Stereo Vision

By Arvinder Singh and Uday Raghuvanshi

ABSTRACT

This project aims to explore stereo vision by using two images taken from different perspectives to extract 3D information about a scene. The project involves detecting and matching features between the two images, estimating the Fundamental Matrix using RANSAC to eliminate outliers, computing a dense disparity map using Fundamental matrix to help reduce the search space, and writing a report discussing the results and providing sample images to illustrate the findings.

For the project, we have used python as the programming language and used modules including cv2, numpy and os.

DESCRIPTION OF ALGORITHM

In order to implement Stereo Vision, we started by forming a StereoVision class which has several methods to perform the steps involved in obtaining a dense disparity map. The instance of the Stereo Vision class takes 'array_of_images' as an array of grayscale images, 'type_of_derivative_filter' to choose from sobel or prewitt derivative filter, 'threshold' for extracting corners, 'hc_window_size' as the size of window used to obtain r_score for harris corners, 'ncc_window' as the window size of the template used while performing Normalized Cross-Correlation and 'ransac iterations' as the number of times we perform ransac to get the best Fundamental matrix as inputs.

The **StereoVision** class is **initialized** as follows:

```python
class StereoVision:
    def __init__(self, array_of_images: list, type_of_derivative_filter="sobel", hc_window_size=(5, 5),
                 ncc_threshold=100000, ncc_window=(7, 7), ransac_iterations=10):
        self.array_of_images = array_of_images
        self.type_of_derivative_filter = type_of_derivative_filter
        self.window_size = hc_window_size
        self.threshold = ncc_threshold
        self.ncc_window = ncc_window
        self.iterations = ransac_iterations

    def derivative(self):
        i_x = []
        i_y = []
        for image in self.array_of_images:
            image = cv2.boxFilter(image, -1, (3, 3))
            if self.type_of_derivative_filter.lower() == "sobel":
                sobel_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                sobel_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_y))
            elif self.type_of_derivative_filter.lower() == "prewitt":
                prewitt_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                prewitt_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_y))
            else:
                print("Incorrect input")
                return
        return i_x, i_y
```

Fig: **StereoVision** *class initialization*

The description of each method of our algorithm is as follows:

1. derivative: We start by computing the derivative of the two images. The 'derivative' method reads all the gray scale images in the 'array_of_images' (two in our case), type of derivative filter- 'prewitt' or 'sobel' and finally returns their derivatives in x and y direction as i_x and i_y arrays.

```
def derivative(self):
    i_x = []
    i_y = []
    for image in self.array_of_images:
        image = cv2.boxFilter(image, -1, (3, 3))
        if self.type_of_derivative_filter.lower() == "sobel":
            sobel_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
            sobel_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
            i_x.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_x))
            i_y.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_y))
        elif self.type_of_derivative_filter.lower() == "prewitt":
            prewitt_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
            prewitt_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
            i_x.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_x))
            i_y.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_y))
        else:
            print("Incorrect input")
            return
    return i_x, i_y
```

Fig: derivative method

2. harris_corner_detector: This method takes the derivatives i_x and i_y, forms a window for corner detection according to the hc_window_size. For each pixel, it iterates over all the neighbouring pixels in the window to obtain the r_score. Then it thresholds the r_score value to keep only relevant corner features, setting others to 0. We use the value of k (empirically determined constant as 0.06). This returns the array of of matrices of same size as image containing the r_scores

```
def harris_corner_detector(self, k=0.06):
    i_x, i_y = self.derivative()
    harris_r_array = []
    for ind in range(len(i_x)):
        fx = i_x[ind]
        fy = i_y[ind]
        harris_r = np.zeros(np.shape(fx))
        rows, columns = np.shape(fx)[0], np.shape(fx)[1]
        for row in range(rows):
            for column in range(columns):
                i = row - self.window_size[0] // 2
                j = column - self.window_size[0] // 2
                i_x_2 = 0
                i_y_2 = 0
                i_x_y = 0
                while i <= row + self.window_size[0] // 2:
                    while j <= column + self.window_size[0] // 2:
                        if 0 < i < rows and 0 < j < columns:
                            i_x_2 += np.square(fx[i][j])
                            i_y_2 += np.square(fy[i][j])
                            i_x_y += fx[i][j] * fy[i][j]
                        j += 1
                    i += 1
                m = np.array([[i_x_2, i_x_y], [i_x_y, i_y_2]])
                r_score = np.linalg.det(m) - k * (np.square(m[0][0] + m[1][1]))
                if self.threshold < r_score:
                    harris_r[row][column] = r_score
        harris_r_array.append(harris_r)
    return harris_r_array
```

Fig: harris_corner_detector

3. non_max_suppression: We use this function to obtain a sparse set of harris corners. To perform NMS, we split each matrix returned by harris_corner_detector into tiles of size 64x64 pixels. For each tile, we only keep those r_score values which are maximum in a window of 5x5, keeping others as 0. So, we essentially have one corner in a window of 5x5. Then we sort all the r_score values in the tile and keep only the 10 largest values. Thus, each tile gives us at most 10 corner features. The value of window and the total number of corners to be retained in a tile can be changed. This returns an array of matrices (size 2 in our case) with a sparse set of corner features.

```
def non_max_supression(self):
    harris_r_array = self.harris_corner_detector()
    nms_harris_arr = []
    pix_dist = 5
    h, w = np.shape(harris_r_array[0])
    for ind in harris_r_array:
        nms_harris = np.zeros((h, w))
        for i in range(0, h, 64):
            for j in range(0, w, 64):
                r_array = []
                for m in range(i + pix_dist, i + 64 - pix_dist):
                    for n in range(j + pix_dist, j + 64 - pix_dist):
                        if m < h and n < w:
                            if ind[m, n] > 0 and ind[m, n] == np.max(
                                    ind[m - pix_dist:m + pix_dist + 1, n - pix_dist:n + pix_dist + 1]):
                                r_array.append((ind[m, n], m, n))
                if len(r_array) < 10:
                    for p in range(len(r_array)):
                        nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]
                else:
                    r_array = sorted(r_array, reverse=True)
                    for p in range(10):
                        nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]

        nms_harris_arr.append(nms_harris)
    return nms_harris_arr
```

Fig: non_max_suppression

4. disp_img: This method takes in the two rgb images where we detected corners and overlays the corner points detected with harris_detector or non_max_suppresion.

```
def disp_img(self, img1, img2):
    nms_harris = self.non_max_supression()
    rgb_imgs_arr = [img1, img2]
    corner_imgs = []
    for k in range(len(nms_harris)):
        rgb = np.copy(rgb_imgs_arr[k])
        r, c = np.shape(rgb)[0], np.shape(rgb)[1]
        for i in range(r):
            for j in range(c):
                if nms_harris[k][i, j] > 0:
                    rgb[i, j] = [0, 0, 255]
        corner_imgs.append(rgb)
    return corner_imgs
```

Fig: disp_image

5. find_correspondences: This method takes as input the sparse set of corner features of the two images and finds correspondences between them using Normalized Cross-Correlation (NCC). For one image, it forms a window of size 'ncc_window' and iterates over the image looking for potential corner features (corners_0[row][column]>0). Upon encountering a corner, it forms a template of size ncc_window around it. Then it slides that template in the second image, calculating the NCC scores for all corners in the second image. If the NCC score is above 0.90, it appends the location of pixels in the two images in two lists, correspondences_picture0 and correspondences_picture1. Finally, it returns us the corresponding features in the two images.

```
def find_correspondences(self):
    corners_array = self.non_max_supression()
    corners_0 = corners_array[0]
    corners_1 = corners_array[1]
    rows, columns = np.shape(corners_0)[0], np.shape(corners_0)[1]
    correspondences_picture0 = []
    correspondences_picture1 = []
    for row in range(rows):
        for column in range(columns):
            if corners_0[row][column] > 0:
                a = row - (self.ncc_window[0] // 2)
                b = row + (self.ncc_window[0] // 2) + 1
                c = column - (self.ncc_window[0] // 2)
                d = column + (self.ncc_window[0] // 2) + 1
                if 0 < a < rows - self.ncc_window[0] and 0 < c < columns - self.ncc_window[0]:
                    template = np.array(self.array_of_images[0][a:b, c:d])
                    max_ncc = 0
                    max_row = None
                    max_column = None
                    for row2 in range(rows):
                        for column2 in range(columns):
                            if corners_1[row2][column2] > 0:
                                a2 = row2 - (self.ncc_window[0] // 2)
                                b2 = row2 + (self.ncc_window[0] // 2) + 1
                                c2 = column2 - (self.ncc_window[0] // 2)
                                d2 = column2 + (self.ncc_window[0] // 2) + 1
                                if 0 < a2 < rows - self.ncc_window[0] and 0 < c2 < columns - self.ncc_window[0]:
                                    match_to = np.array(self.array_of_images[1][a2:b2, c2:d2])
                                    f = (match_to - match_to.mean())/(match_to.std() * np.sqrt(match_to.size))
                                    g = (template - template.mean())/(template.std() * np.sqrt(template.size))
                                    product = f * g
                                    stds = np.sum(product)
                                    if stds > max_ncc:
                                        max_ncc = stds
                                        max_row = row2
                                        max_column = column2
                    if max_row is not None:
                        if max_ncc > 0.90:
                            correspondences_picture0.append([row, column])
                            correspondences_picture1.append([max_row, max_column])
    return correspondences_picture0, correspondences_picture1
```

