Project #3

Stereo Depth Mapping using the Fundamental Matrix to Reduce the Search Space

EECE 5639

Instructor: Octavia Camps

Authors: Juan Pablo Bernal Medina, Shivam Sharma

Abstract

In this project, we used our previous Harris corner detector and non-max suppression algorithm to find point correspondences between a stereo image pair. The corners between the two images are then correlated with a normalized cross-correlation to generate pairs of corners with the highest NCC score. We then use RANSAC to estimate the fundamental matrix between the two images. The fundamental matrix allows us to drastically reduce the search space for finding correspondences on the right images for every pixel on the left. With these correspondences, we find the x and y disparity for each pixel and plot them as depth maps.

Algorithm

Harris corner detector: We began by computing the gradient of the image by using a 3x3 Sobel operator. We then computed the C matrix for each pixel given by:

$$C = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

Where I_x and I_y are the gradients at (x, y). The output of the function gave us a Harris corner response at each pixel which we were then able to use to pinpoint corners on the image.

Non-max suppression: The output of the corner detector is then fed into a non-max suppression algorithm to find local maxima and therefore generate a sparse set of corners. To accomplish this we used a slightly different route and utilized the mean shift to find every local maximum. The algorithm would be given a window size and would start searching the image at coordinates half the window size. It would start at that pixel and use the mean shift algorithm to find the nearest maxima. It would look in a window around that pixel and find the maximum. If the maximum was greater than the threshold it would move to that pixel location and repeat the process. This algorithm allowed us to find the corners with the greatest response and limit the number that we would find. The window size, in this case, limited the closest distance between observed corners.

Normalized cross-correlation (NCC): After running the Harris detector and non-max suppression on both images, we were ready to find correlations between the two. For every pair of corners across both images, an NCC score was assigned:

$$NCC\ score = \sum_{i,j}^{N} \frac{A_{i,j} - \mu_A}{\sigma_A} \cdot \frac{B_{i,j} - \mu_B}{\sigma_B}$$

Where $A_{i,j}$ and $B_{i,j}$ are the pixel values for each image respectively in the window and μ and σ are the mean and standard deviation respectively of the images in the window N.
The NCC pairs were then sorted and filter based on a threshold score. But we did not want to lose too many corners so we had a minimum length as well for output pairings so that in the case the threshold was too high, we would still have enough corners.

Fundamental Matrix estimation (8 point algorithm): To estimate the fundamental matrix, the 8 point algorithm was used. First, we normalized the 8 correspondences (16 points in total) by subtracting the mean of the x and y coordinates (x and y independently) and dividing by the standard deviation. With these values of the mean and the standard deviation from each image, two T matrices were set up

$$T = \begin{bmatrix} \frac{1}{\sigma_x} & 0 & -\mu_x \\ 0 & \frac{1}{\sigma_y} & -\mu_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where **σ** is the standard deviation and μ is the mean. Once the points were normalized, matrix A was computed using the x and y coordinates of each of the 8 correspondences. Using SVD the eigenvectors of $A^T A$ were computed. The last column of the eigenvectors V (the smaller eigenvalue) was extracted and reshaped to a 3x3 matrix to be the Fundamental matrix. To make this matrix more accurate, the SVD was recomputed on the F matrix. Then the last value of D, the square roots of the eigenvalues, was set to 0. The F matrix was then reconstructed using the new D, and the U and V gotten from the SVD. Finally, we denormalized the F matrix by applying $T_2{}^T F T_1$ where T_2 is the matrix from the left image and T_1 is the matrix from the right image.

Disparity Map computation: With the fundamental matrix computed, we could drastically limit the search space for the correspondences of each pixel. Since the fundamental matrix gives the epipolar lines at a given point, our search space for a given pixel reduces to a line on the other image. Additionally, we also used the corner correspondences to estimate the min and max disparity, so our search space further reduces to a bounded line segment. We then looped across the left image and found correspondences for each pixel using the sum of squared differences (SSD). metric We then measured the x and y disparity between each point pair and created two disparity images. Then a third vectorized disparity image was created with the magnitude and angle of the disparity vector calculated with the x and y disparities. The vector disparity was plotted in HSI color space with the angle representing hue, the magnitude representing saturation and intensity set at 1 for each pixel.

