_two[1], e_two[0], 0]])
        P_value_two = np.matmul(e_two_1, F)
        P_value_two = np.append(P_value_two, np.array([e_two[0],
           e_two[1], e_two[2]]), axis=1)
        for point_index in range(len(c_one[0])):
            matches.append([[c_one[0, point_index], c_one[1,
               point_index]], [c_two[0, point_index], c_two[1,
               point_index]]])
        world_pts = self.triangulate_world_points(matches,
           P_value_two)
        world_pts = np.array(world_pts).T
        P_value_one = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0,
           0, 1, 0]])
        projection_one = np.matmul(P_value_one, world_pts)
        projection_one = projection_one/projection_one[2]
        projection_two = np.matmul(P_value_two, world_pts)
        projection_two =projection_two / projection_two[2]
        cost = np.append(projection_one[:2] - c_one[:2],
           projection_two[:2] - c_two[:2])
        return cost

    def condition_F(self,F):
        F = F.flatten()
        F = F / F[-1]
        F = F[:-1]
        F = F.reshape(8, 1)
        return F

    def levenberg_marquardt_optimization(self):
        """
        Function to improve our F estimation using LM
           optimization
        :return:
```

```
        """
        c_one,c_two = list(),list()
        for i in range(len(self.parameter_dict['Correspondence'])):
            c_one.append([self.parameter_dict['Correspondence'][i][0][0], self.parameter_dict['Correspondence'][i][0][1], 1])
            c_two.append([self.parameter_dict['Correspondence'][i][1][0], self.parameter_dict['Correspondence'][i][1][1], 1])
        c_one = np.array(c_one).T
        c_two = np.array(c_two).T
        F = self.condition_F(self.parameter_dict['F_beta'])
        updated_value = optimize.leastsq(self.cost, F, args=(c_one, c_two))
        F = np.append(updated_value[0], 1).reshape(3, 3)
        e_two = null_space(F.T)
        e_two =e_two/ e_two[2]
        e_two_1 = np.array(
            [[0, -1 * e_two[2], e_two[1]], [e_two[2], 0, -1 * e_two[0]],
             [-1 * e_two[1], e_two[0], 0]])
        P_dash_updated = np.matmul(e_two_1, F)
        P_dash_updated = np.append(P_dash_updated, np.array([e_two[0], e_two[1], e_two[2]]), axis=1)
        self.parameter_dict['F_updated'] = F
        self.parameter_dict['P_dash_updated'] = P_dash_updated
        print('Optimization complete')

if __name__ == "__main__":
    """
    Code starts here

    """
    tester = Reconstruct(['Task2_Images/Left.jpg','Task2_Images/Right.jpg'],['Task2_Images/left_truedisp.pgm','Task2_Images/mask0nocc.png'])
    tester.schedule()
    print('Task 1 complete')
    tester.estimate_disparity_maps(M_value=3,Max_D=5)
    tester.estimate_error_mask()
    print('Task 2 complete')
```