Fig: find_correspondences

6. find_fundamental_matrix: This method is used to estimate the fundamental matrix using correspondences. This method takes as input the corresponding points on the left and right images. Then it converts them into homogenous coordinates by adding 1 to them. It also normalizes the pixel coordinates before forming the a_matrix using the pixel correspondences. After forming the a_matrix, it performs Singular-Value Decomposition on it, extracts the fundamental matrix (f) as the last column of V (or last row of V_transpose), reshapes it and enforces a rank 2 constraint on it. Finally, it denormalizes the fundamental matrix f and returns it.

```
def find_fundamental_matrix(pts_src, pts_dst):
    pts_src=np.array(pts_src)
    pts_dst = np.array(pts_dst)
    ones_arr_src=np.ones((pts_src.shape[0],1))
    ones_arr_dst=np.ones((pts_dst.shape[0],1))
    pts_src=np.hstack((pts_src,ones_arr_src))
    pts_dst=np.hstack((pts_dst,ones_arr_dst))
    mean_src=np.mean(pts_src[:,:2],axis=0)
    S_src=np.sqrt(2)/np.std(pts_src[:,:2])
    T_src=np.array([[S_src, 0, -S_src*mean_src[0]],[0,S_src,-S_src*mean_src[1]],[0,0,1]])
    pts_src=np.dot(T_src,np.transpose(pts_src))
    pts_src=np.transpose(pts_src)
    mean_dst = np.mean(pts_dst[:, :2], axis=0)
    S_dst = np.sqrt(2)/ np.std(pts_dst[:, :2])
    T_dst = np.array([[S_dst, 0, -S_dst * mean_dst[0]], [0, S_dst, -S_dst * mean_dst[1]], [0, 0, 1]])
    pts_dst=np.dot(T_dst,np.transpose(pts_dst))
    pts_dst=np.transpose(pts_dst)
    # forming A matrix
    a_matrix = np.zeros((len(pts_src), 9))
    for i in range(len(pts_src)):
        a_matrix[i, 0] = pts_src[i][1] * pts_dst[i][1]
        a_matrix[i, 1] = pts_src[i][1] * pts_dst[i][0]
        a_matrix[i, 2] = pts_src[i][1]
        a_matrix[i, 3] = pts_src[i][0] * pts_dst[i][1]
        a_matrix[i, 4] = pts_src[i][0] * pts_dst[i][0]
        a_matrix[i, 5] = pts_src[i][0]
        a_matrix[i, 6] = pts_dst[i][1]
        a_matrix[i, 7] = pts_dst[i][0]
        a_matrix[i, 8] = 1
    u, d, v_transpose = np.linalg.svd(a_matrix, full_matrices=True)
    f = np.transpose(v_transpose[-1].reshape(3, 3))
    u_f, d_f, v_f_transpose = np.linalg.svd(f, full_matrices=True)
    d_f[-1] = 0
    f = np.dot(u_f,np.dot(np.diag(d_f), v_f_transpose))
    f=np.dot(np.transpose(T_dst),np.dot(f,T_src))
    # f = f / f[2, 2]
    return f
```

Fig: find_fundamental_matrix

7. ransac: To have the best estimate of our fundamental matrix, we need to eliminate outliers. The correspondences we found using find_correspondences have many outliers. So, we implement ransac. This method takes the correspondences we found and randomly samples 8 correspondences. It then computes a fundamental matrix from these 8 and checks for the inliers using Longuet Higgins equation

(pr_t*fundamental_matrix*pl=0). Since we know that the observed points in the right image will lie on the epipolar line in left image if it's an inlier, we compute the estimated distance between the point on right and the corresponding epipolar line on left and if the distance is less than 0.007, we keep the corresponding points as inliers. We keep doing this for all the iterations and the fundamental matrix that gives us the highest number of inliers is the one we keep. As a last step, we find the fundamental matrix using the set of all inliers to attain the best fundamental matrix.

```
def ransac(self):
    corres_0, corres_1 = self.find_correspondences()
    max_inliers = 0
    fundamental_matrix_with_all = None
    max_inliers_0 = []
    max_inliers_1 = []
    for it in range(self.iterations):
        random_sampled_indx = []
        pts_src = []
        pts_dst = []
        i = 0
        while i < 8:
            random_sample = int(np.random.randint(0, high=len(corres_0) - 1, size=1, dtype=int))
            if random_sample not in random_sampled_indx:
                pts_src.append(corres_0[random_sample])
                pts_dst.append(corres_1[random_sample])
                random_sampled_indx.append(random_sample)
                i += 1
        fundamental_matrix = find_fundamental_matrix(pts_src, pts_dst)
        j = 0
        inliers = 0
        inliers_0 = []
        inliers_1 = []
        while j < len(corres_0):
            pl = np.transpose(np.array([[corres_0[j][1], corres_0[j][0], 1]]))
            pr_t = np.array([[corres_1[j][1], corres_1[j][0], 1]])
            est = np.dot(pr_t, np.dot(fundamental_matrix, pl))
            # print(est)
            if abs(est) < 0.007:
                inliers += 1
                inliers_0.append(corres_0[j])
                inliers_1.append(corres_1[j])
            j += 1
        if inliers > max_inliers:
            max_inliers = inliers
            max_inliers_0 = inliers_0
            max_inliers_1 = inliers_1
            if max_inliers > 7:
                fundamental_matrix_with_all = find_fundamental_matrix(max_inliers_0, max_inliers_1)
    return fundamental_matrix_with_all, max_inliers_0, max_inliers_1
```

Fig: ransac

8. feature_matching: This method takes as input the two images whose features need to be matched and the correspondences found in find_correspondences, concatenates the two images and draws lines joining the two images. It returns an image showing potential corner feature location matches between the two images.

```
@staticmethod
def feature_matching(img_l, img_r, corres_0, corres_1):
    img_l = np.copy(img_l)
    img_r = np.copy(img_r)
    shift = np.shape(img_r)[1]
    full_img = np.concatenate((img_l, img_r), axis=1)
    for i in range(len(corres_0)):
        color = list(np.random.random_sample(size=3) * 256)
        cv2.line(full_img, (corres_0[i][1], corres_0[i][0]), (corres_1[i][1] + shift, corres_1[i][0]), color,
                 thickness=1)
    return full_img
```

Fig: feature_matching

9. draw_epilines: It takes as input the output from RANSAC (fundamental matrix, correspondences on the left and correspondences on the right) and a list of rgb_images, then we compute the epilines for the right and for the left image using the correspondences we found from RANSAC and the fundamental matrix and draw them on the image using cv2.line and return the images with epipolar lines.

```
def draw_epilines(self, ransac_output, rgb_img_list):
    left_image = rgb_img_list[0]
    right_image = rgb_img_list[1]
    img_shape = self.array_of_images[0].shape
    fundamental_matrix = ransac_output[0]
    fundamental_matrix = fundamental_matrix/fundamental_matrix[2,2]

    inliers_left = np.array(ransac_output[1])
    inliers_right = np.array(ransac_output[2])
    columns = img_shape[1]

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

        pt_left = np.array([[inliers_left[i, 1]], [inliers_left[i, 0]], [1]])
        a_b_c_right = np.matmul(fundamental_matrix, pt_left)
        slope_right = - a_b_c_right[0] / a_b_c_right[1]
        intercept_right = - a_b_c_right[2] / a_b_c_right[1]
        column_vec_right = np.arange(columns)
        row_vec_right = column_vec_right * int(slope_right) + int(intercept_right)
        cv2.line(right_image, (column_vec_right[0], row_vec_right[0]), (column_vec_right[-1], row_vec_right[-1]), (0, 0, 255), thickness=1)

        pt_right = np.array([[inliers_right[i, 1]], [inliers_right[i, 0]], [1]])
        a_b_c_left = np.matmul(fundamental_matrix.T, pt_right)
        slope_left = - a_b_c_left[0] / a_b_c_left[1]
        intercept_left = - a_b_c_left[2] / a_b_c_left[1]
        column_vec_left = np.arange(columns)
        row_vec_left = column_vec_left * int(slope_left) + int(intercept_left)
        cv2.line(left_image, (column_vec_left[0], row_vec_left[0]), (column_vec_left[-1], row_vec_left[-1]), (0, 0, 255), thickness=1)

    return left_image, right_image
```

Fig: draw_epilines

10. Disparity_map: It takes as input an array of images (having left and right images) along with the fundamental matrix and returns 3 images, horizontal disparity, vertical disparity, and the colored disparity display. The aim is to form a window at every pixel on left image and look for a matching template along the epipolar line in the right image. Similarity between templates is calculated using the Sum of Absolute Differences (SADs) between them. The column for which it gets minimum SAD is taken as the best matching column. Rather than looking for the match on the entire epipolar line, it only looks until a maximum disparity range of 50 pixels. This reduces the search space as well as the computation time. As a first step, it calculates the a,b and c values for epipolar lines in the right image corresponding to every pixel in the left image and stores it in an epi_lines_right matrix. Next, it iterates over all the pixels in left image, and forms a window of size 7 by 7 around each. For each pixel in left image, it looks for the best

matching pixel along the epipolar line in the right image over a range of 50 pixels. The horizontal disparity is calculated by subtracting the difference of the current column in left image and best matching column in the right image from the current column in left image. Similarly, the vertical disparity is calculated using current row and best match row.

The disparity values are then scaled from a range of 0 to 255 where 0 shows least disparity and 255 shows highest disparity. The third image represents the disparity vector using color. For each pixel, the disparity vector is calculated using the horizontal and vertical disparities. It is then normalized in the range of 0 to 1. Then it is converted into hue and saturation values.

```
def disparity_map(array_of_images, fund_matrix):
    left_img=array_of_images[0]
    right_img=array_of_images[1]
    rows,cols=left_img.shape[0],left_img.shape[1]
    window_size=7
    block=window_size//2
    hori_disp_img=np.zeros_like(left_img,np.uint8)
    vert_disp_img=np.zeros_like(left_img,np.uint8)
    disp_color=np.zeros_like(left_img, np.uint8)
    epi_lines_right=np.zeros((rows,cols,3))
    max_disp=50
    for i in range(rows):
        for j in range(cols):
            pt_l=np.transpose(np.array([[j,i,1]]))
            l_r=np.dot(np.transpose(fund_matrix),pt_l)
            epi_lines_right[i, j, :] =l_r.reshape(-1)
    for row in range(block,rows-block):
        for col in range(block,cols-block):
            a, b, c = epi_lines_right[row,col]
            left_win=left_img[row-block:row+block+1,col-block:col+block+1]
            max_similarity = float('inf')
            best_match_col=-1
            best_match_row=-1
            for k in range(max_disp):
                y_r = int((-a * (col+k) - c) / b)
                right_win=right_img[y_r-block:y_r+block+1,col-block+k:col+block+1+k]
                if left_win.shape == right_win.shape:
                    sad = np.sum(abs(left_win - right_win))
                    if sad < max_similarity:
                        max_similarity = sad
```

Fig: Disparity map (i)

```
                if sad < max_similarity:
                    max_similarity = sad
                    best_match_col = k
                    best_match_row = y_r
        hori_disp=col-(col-best_match_col)
        vert_disp=row-(row-best_match_row)
        hori_disp = np.clip(hori_disp, 0, 255)
        vert_disp = np.clip(vert_disp, 0, 255)
        hori_disp_img[row, col] =hori_disp
        vert_disp_img[row,col]=vert_disp
        disp_color[row,col]=np.sqrt(np.sum((hori_disp**2,vert_disp**2)))
disp_color=cv2.normalize(hori_disp_img, None, 0, 1, cv2.NORM_MINMAX)
hue = disp_color*0.5
saturation = np.ones_like(disp_color,dtype=np.float32)
disparity_hsv = np.stack((hue, saturation, saturation), axis=-1)
disparity_hsv=(255*disparity_hsv).astype(np.uint8)
disparity_hsv = cv2.cvtColor(disparity_hsv, cv2.COLOR_HSV2BGR)
hori_disp_img=cv2.normalize(hori_disp_img,None, 0, 255, cv2.NORM_MINMAX)
vert_disp_img= cv2.normalize(vert_disp_img, None, 0, 255, cv2.NORM_MINMAX)
return hori_disp_img,vert_disp_img,disparity_hsv
```

Fig: Disparity map (ii)

Results:-

1. ***Finding correspondences:-*** We tried computing Harris corners using sobel derivative filters,, non-max suppression and thresholding at different values of normalized cross-correlation.