Values of parameters used

Harris Corner detection:

Sobel window size: 3

C matrix window size: 3

Non-max suppression window size: 9

NCC:

Window size: 5

RANSAC:

Probability p: 0.99

Probability e: 0.5 for castle and 0.1 for cones

Threshold: 0.001

With these parameters, we computed the correspondences in both sets of images. In fig.1, we can see the correspondences of the cones set of images. Here almost all the correspondences were accurate and after RANSAC we can say that most of them are inliers. Therefore, there is not much difference between Fig.1 and Fig.2. However, in the castle set of images, there were many more false positives than we expected. Since the image is so homogeneous and many of the corners have the same or similar surroundings, the NCC determined that a corner in the right image might have multiple correspondences in the left image. That is why in Fig.3 we can notice that one corner in the right connects to many corners in the left. After doing RANSAC (Fig.4) we can see a significant decrease in correspondences. After running a couple of tests we found that the probability of a correspondence being an outlier for this set of pictures was around 0.5.

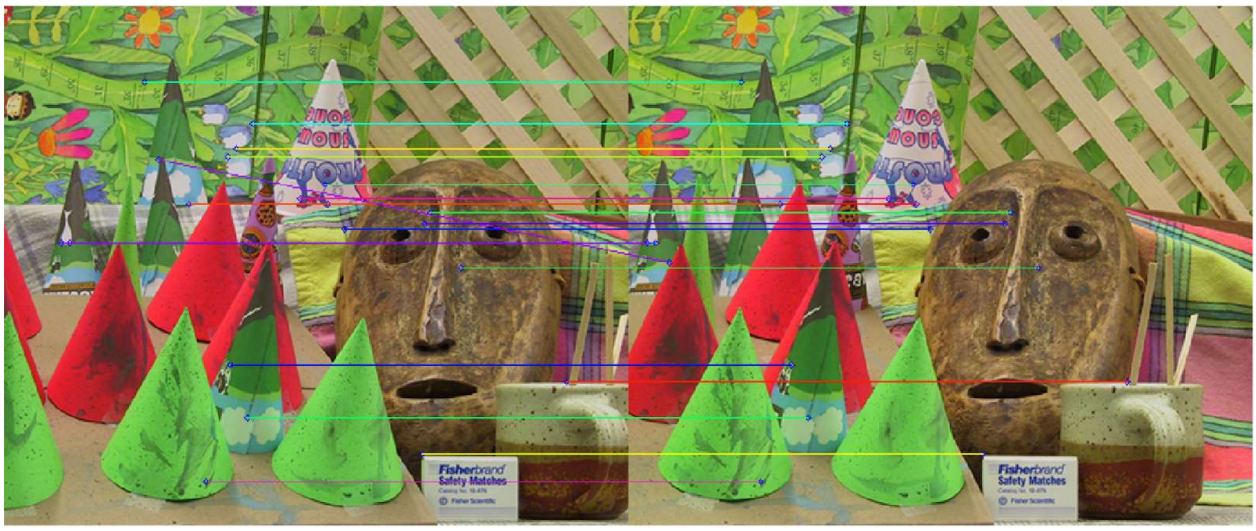

Fig.1 Corners pairings for top NCC pairs - Cones

Fig.2 Corners pairings after RANSAC - Cones

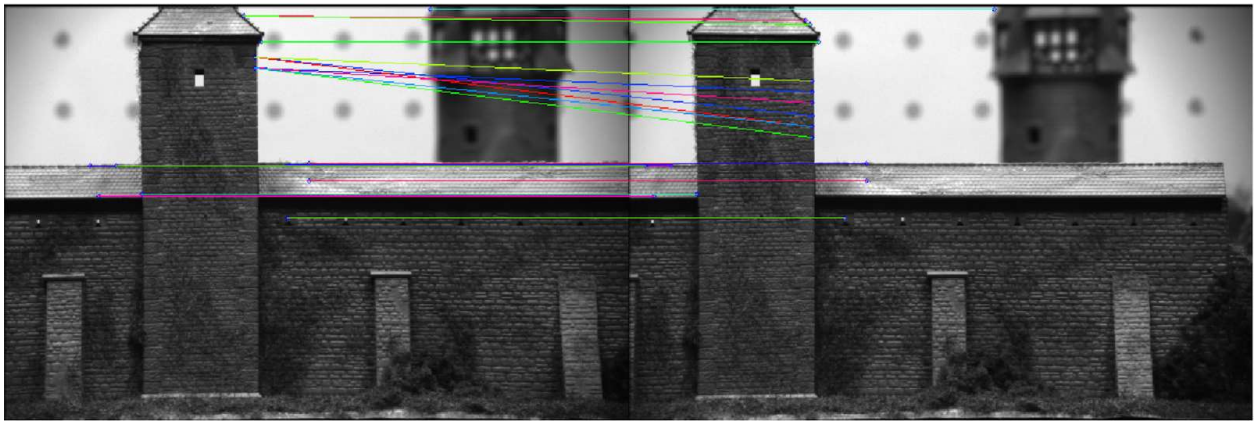

Fig.3 Corners pairings for top NCC pairs - Castle

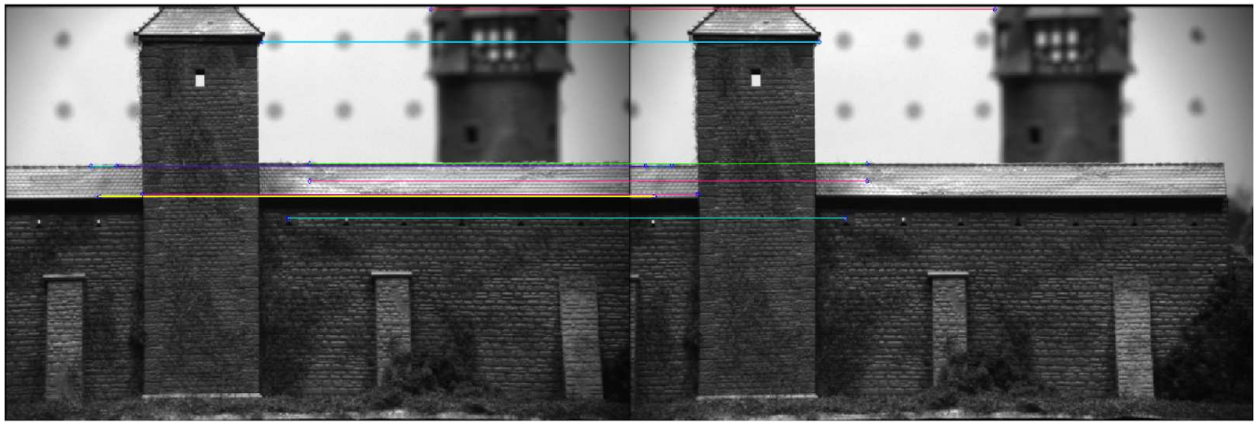

Fig.4 Corners pairings after RANSAC - Castle

After estimating the fundamental matrix, we were ready to move on to compute the disparity image. Since we had the fundamental matrix we could reduce the search space for each point on the left image to a line on the right. Additionally, since we already had some inlier correspondences from our corner set, we could also estimate the min and max disparity found on the image. This was done by simply finding the min and max disparity between the inlier correspondences and multiplying I by 1.5 just to make sure we don't miss anything.