Before RANSAC

Below are the correspondences before RANSAC:-

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.9

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

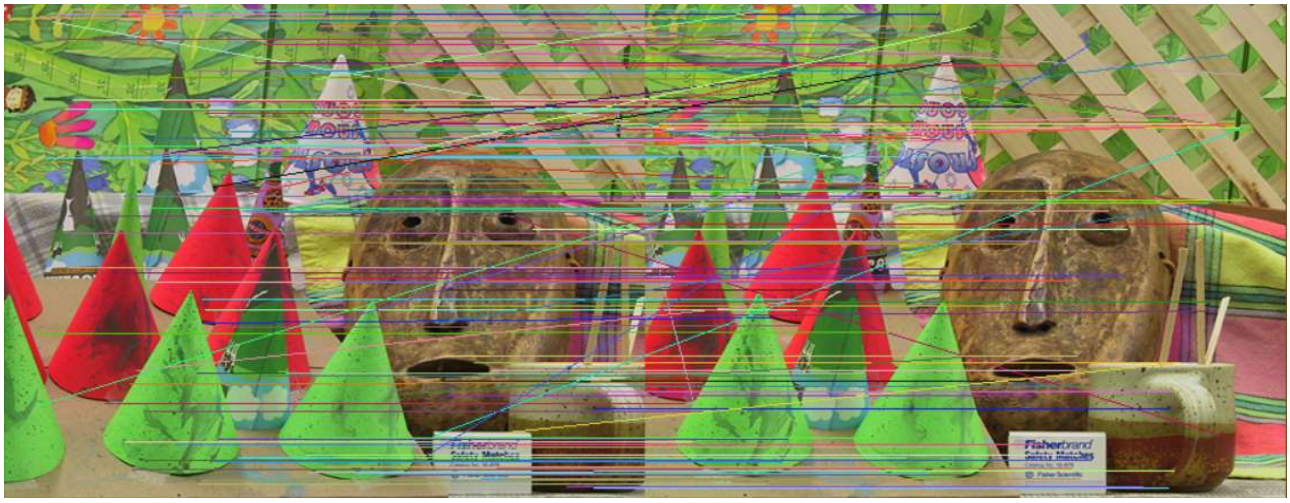

Fig: Correspondences before RANSAC(1)

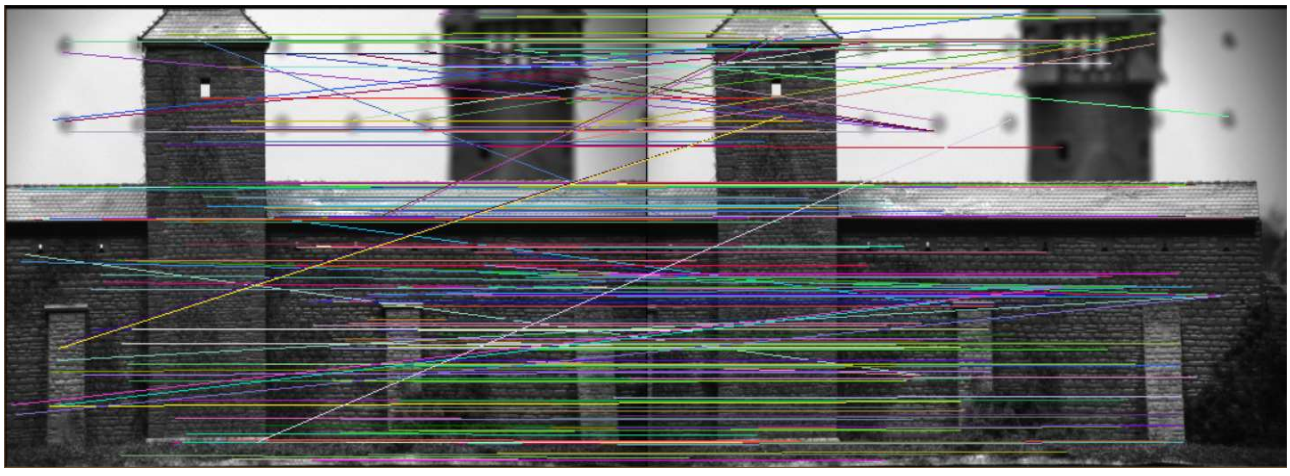

Fig: Correspondences before RANSAC(2)

After RANSAC

After getting the point correspondences, we now run RANSAC on these points to get rid of outliers. We sample 8 random points and then calculate the fundamental matrix using those 8 points. Then we run RANSAC to compute the number of inliers and the outliers. Whatever fundamental matrix gives us the maximum number of inliers we then finally use those inliers to calculate the final fundamental matrix using SVD.

Below are the correspondences after RANSAC:-

derivative_filter = sobel

hc_window_size = (5, 5)

r_score_threshold = 100000

ncc_value = 0.9

ncc_window = (7, 7)

k = 0.06

nms_threshold = 5

RANSAC iterations = 15000

Fig: Correspondences after RANSAC(1)

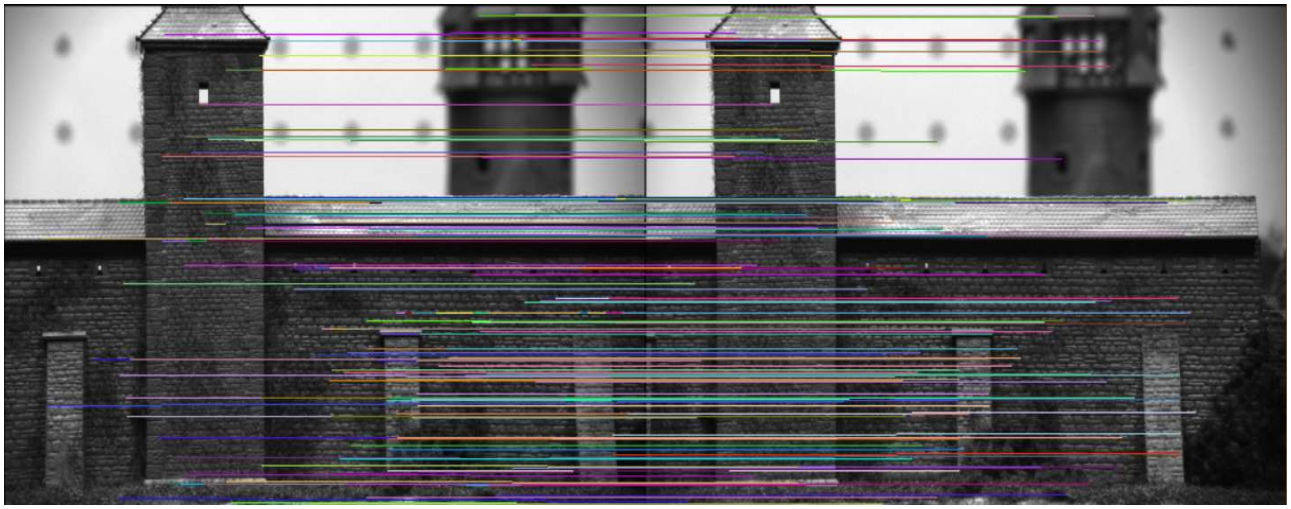

Fig: Correspondences after RANSAC(2)

2. **Epipolar lines-** After getting the final fundamental matrix, we use that fundamental matrix to plot and see the epipolar lines in the right and the left image. We will use these lines to reduce the search space as well.

Below we can observe the epiline on the left and right image for both the datasets. We are using all the inliers that we got after RANSAC to plot epilines on the left and right image.

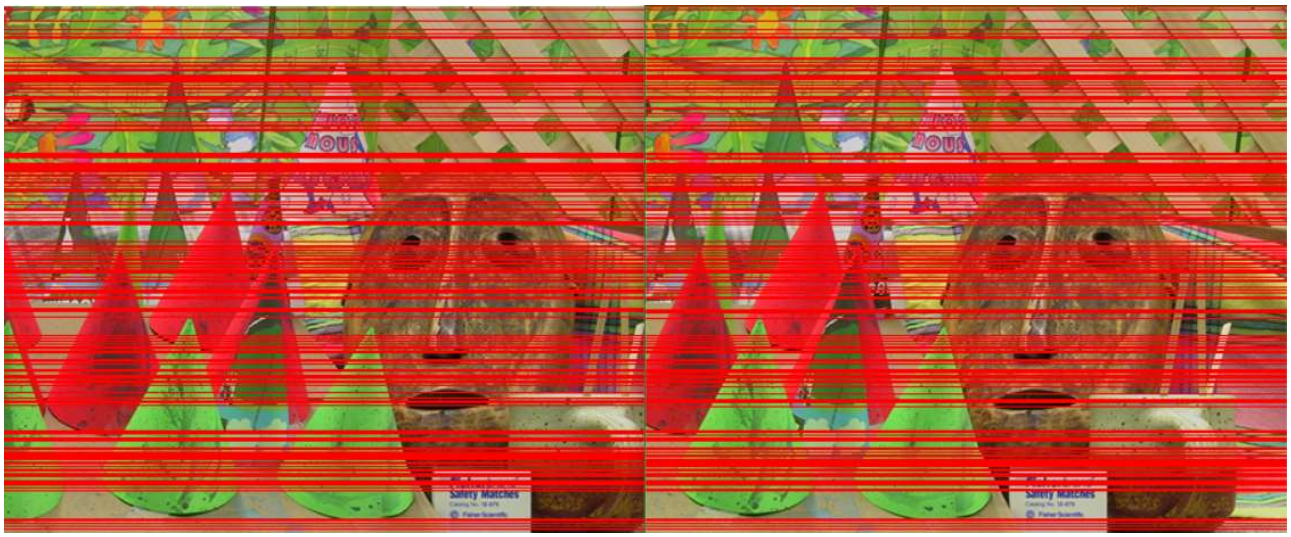

Fig: Epipolar lines on Left and Right image(1)

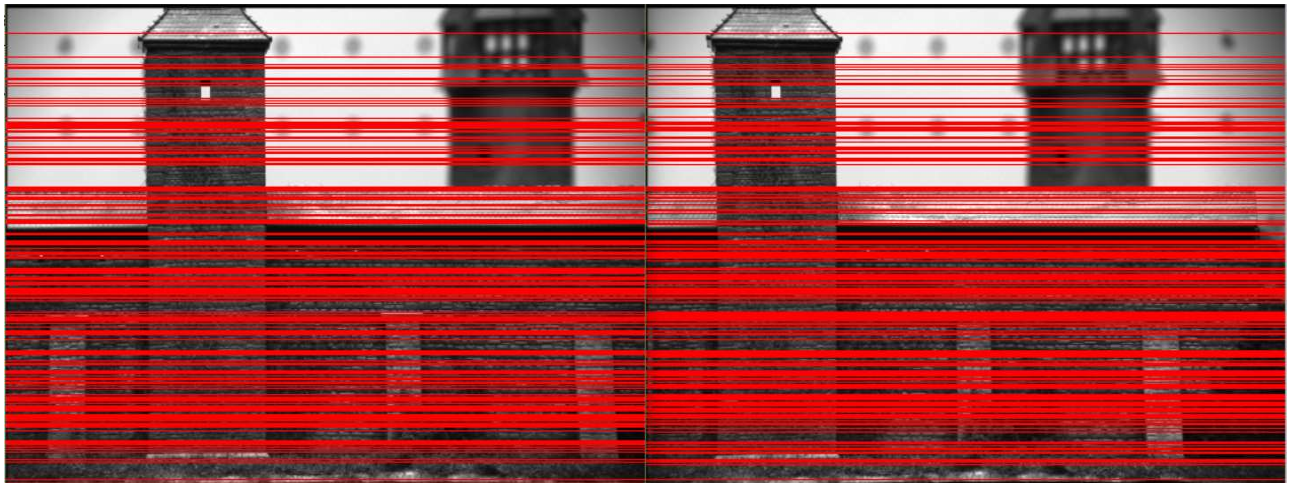

Fig: Epipolar lines on Left and Right image(2)

3. **Dense Disparity maps:** After obtaining the epipolar lines, we use them to find the best matching points in left and right images and compute the horizontal, vertical and colored disparity.
Below is the output for both image sets:

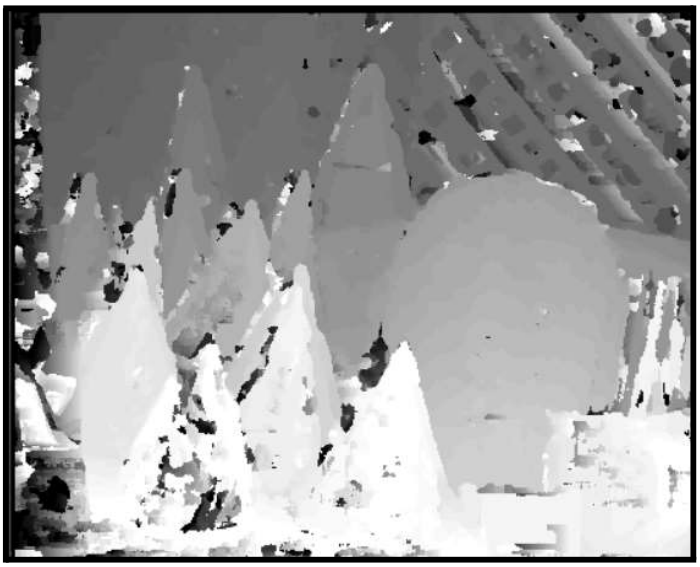

Fig: Horizontal disparity (1)

Fig: Vertical disparity (1)

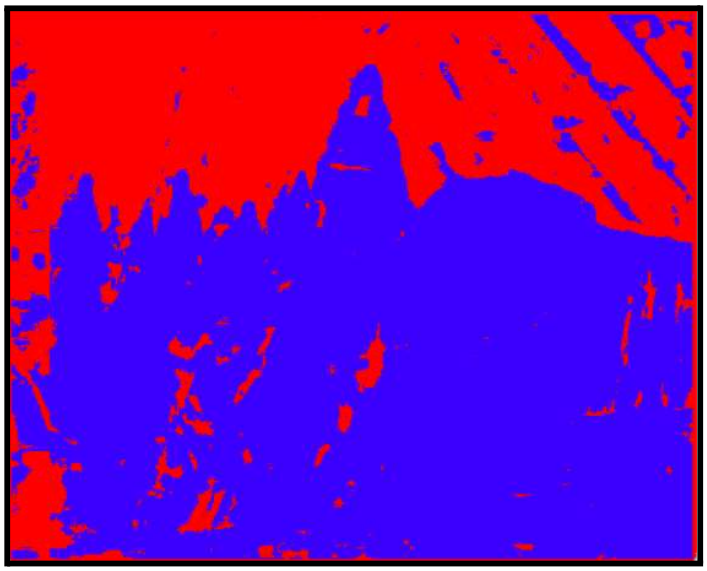

Fig: Color-coded disparity (1)

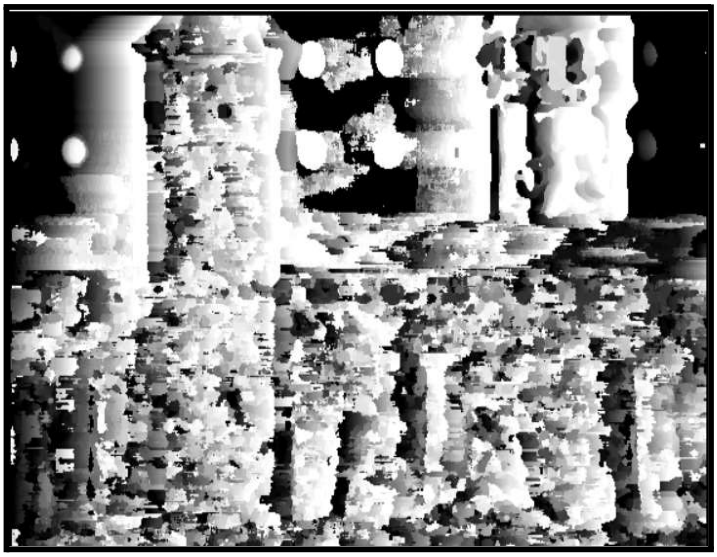

Fig: Horizontal disparity (2)

Fig: Vertical disparity (2)

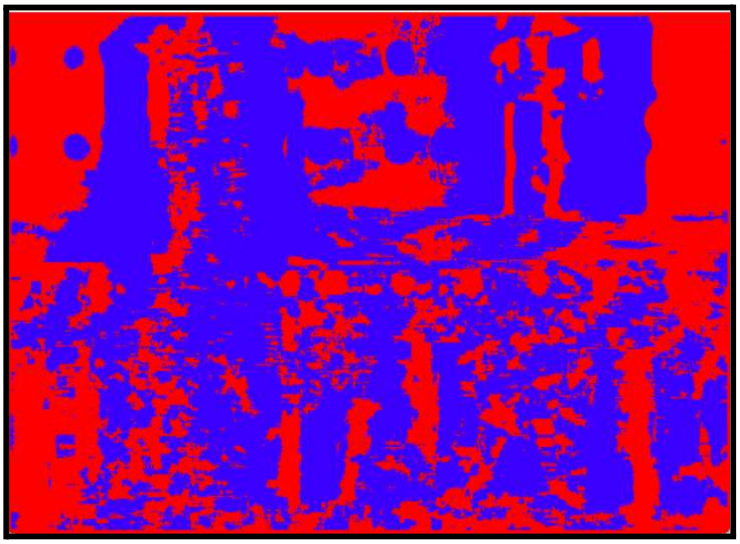

Fig: Color-coded disparity (2)

Conclusion:-

Hartley's pre conditioning is necessary to obtain the fundamental matrix because we tried finding the fundamental matrix without hartley and then plotting the epipolar lines but this causes numerical overflow and once we pre-conditioned the pixel coordinates using hartley and then find the fundamental matrix was a lot better and the values we observed were able to easily plot the epipolar lines. So, Hartley's pre-conditioning is also a must while computing the fundamental matrix. Disparity in y should have been 0 ideally but due to shadows and variation in intensity values we cannot get the ideal disparity = 0. We could have applied smoothing filters on the disparity images but due to shortage of time we weren't able to implement that. Smoothing techniques like gaussian blur or box filter would give us a very nice and smooth image without abrupt changes in intensity values.

From the above experiments and results we observed that the cameras through which these photos were taken were placed parallel to each other which means their epipoles were at infinity and they did not require any rectification given we threshold the error value during RANSAC to be very small. To solidify our belief we observed that after finding the slope and intercept of the epipolar lines we observed that the slope of all the lines were ~ 0 and the intercept were the same for both the lines(left and right). So we could simply traverse the same row in the other image to find the best match using techniques like sum of squared difference or NCC. We also set a minimum and maximum disparity value so that we don't get false positives. For example we don't want the pixel at (0, 0) to match with a pixel at (0, 300) even though it has a higher score because it could be a false positive. So in our case we tried limiting the disparity value in between (0, 64). Another thing to note here is that we could just traverse the same row to find the best match for this case only because the epipoles were already at infinity but in case the epipoles were not at infinity we would have to rectify the image first and only then we could traverse the same row as the template to find the closest match or we could find the row, column values of all the pixels that lie on the epipolar line corresponding to the pixel we want to match. We can then find a match using the row and column values of all the pixels on the epipolar line. This is a bit tricky and usually it is better to rectify the images and then find the best match.

Appendix:-

```
import numpyimport numpy as np
import cv2
import sys
import os
import matplotlib.pyplot as plt
np.set_printoptions(threshold=sys.maxsize)

def find_fundamental_matrix(pts_src, pts_dst):
    pts_src=np.array(pts_src)
    pts_dst = np.array(pts_dst)
    ones_arr_src=np.ones((pts_src.shape[0],1))
    ones_arr_dst=np.ones((pts_dst.shape[0],1))
    pts_src=np.hstack((pts_src,ones_arr_src))
    pts_dst=np.hstack((pts_dst,ones_arr_dst))
    mean_src=np.mean(pts_src[:,:2],axis=0)
    S_src=np.sqrt(2)/np.std(pts_src[:,:2])
    T_src=np.array([[S_src, 0, -S_src*mean_src[0]],[0,S_src,-S_src*mean_src[1]],[0,0,1]])
    pts_src=np.dot(T_src,np.transpose(pts_src))
    pts_src=np.transpose(pts_src)
    mean_dst = np.mean(pts_dst[:, :2], axis=0)
    S_dst = np.sqrt(2)/ np.std(pts_dst[:, :2])
    T_dst = np.array([[S_dst, 0, -S_dst * mean_dst[0]], [0, S_dst, -S_dst * mean_dst[1]], [0, 0, 1]])
    pts_dst=np.dot(T_dst,np.transpose(pts_dst))
    pts_dst=np.transpose(pts_dst)
    # forming A matrix
    a_matrix = np.zeros((len(pts_src), 9))
    for i in range(len(pts_src)):
        a_matrix[i, 0] = pts_src[i][1] * pts_dst[i][1]
        a_matrix[i, 1] = pts_src[i][1] * pts_dst[i][0]
        a_matrix[i, 2] = pts_src[i][1]
        a_matrix[i, 3] = pts_src[i][0] * pts_dst[i][1]
        a_matrix[i, 4] = pts_src[i][0] * pts_dst[i][0]
        a_matrix[i, 5] = pts_src[i][0]
        a_matrix[i, 6] = pts_dst[i][1]
        a_matrix[i, 7] = pts_dst[i][0]
        a_matrix[i, 8] = 1
    u, d, u_transpose = np.linalg.svd(a_matrix, full_matrices=True)
    f = np.transpose(u_transpose[-1].reshape(3, 3))
    u_f, d_f, v_f_transpose = np.linalg.svd(f, full_matrices=True)
    d_f[-1] = 0
    f = np.dot(u_f,np.dot(np.diag(d_f), v_f_transpose))
    f=np.dot(np.transpose(T_dst),np.dot(f,T_src))
    # f = f / f[2, 2]
    return f

class StereoVision:
    def __init__(self, array_of_images: list, ransac_iterations, type_of_derivative_filter="sobel",
                 hc_window_size=(5, 5),ncc_threshold=100000, ncc_window=(7, 7)):
        self.array_of_images = array_of_images
        self.type_of_derivative_filter = type_of_derivative_filter
```

```
        self.window_size = hc_window_size
        self.threshold = ncc_threshold
        self.ncc_window = ncc_window
        self.iterations = ransac_iterations

    def derivative(self):
        i_x = []
        i_y = []
        for image in self.array_of_images:
            image = cv2.boxFilter(image, -1, (3, 3))
            if self.type_of_derivative_filter.lower() == "sobel":
                sobel_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                sobel_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_y))
            elif self.type_of_derivative_filter.lower() == "prewitt":
                prewitt_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                prewitt_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_y))
            else:
                print("Incorrect input")
                return
        return i_x, i_y

    def harris_corner_detector(self, k=0.06):
        i_x, i_y = self.derivative()
        harris_r_array = []
        for ind in range(len(i_x)):
            fx = i_x[ind]
            fy = i_y[ind]
            harris_r = np.zeros(np.shape(fx))
            rows, columns = np.shape(fx)[0], np.shape(fx)[1]
            for row in range(rows):
                for column in range(columns):
                    i = row - self.window_size[0] // 2
                    j = column - self.window_size[0] // 2
                    i_x_2 = 0
                    i_y_2 = 0
                    i_x_y = 0
                    while i <= row + self.window_size[0] // 2:
                        while j <= column + self.window_size[0] // 2:
                            if 0 < i < rows and 0 < j < columns:
                                i_x_2 += np.square(fx[i][j])
                                i_y_2 += np.square(fy[i][j])
                                i_x_y += fx[i][j] * fy[i][j]
                            j += 1
                        i += 1
                    m = np.array([[i_x_2, i_x_y], [i_x_y, i_y_2]])
                    r_score = np.linalg.det(m) - k * (np.square(m[0][0] + m[1][1]))
                    if self.threshold < r_score:
                        harris_r[row][column] = r_score
```

```
            harris_r_array.append(harris_r)
        return harris_r_array

    def non_max_supression(self):
        harris_r_array = self.harris_corner_detector()
        nms_harris_arr = []
        pix_dist = 5
        h, w = np.shape(harris_r_array[0])
        for ind in harris_r_array:
            nms_harris = np.zeros((h, w))
            for i in range(0, h, 64):
                for j in range(0, w, 64):
                    r_array = []
                    for m in range(i + pix_dist, i + 64 - pix_dist):
                        for n in range(j + pix_dist, j + 64 - pix_dist):
                            if m < h and n < w:
                                if ind[m, n] > 0 and ind[m, n] == np.max(
                                        ind[m - pix_dist:m + pix_dist + 1, n - pix_dist:n + pix_dist
+ 1]):
                                    r_array.append((ind[m, n], m, n))
                    if len(r_array) < 10:
                        for p in range(len(r_array)):
                            nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]
                    else:
                        r_array = sorted(r_array, reverse=True)
                        for p in range(10):
                            nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]

            nms_harris_arr.append(nms_harris)
        return nms_harris_arr

    def find_correspondences(self):
        corners_array = self.non_max_supression()
        corners_0 = corners_array[0]
        corners_1 = corners_array[1]
        rows, columns = np.shape(corners_0)[0], np.shape(corners_0)[1]
        correspondences_picture0 = []
        correspondences_picture1 = []
        for row in range(rows):
            for column in range(columns):
                if corners_0[row][column] > 0:
                    a = row - (self.ncc_window[0] // 2)
                    b = row + (self.ncc_window[0] // 2) + 1
                    c = column - (self.ncc_window[0] // 2)
                    d = column + (self.ncc_window[0] // 2) + 1
                    if 0 < a < rows - self.ncc_window[0] and 0 < c < columns - self.ncc_window[0]:
                        template = np.array(self.array_of_images[0][a:b, c:d])
                        max_ncc = 0
                        max_row = None
                        max_column = None
                        for row2 in range(rows):
                            for column2 in range(columns):
```

```
                                if corners_1[row2][column2] > 0:
                                    a2 = row2 - (self.ncc_window[0] // 2)
                                    b2 = row2 + (self.ncc_window[0] // 2) + 1
                                    c2 = column2 - (self.ncc_window[0] // 2)
                                    d2 = column2 + (self.ncc_window[0] // 2) + 1
                                        if 0 < a2 < rows - self.ncc_window[0] and 0 < c2 < columns -
self.ncc_window[0]:
                                    match_to = np.array(self.array_of_images[1][a2:b2, c2:d2])
                                            f = (match_to - match_to.mean()) / (match_to.std() *
np.sqrt(match_to.size))
                                            g = (template - template.mean()) / (template.std() *
np.sqrt(template.size))
                                    product = f * g
                                    stds = np.sum(product)
                                    if stds > max_ncc:
                                        max_ncc = stds
                                        max_row = row2
                                        max_column = column2
                            if max_row is not None:
                                if max_ncc > 0.90:
                                    correspondences_picture0.append([row, column])
                                    correspondences_picture1.append([max_row, max_column])
        return correspondences_picture0, correspondences_picture1

    def feature_matching(self, img_1, img_r, in_0, in_1):
        pts_0, pts_1 = in_0,in_1
        img_1 = np.copy(img_1)
        img_r = np.copy(img_r)
        shift = np.shape(img_r)[1]
        full_img = np.concatenate((img_1, img_r), axis=1)

        for i in range(len(pts_0)):
            color = list(np.random.random_sample(size=3) * 256)
            cv2.line(full_img, (pts_0[i][1], pts_0[i][0]), (pts_1[i][1] + shift, pts_1[i][0]),
                     color, thickness=1)

        return full_img

    def ransac(self):
        corres_0, corres_1 = self.find_correspondences()
        max_inliers = 0
        fundamental_matrix_with_all = None
        max_inliers_0 = []
        max_inliers_1 = []
        for it in range(self.iterations):
            random_sampled_indx = []
            pts_src = []
            pts_dst = []
            i = 0
            while i < 8:
                random_sample = int(np.random.randint(0, high=len(corres_0) - 1, size=1, dtype=int))
                if random_sample not in random_sampled_indx:
```

```
                    pts_src.append(corres_0[random_sample])
                    pts_dst.append(corres_1[random_sample])
                    random_sampled_indx.append(random_sample)
                    i += 1
            fundamental_matrix = find_fundamental_matrix(pts_src, pts_dst)
            j = 0
            inliers = 0
            inliers_0 = []
            inliers_1 = []
            while j < len(corres_0):
                pl = np.transpose(np.array([[corres_0[j][1], corres_0[j][0], 1]]))
                pr_t = np.array([[corres_1[j][1], corres_1[j][0], 1]])
                est = np.dot(pr_t,np.dot(fundamental_matrix,pl))
                # print(est)
                if abs(est) < 0.007:
                    inliers += 1
                    inliers_0.append(corres_0[j])
                    inliers_1.append(corres_1[j])
                j += 1
            if inliers > max_inliers:
                max_inliers = inliers
                max_inliers_0 = inliers_0
                max_inliers_1 = inliers_1
                if max_inliers > 7:
                                fundamental_matrix_with_all  =  find_fundamental_matrix(max_inliers_0,
max_inliers_1)
        return fundamental_matrix_with_all, max_inliers_0, max_inliers_1

    def disp_img(self, img1, img2):
        nms_harris = self.non_max_supression()
        rgb_imgs_arr = [img1, img2]
        corner_imgs = []
        for k in range(len(nms_harris)):
            rgb = np.copy(rgb_imgs_arr[k])
            r, c = np.shape(rgb)[0], np.shape(rgb)[1]
            for i in range(r):
                for j in range(c):
                    if nms_harris[k][i, j] > 0:
                        rgb[i, j] = [0, 0, 255]
            corner_imgs.append(rgb)
        return corner_imgs

def disparity_map(array_of_images, fund_matrix):
    left_img=array_of_images[0]
    right_img=array_of_images[1]
    rows,cols=left_img.shape[0],left_img.shape[1]
    window_size=7
    block=window_size//2
    hori_disp_img=np.zeros_like(left_img,np.uint8)
    vert_disp_img=np.zeros_like(left_img,np.uint8)
    disp_color=np.zeros_like(left_img, np.uint8)
    epi_lines_right=np.zeros((rows,cols,3))
```

```
    max_disp=50
    for i in range(rows):
        for j in range(cols):
            pt_1=np.transpose(np.array([[j,i,1]]))
            l_r=np.dot(np.transpose(fund_matrix),pt_1)
            epi_lines_right[i, j, :] =l_r.reshape(-1)
    for row in range(block,rows-block):
        for col in range(block,cols-block):
            a, b, c = epi_lines_right[row,col]
            left_win=left_img[row-block:row+block+1,col-block:col+block+1]
            max_similarity = float('inf')
            best_match_col=-1
            best_match_row=-1
            for k in range(max_disp):
                y_r = int((-a * (col+k) - c) / b)
                  right_win = right_img[row - block:row+ block + 1, col - block - k:col + block + 1 -
k]
                # right_win=right_img[y_r-block:y_r+block+1,col-block+k:col+block+1+k]
                if left_win.shape == right_win.shape:
                    sad = np.sum(abs(left_win - right_win))
                    if sad < max_similarity:
                        max_similarity = sad
                        best_match_col = k
                        best_match_row = row
            hori_disp=col-(col-best_match_col)
            vert_disp=row-(row-best_match_row)
            hori_disp = np.clip(hori_disp, 0, 255)
            vert_disp = np.clip(vert_disp, 0, 255)
            hori_disp_img[row, col] =hori_disp
            vert_disp_img[row,col]=vert_disp
            disp_color[row,col]=np.sqrt(np.sum((hori_disp**2,vert_disp**2)))
    disp_color=cv2.normalize(hori_disp_img, None, 0, 1, cv2.NORM_MINMAX)
    hue = disp_color*0.5
    saturation = np.ones_like(disp_color,dtype=np.float32)
    disparity_hsv = np.stack((hue, saturation, saturation), axis=-1)
    disparity_hsv=(255*disparity_hsv).astype(np.uint8)
    disparity_hsv = cv2.cvtColor(disparity_hsv, cv2.COLOR_HSV2BGR)
    hori_disp_img=cv2.normalize(hori_disp_img,None, 0, 255, cv2.NORM_MINMAX)
    vert_disp_img= cv2.normalize(vert_disp_img, None, 0, 255, cv2.NORM_MINMAX)
    return hori_disp_img,vert_disp_img,disparity_hsv

def drawlines(left_img,right_img,pts1, pts2,fund_matrix):
        ''' img1 - image on which we draw the epilines for the points in img2
            lines - corresponding epilines '''
        img1 = right_img
        img2 = left_img
        r, c = img1.shape[0],img1.shape[1]
        for i in range(len(pts1)):
            pt_1 = np.transpose(np.array([[pts1[i][1], pts1[i][0], 1]]))
            l_r = np.dot(np.transpose(fund_matrix), pt_1)
            color = list(np.random.random_sample(size=3) * 256)
            x0, y0 = map(int, [0, -l_r[2] / l_r[1]])
```

```
            x1, y1 = map(int, [c, -(l_r[2] + l_r[0] * c) / l_r[1]])
            img1 = cv2.line(img1, (x0, y0), (x1, y1), color, 1)
        for j in range(len(pts2)):
            pr_t=np.array([pts2[j][1],pts2[j][0],1])
            l_l=np.dot(pr_t,fund_matrix)
            color = list(np.random.random_sample(size=3) * 256)
            x0_l, y0_l = map(int, [0, -l_l[2] / l_l[1]])
            x1_l, y1_l = map(int, [c, -(l_l[2] + l_l[0] * c) / l_l[1]])
            img2 = cv2.line(img2, (x0_l, y0_l), (x1_l, y1_l), color, 1)
        return img1,img2


if __name__=='__main__':
    data1_l = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\cast-left-1.jpg"
    data1_r = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\cast-right-1.jpg"
    data1 = [data1_l, data1_r]

    data2_l = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\image-3.jpeg"
    data2_r = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\image-4.jpeg"
    data2 = [data2_l, data2_r]
    imgs_arr_gray_data2 = []
    imgs_arr_rgb_data2 = []
    for img_path in data2:
        img_gray = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)  # Read and convert the image to
grayscale
        imgs_arr_gray_data2.append(np.asarray(img_gray).astype(float))  # Create an array of all the
images

        img_rgb = cv2.imread(img_path, cv2.IMREAD_COLOR)  # Read the image as RGB
        imgs_arr_rgb_data2.append(np.asarray(img_rgb))
    stereo_v_data2 = StereoVision(imgs_arr_gray_data2, 10000)
    # bef_rans_0, bef_rans_1=stereo_v_data2.find_correspondences()
                                                                                                #
img_bef=stereo_v_data2.feature_matching(imgs_arr_rgb_data2[0],imgs_arr_rgb_data2[1],bef_rans_0,bef_r
ans_1)
    # cv2.imshow('img_before',img_bef)
    # cv2.waitKey(0)

    if os.path.exists('fundamental_matrix_data2.npy'):
        F_data2 = np.load('fundamental_matrix_data2.npy')
        in1_data2=np.load('in1_data2.npy')
        in2_data2=np.load('in2_data2.npy')
    else:
        # Compute the fundamental matrix using RANSAC
        F_data2, in1_data2, in2_data2=stereo_v_data2.ransac()
        np.save('fundamental_matrix_data2.npy', F_data2)
        np.save('in1_data2.npy',in1_data2)
        np.save('in2_data2.npy',in2_data2)

      #  epi_lines_r,epi_lines_l  =  drawlines(imgs_arr_rgb_data2[0],imgs_arr_rgb_data2[1],in1_data2,
in2_data2, F_data2)
    # full_img=np.concatenate((epi_lines_l,epi_lines_r),axis=1)
```

```
    # cv2.imshow('Epilines on left and right image',full_img)
    # cv2.waitKey(0)

    #  retBool, rectmat1, rectmat2 = cv2.stereoRectifyUncalibrated(in1_data2, in2_data2, F_data2,
(450, 375))
    # left_img_rect = cv2.warpPerspective(imgs_arr_gray_data2[0], rectmat1, (450, 375))
    # right_img_rect=cv2.warpPerspective(imgs_arr_gray_data2[1], rectmat2, (450, 375))
    # cv2.imshow('img',left_img_rect.astype(np.uint8))
    # cv2.waitKeyEx(0)

    # print(left_img_rect)
    # cv2.imshow('img',full_img)
    # cv2.waitKey(0)
    # cv2.imshow('img', right_img_rect)
    # cv2.waitKey(0)

    hori_disp_img,vertical_disp_img,colored_disp=disparity_map(imgs_arr_gray_data2, F_data2)
    full_img = np.concatenate((hori_disp_img, vertical_disp_img), axis=1)
    cv2.imshow('img',hori_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img',vertical_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img',colored_disp)
    cv2.waitKey(0)
                                                                                                #
img_aft=stereo_v_data2.feature_matching(imgs_arr_rgb_data2[0],imgs_arr_rgb_data2[1],in1_data2,in2_da
ta2)
    # cv2.imshow('img_before',img_aft)
    # cv2.waitKey(0)
    #
    imgs_arr_gray_data1 = []
    imgs_arr_rgb_data1 = []
    for img_path in data1:
            img_gray = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)   # Read and convert the image to
grayscale
            imgs_arr_gray_data1.append(np.asarray(img_gray).astype(float))  # Create an array of all the
images

            img_rgb = cv2.imread(img_path, cv2.IMREAD_COLOR)  # Read the image as RGB
            imgs_arr_rgb_data1.append(np.asarray(img_rgb))
    stereo_v_data1 = StereoVision(imgs_arr_gray_data1, 10000)
    # F,in1,in2=stereo_v_data1.ransac()
                                                                                                #
img_bef=stereo_v_data1.feature_matching(imgs_arr_rgb_data1[0],imgs_arr_rgb_data1[1],bef_rans_0,bef_r
ans_1)
    # cv2.imshow('img_before',img_bef)
    # # cv2.waitKey(0)

    if os.path.exists('fundamental_matrix_data1.npy'):
            F_data1 = np.load('fundamental_matrix_data1.npy')
            in1_data1 = np.load('in1_data1.npy')
            in2_data1 = np.load('in2_data1.npy')
```

```
    else:
        # Compute the fundamental matrix using RANSAC
        F_data1, in1_data1, in2_data1 = stereo_v_data1.ransac()
        np.save('fundamental_matrix_data1.npy', F_data1)
        np.save('in1_data1.npy', in1_data1)
        np.save('in2_data1.npy', in2_data1)

                                                                                #
img_aft=stereo_v_data1.feature_matching(imgs_arr_rgb_data1[0],imgs_arr_rgb_data1[1],in1_data1,in2_da
ta1)
    # cv2.imshow('img_before',img_aft)
    # cv2.waitKey(0)
    hori_disp_img,vertical_disp_img,colored_disp=disparity_map(imgs_arr_gray_data1, F_data1)
    full_img = np.concatenate((hori_disp_img, vertical_disp_img), axis=1)
    cv2.imshow('img', hori_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img', vertical_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img', colored_disp)
    cv2.waitKey(0)
      #  epi_lines_r,epi_lines_l = drawlines(imgs_arr_gray_data1[0],imgs_arr_gray_data1[1],in1_data1,
in2_data1, F_data1)
    # full_img=np.concatenate((epi_lines_l.astype(np.uint8),epi_lines_r.astype(np.uint8)),axis=1)
    # cv2.imshow('img',full_img)
    # cv2.waitKey(0)
 as np
import cv2
import sys
import os
import matplotlib.pyplot as plt
np.set_printoptions(threshold=sys.maxsize)

def find_fundamental_matrix(pts_src, pts_dst):
    pts_src=np.array(pts_src)
    pts_dst = np.array(pts_dst)
    ones_arr_src=np.ones((pts_src.shape[0],1))
    ones_arr_dst=np.ones((pts_dst.shape[0],1))
    pts_src=np.hstack((pts_src,ones_arr_src))
    pts_dst=np.hstack((pts_dst,ones_arr_dst))
    mean_src=np.mean(pts_src[:,:2],axis=0)
    S_src=np.sqrt(2)/np.std(pts_src[:,:2])
    T_src=np.array([[S_src, 0, -S_src*mean_src[0]],[0,S_src,-S_src*mean_src[1]],[0,0,1]])
    pts_src=np.dot(T_src,np.transpose(pts_src))
    pts_src=np.transpose(pts_src)
    mean_dst = np.mean(pts_dst[:, :2], axis=0)
    S_dst = np.sqrt(2)/ np.std(pts_dst[:, :2])
    T_dst = np.array([[S_dst, 0, -S_dst * mean_dst[0]], [0, S_dst, -S_dst * mean_dst[1]], [0, 0, 1]])
    pts_dst=np.dot(T_dst,np.transpose(pts_dst))
    pts_dst=np.transpose(pts_dst)
    # forming A matrix
    a_matrix = np.zeros((len(pts_src), 9))
    for i in range(len(pts_src)):
```

```
        a_matrix[i, 0] = pts_src[i][1] * pts_dst[i][1]
        a_matrix[i, 1] = pts_src[i][1] * pts_dst[i][0]
        a_matrix[i, 2] = pts_src[i][1]
        a_matrix[i, 3] = pts_src[i][0] * pts_dst[i][1]
        a_matrix[i, 4] = pts_src[i][0] * pts_dst[i][0]
        a_matrix[i, 5] = pts_src[i][0]
        a_matrix[i, 6] = pts_dst[i][1]
        a_matrix[i, 7] = pts_dst[i][0]
        a_matrix[i, 8] = 1
    u, d, u_transpose = np.linalg.svd(a_matrix, full_matrices=True)
    f = np.transpose(u_transpose[-1].reshape(3, 3))
    u_f, d_f, v_f_transpose = np.linalg.svd(f, full_matrices=True)
    d_f[-1] = 0
    f = np.dot(u_f,np.dot(np.diag(d_f), v_f_transpose))
    f=np.dot(np.transpose(T_dst),np.dot(f,T_src))
    # f = f / f[2, 2]
    return f

class StereoVision:
    def __init__(self, array_of_images: list, ransac_iterations, type_of_derivative_filter="sobel",
                 hc_window_size=(5, 5),ncc_threshold=100000, ncc_window=(7, 7)):
        self.array_of_images = array_of_images
        self.type_of_derivative_filter = type_of_derivative_filter
        self.window_size = hc_window_size
        self.threshold = ncc_threshold
        self.ncc_window = ncc_window
        self.iterations = ransac_iterations

    def derivative(self):
        i_x = []
        i_y = []
        for image in self.array_of_images:
            image = cv2.boxFilter(image, -1, (3, 3))
            if self.type_of_derivative_filter.lower() == "sobel":
                sobel_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                sobel_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=sobel_mask_y))
            elif self.type_of_derivative_filter.lower() == "prewitt":
                prewitt_mask_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
                prewitt_mask_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
                i_x.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_x))
                i_y.append(cv2.filter2D(image, ddepth=-1, kernel=prewitt_mask_y))
            else:
                print("Incorrect input")
                return
        return i_x, i_y

    def harris_corner_detector(self, k=0.06):
        i_x, i_y = self.derivative()
        harris_r_array = []
        for ind in range(len(i_x)):
```

```
            fx = i_x[ind]
            fy = i_y[ind]
            harris_r = np.zeros(np.shape(fx))
            rows, columns = np.shape(fx)[0], np.shape(fx)[1]
            for row in range(rows):
                for column in range(columns):
                    i = row - self.window_size[0] // 2
                    j = column - self.window_size[0] // 2
                    i_x_2 = 0
                    i_y_2 = 0
                    i_x_y = 0
                    while i <= row + self.window_size[0] // 2:
                        while j <= column + self.window_size[0] // 2:
                            if 0 < i < rows and 0 < j < columns:
                                i_x_2 += np.square(fx[i][j])
                                i_y_2 += np.square(fy[i][j])
                                i_x_y += fx[i][j] * fy[i][j]
                            j += 1
                        i += 1
                    m = np.array([[i_x_2, i_x_y], [i_x_y, i_y_2]])
                    r_score = np.linalg.det(m) - k * (np.square(m[0][0] + m[1][1]))
                    if self.threshold < r_score:
                        harris_r[row][column] = r_score
            harris_r_array.append(harris_r)
        return harris_r_array

    def non_max_supression(self):
        harris_r_array = self.harris_corner_detector()
        nms_harris_arr = []
        pix_dist = 5
        h, w = np.shape(harris_r_array[0])
        for ind in harris_r_array:
            nms_harris = np.zeros((h, w))
            for i in range(0, h, 64):
                for j in range(0, w, 64):
                    r_array = []
                    for m in range(i + pix_dist, i + 64 - pix_dist):
                        for n in range(j + pix_dist, j + 64 - pix_dist):
                            if m < h and n < w:
                                if ind[m, n] > 0 and ind[m, n] == np.max(
                                        ind[m - pix_dist:m + pix_dist + 1, n - pix_dist:n + pix_dist
+ 1]):
                                    r_array.append((ind[m, n], m, n))
                    if len(r_array) < 10:
                        for p in range(len(r_array)):
                            nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]
                    else:
                        r_array = sorted(r_array, reverse=True)
                        for p in range(10):
                            nms_harris[r_array[p][1], r_array[p][2]] = r_array[p][0]

            nms_harris_arr.append(nms_harris)
```

```
        return nms_harris_arr

    def find_correspondences(self):
        corners_array = self.non_max_supression()
        corners_0 = corners_array[0]
        corners_1 = corners_array[1]
        rows, columns = np.shape(corners_0)[0], np.shape(corners_0)[1]
        correspondences_picture0 = []
        correspondences_picture1 = []
        for row in range(rows):
            for column in range(columns):
                if corners_0[row][column] > 0:
                    a = row - (self.ncc_window[0] // 2)
                    b = row + (self.ncc_window[0] // 2) + 1
                    c = column - (self.ncc_window[0] // 2)
                    d = column + (self.ncc_window[0] // 2) + 1
                    if 0 < a < rows - self.ncc_window[0] and 0 < c < columns - self.ncc_window[0]:
                        template = np.array(self.array_of_images[0][a:b, c:d])
                        max_ncc = 0
                        max_row = None
                        max_column = None
                        for row2 in range(rows):
                            for column2 in range(columns):
                                if corners_1[row2][column2] > 0:
                                    a2 = row2 - (self.ncc_window[0] // 2)
                                    b2 = row2 + (self.ncc_window[0] // 2) + 1
                                    c2 = column2 - (self.ncc_window[0] // 2)
                                    d2 = column2 + (self.ncc_window[0] // 2) + 1
                                    if 0 < a2 < rows - self.ncc_window[0] and 0 < c2 < columns -
self.ncc_window[0]:
                                        match_to = np.array(self.array_of_images[1][a2:b2, c2:d2])
                                        f = (match_to - match_to.mean()) / (match_to.std() *
np.sqrt(match_to.size))
                                        g = (template - template.mean()) / (template.std() *
np.sqrt(template.size))
                                        product = f * g
                                        stds = np.sum(product)
                                        if stds > max_ncc:
                                            max_ncc = stds
                                            max_row = row2
                                            max_column = column2
                        if max_row is not None:
                            if max_ncc > 0.90:
                                correspondences_picture0.append([row, column])
                                correspondences_picture1.append([max_row, max_column])
        return correspondences_picture0, correspondences_picture1

    def feature_matching(self, img_l, img_r, in_0, in_1):
        pts_0, pts_1 = in_0,in_1
        img_l = np.copy(img_l)
        img_r = np.copy(img_r)
        shift = np.shape(img_r)[1]
```

```
        full_img = np.concatenate((img_1, img_r), axis=1)

        for i in range(len(pts_0)):
            color = list(np.random.random_sample(size=3) * 256)
            cv2.line(full_img, (pts_0[i][1], pts_0[i][0]), (pts_1[i][1] + shift, pts_1[i][0]),
                     color, thickness=1)

        return full_img

    def ransac(self):
        corres_0, corres_1 = self.find_correspondences()
        max_inliers = 0
        fundamental_matrix_with_all = None
        max_inliers_0 = []
        max_inliers_1 = []
        for it in range(self.iterations):
            random_sampled_indx = []
            pts_src = []
            pts_dst = []
            i = 0
            while i < 8:
                random_sample = int(np.random.randint(0, high=len(corres_0) - 1, size=1, dtype=int))
                if random_sample not in random_sampled_indx:
                    pts_src.append(corres_0[random_sample])
                    pts_dst.append(corres_1[random_sample])
                    random_sampled_indx.append(random_sample)
                    i += 1
            fundamental_matrix = find_fundamental_matrix(pts_src, pts_dst)
            j = 0
            inliers = 0
            inliers_0 = []
            inliers_1 = []
            while j < len(corres_0):
                p1 = np.transpose(np.array([[corres_0[j][1], corres_0[j][0], 1]]))
                pr_t = np.array([[corres_1[j][1], corres_1[j][0], 1]])
                est = np.dot(pr_t,np.dot(fundamental_matrix,p1))
                # print(est)
                if abs(est) < 0.007:
                    inliers += 1
                    inliers_0.append(corres_0[j])
                    inliers_1.append(corres_1[j])
                j += 1
            if inliers > max_inliers:
                max_inliers = inliers
                max_inliers_0 = inliers_0
                max_inliers_1 = inliers_1
                if max_inliers > 7:
                                fundamental_matrix_with_all  =  find_fundamental_matrix(max_inliers_0,
max_inliers_1)
        return fundamental_matrix_with_all, max_inliers_0, max_inliers_1

    def disp_img(self, img1, img2):
```

```
        nms_harris = self.non_max_supression()
        rgb_imgs_arr = [img1, img2]
        corner_imgs = []
        for k in range(len(nms_harris)):
            rgb = np.copy(rgb_imgs_arr[k])
            r, c = np.shape(rgb)[0], np.shape(rgb)[1]
            for i in range(r):
                for j in range(c):
                    if nms_harris[k][i, j] > 0:
                        rgb[i, j] = [0, 0, 255]
            corner_imgs.append(rgb)
        return corner_imgs

def disparity_map(array_of_images, fund_matrix):
    left_img=array_of_images[0]
    right_img=array_of_images[1]
    rows,cols=left_img.shape[0],left_img.shape[1]
    window_size=7
    block=window_size//2
    hori_disp_img=np.zeros_like(left_img,np.uint8)
    vert_disp_img=np.zeros_like(left_img,np.uint8)
    disp_color=np.zeros_like(left_img, np.uint8)
    epi_lines_right=np.zeros((rows,cols,3))
    max_disp=50
    for i in range(rows):
        for j in range(cols):
            pt_l=np.transpose(np.array([[j,i,1]]))
            l_r=np.dot(np.transpose(fund_matrix),pt_l)
            epi_lines_right[i, j, :] =l_r.reshape(-1)
    for row in range(block,rows-block):
        for col in range(block,cols-block):
            a, b, c = epi_lines_right[row,col]
            left_win=left_img[row-block:row+block+1,col-block:col+block+1]
            max_similarity = float('inf')
            best_match_col=-1
            best_match_row=-1
            for k in range(max_disp):
                y_r = int((-a * (col+k) - c) / b)
                  right_win = right_img[row - block:row+ block + 1, col - block - k:col + block + 1 -
k]
                # right_win=right_img[y_r-block:y_r+block+1,col-block+k:col+block+1+k]
                if left_win.shape == right_win.shape:
                    sad = np.sum(abs(left_win - right_win))
                    if sad < max_similarity:
                        max_similarity = sad
                        best_match_col = k
                        best_match_row = row
            hori_disp=col-(col-best_match_col)
            vert_disp=row-(row-best_match_row)
            hori_disp = np.clip(hori_disp, 0, 255)
            vert_disp = np.clip(vert_disp, 0, 255)
            hori_disp_img[row, col] =hori_disp
```

```
            vert_disp_img[row,col]=vert_disp
            disp_color[row,col]=np.sqrt(np.sum((hori_disp**2,vert_disp**2)))
    disp_color=cv2.normalize(hori_disp_img, None, 0, 1, cv2.NORM_MINMAX)
    hue = disp_color*0.5
    saturation = np.ones_like(disp_color,dtype=np.float32)
    disparity_hsv = np.stack((hue, saturation, saturation), axis=-1)
    disparity_hsv=(255*disparity_hsv).astype(np.uint8)
    disparity_hsv = cv2.cvtColor(disparity_hsv, cv2.COLOR_HSV2BGR)
    hori_disp_img=cv2.normalize(hori_disp_img,None, 0, 255, cv2.NORM_MINMAX)
    vert_disp_img= cv2.normalize(vert_disp_img, None, 0, 255, cv2.NORM_MINMAX)
    return hori_disp_img,vert_disp_img,disparity_hsv

def drawlines(left_img,right_img,pts1, pts2,fund_matrix):
        ''' img1 - image on which we draw the epilines for the points in img2
            lines - corresponding epilines '''
        img1 = right_img
        img2 = left_img
        r, c = img1.shape[0],img1.shape[1]
        for i in range(len(pts1)):
            pt_1 = np.transpose(np.array([[pts1[i][1], pts1[i][0], 1]]))
            l_r = np.dot(np.transpose(fund_matrix), pt_1)
            color = list(np.random.random_sample(size=3) * 256)
            x0, y0 = map(int, [0, -l_r[2] / l_r[1]])
            x1, y1 = map(int, [c, -(l_r[2] + l_r[0] * c) / l_r[1]])
            img1 = cv2.line(img1, (x0, y0), (x1, y1), color, 1)
        for j in range(len(pts2)):
            pr_t=np.array([pts2[j][1],pts2[j][0],1])
            l_l=np.dot(pr_t,fund_matrix)
            color = list(np.random.random_sample(size=3) * 256)
            x0_l, y0_l = map(int, [0, -l_l[2] / l_l[1]])
            x1_l, y1_l = map(int, [c, -(l_l[2] + l_l[0] * c) / l_l[1]])
            img2 = cv2.line(img2, (x0_l, y0_l), (x1_l, y1_l), color, 1)
        return img1,img2


if __name__=='__main__':
    data1_l = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\cast-left-1.jpg"
    data1_r = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\cast-right-1.jpg"
    data1 = [data1_l, data1_r]

    data2_l = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\image-3.jpeg"
    data2_r = r"C:\Users\udayr\PycharmProjects\CVfiles\project3\images\image-4.jpeg"
    data2 = [data2_l, data2_r]
    imgs_arr_gray_data2 = []
    imgs_arr_rgb_data2 = []
    for img_path in data2:
            img_gray = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)   # Read and convert the image to
grayscale
        imgs_arr_gray_data2.append(np.asarray(img_gray).astype(float))  # Create an array of all the
images

        img_rgb = cv2.imread(img_path, cv2.IMREAD_COLOR)  # Read the image as RGB
```

```
        imgs_arr_rgb_data2.append(np.asarray(img_rgb))
    stereo_v_data2 = StereoVision(imgs_arr_gray_data2, 10000)
    # bef_rans_0, bef_rans_1=stereo_v_data2.find_correspondences()
    # img_bef=stereo_v_data2.feature_matching(imgs_arr_rgb_data2[0],imgs_arr_rgb_data2[1],bef_rans_0,bef_rans_1)
    # cv2.imshow('img_before',img_bef)
    # cv2.waitKey(0)

    if os.path.exists('fundamental_matrix_data2.npy'):
        F_data2 = np.load('fundamental_matrix_data2.npy')
        in1_data2=np.load('in1_data2.npy')
        in2_data2=np.load('in2_data2.npy')
    else:
        # Compute the fundamental matrix using RANSAC
        F_data2, in1_data2, in2_data2=stereo_v_data2.ransac()
        np.save('fundamental_matrix_data2.npy', F_data2)
        np.save('in1_data2.npy',in1_data2)
        np.save('in2_data2.npy',in2_data2)

    # epi_lines_r,epi_lines_l = drawlines(imgs_arr_rgb_data2[0],imgs_arr_rgb_data2[1],in1_data2,in2_data2, F_data2)
    # full_img=np.concatenate((epi_lines_l,epi_lines_r),axis=1)
    # cv2.imshow('Epilines on left and right image',full_img)
    # cv2.waitKey(0)

    # retBool, rectmat1, rectmat2 = cv2.stereoRectifyUncalibrated(in1_data2, in2_data2, F_data2, (450, 375))
    # left_img_rect = cv2.warpPerspective(imgs_arr_gray_data2[0], rectmat1, (450, 375))
    # right_img_rect=cv2.warpPerspective(imgs_arr_gray_data2[1], rectmat2, (450, 375))
    # cv2.imshow('img',left_img_rect.astype(np.uint8))
    # cv2.waitKeyEx(0)

    # print(left_img_rect)
    # cv2.imshow('img',full_img)
    # cv2.waitKey(0)
    # cv2.imshow('img', right_img_rect)
    # cv2.waitKey(0)

    hori_disp_img,vertical_disp_img,colored_disp=disparity_map(imgs_arr_gray_data2, F_data2)
    full_img = np.concatenate((hori_disp_img, vertical_disp_img), axis=1)
    cv2.imshow('img',hori_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img',vertical_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img',colored_disp)
    cv2.waitKey(0)
    # img_aft=stereo_v_data2.feature_matching(imgs_arr_rgb_data2[0],imgs_arr_rgb_data2[1],in1_data2,in2_data2)
    # cv2.imshow('img_before',img_aft)
    # cv2.waitKey(0)
```

```
    #
    imgs_arr_gray_data1 = []
    imgs_arr_rgb_data1 = []
    for img_path in data1:
            img_gray = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)  # Read and convert the image to
grayscale
        imgs_arr_gray_data1.append(np.asarray(img_gray).astype(float))  # Create an array of all the
images

        img_rgb = cv2.imread(img_path, cv2.IMREAD_COLOR)  # Read the image as RGB
        imgs_arr_rgb_data1.append(np.asarray(img_rgb))
    stereo_v_data1 = StereoVision(imgs_arr_gray_data1, 10000)
    # F,in1,in2=stereo_v_data1.ransac()
                                                                                                #
img_bef=stereo_v_data1.feature_matching(imgs_arr_rgb_data1[0],imgs_arr_rgb_data1[1],bef_rans_0,bef_r
ans_1)
    # cv2.imshow('img_before',img_bef)
    # # cv2.waitKey(0)

    if os.path.exists('fundamental_matrix_data1.npy'):
        F_data1 = np.load('fundamental_matrix_data1.npy')
        in1_data1 = np.load('in1_data1.npy')
        in2_data1 = np.load('in2_data1.npy')
    else:
        # Compute the fundamental matrix using RANSAC
        F_data1, in1_data1, in2_data1 = stereo_v_data1.ransac()
        np.save('fundamental_matrix_data1.npy', F_data1)
        np.save('in1_data1.npy', in1_data1)
        np.save('in2_data1.npy', in2_data1)

                                                                                                #
img_aft=stereo_v_data1.feature_matching(imgs_arr_rgb_data1[0],imgs_arr_rgb_data1[1],in1_data1,in2_da
ta1)
    # cv2.imshow('img_before',img_aft)
    # cv2.waitKey(0)
    hori_disp_img,vertical_disp_img,colored_disp=disparity_map(imgs_arr_gray_data1, F_data1)
    full_img = np.concatenate((hori_disp_img, vertical_disp_img), axis=1)
    cv2.imshow('img', hori_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img', vertical_disp_img)
    cv2.waitKey(0)
    cv2.imshow('img', colored_disp)
    cv2.waitKey(0)
      # epi_lines_r,epi_lines_1 = drawlines(imgs_arr_gray_data1[0],imgs_arr_gray_data1[1],in1_data1,
in2_data1, F_data1)
    # full_img=np.concatenate((epi_lines_1.astype(np.uint8),epi_lines_r.astype(np.uint8)),axis=1)
    # cv2.imshow('img',full_img)
    # cv2.waitKey(0)
```