Once we had our search space parameters, we looped through the left image and calculated the sum of squared differences for each pixel in that search space and took the one that produced the least difference. Once we found the point pair we simply computed the x and y disparity and stored it separate images. This was done for every single pixel that existed in both images to produce a full x and y disparity image shown below. The x and y disparities were first normalized before being displayed so that they ranged from 0-255.

We then computer a third vectorized disparity image that had the magnitude and angle of the disparity vector computed from the x and y disparities. This image was then displayed in HSV color space using the angle as the hue, the magnitude as the intensity and setting the value to 1.

Before doing this however we normalized the image by limiting the hue to 275-315 degrees do get that bluish color hue that makes the image easier to interpret. The magnitude was normalized so that we had the full range of saturation from 0-100%, with closer objects being more saturated

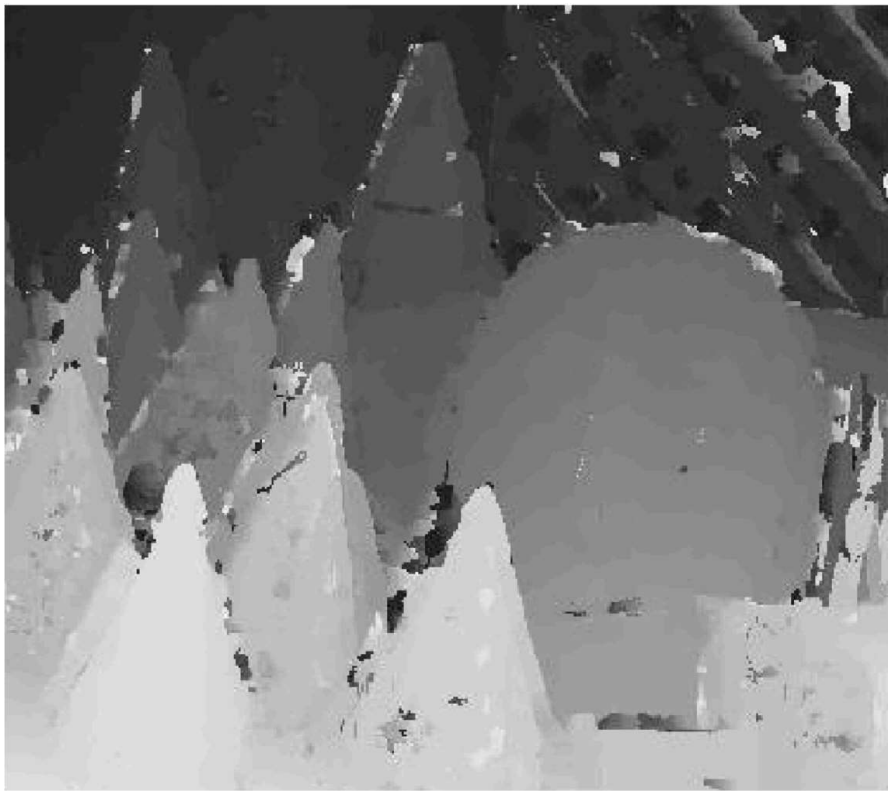

Fig.5 x disparity (black = 12 pixels, white = 57 pixels) - Cones

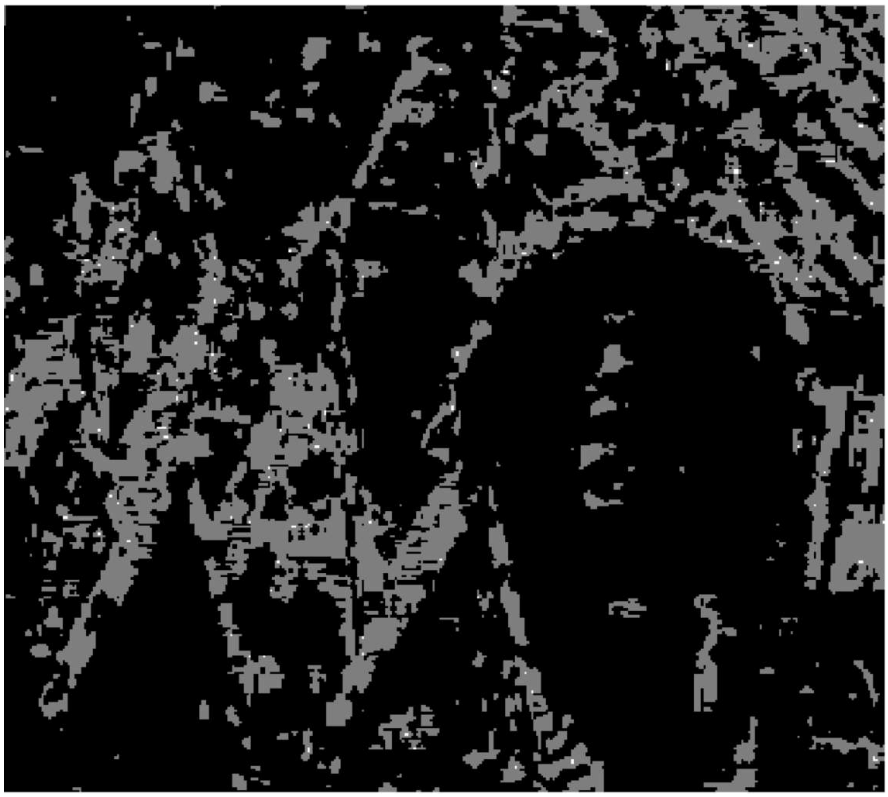

Fig.6 y disparity (black = 0 pixels, white = 2 pixels) - Cones

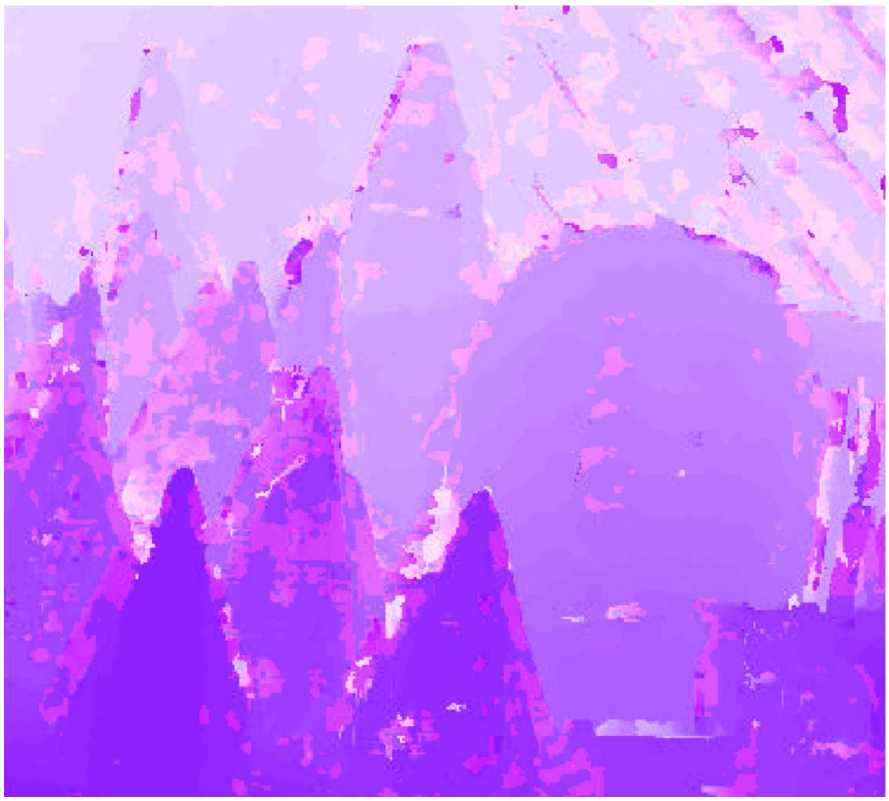

Fig.7 Vector disparity in HSI space - Cones

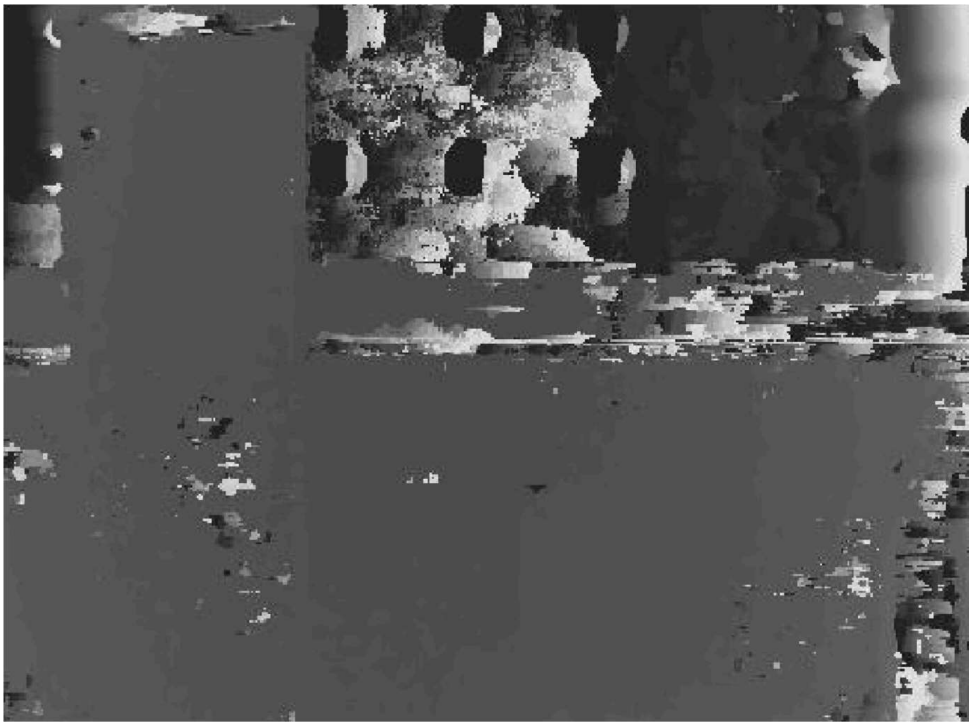

Fig.8 x disparity (black = 47 pixels, white = 97 pixels) - Castle

Fig.9 y disparity (black = 0 pixels, white = 2 pixels) - Castle

Fig.10 Vector disparity in HSI space - Castle

Conclusions

In conclusion, by finding point correspondences between the two images, we were able to calculate the x and y disparity and hence estimate the depth of pixels in the image. By previously finding known point correspondences such as corners, we were able to estimate the fundamental matrix which helped to reduce our search space when we needed to find correspondences for each pixel.

By finding a correspondent for each pixel, we were also able to find an x and y disparity at every location of the image. The disparities were displayed in grayscale for each x and y component, as well as in HSV color space for the vectorized disparity image.

Appendix

Code:

```
import numpy as np
from matplotlib import pyplot as plt
import math
import cv2
import os
import gc
import time
import colorsys


def Harris(img, window_size=3, sobel_size=3, k=0.04, step_size=1):
    if len(img.shape) > 2:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    h, w = img.shape[0], img.shape[1]

    sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=sobel_size)
    sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=sobel_size)

    Ixx = sobelx ** 2
    Iyy = sobely ** 2
    Ixy = sobelx * sobely

    R = np.zeros_like(img, dtype=np.float64)

    offset = np.floor(window_size / 2).astype(np.int8)
    for y in range(offset, h - offset, step_size):
        for x in range(offset, w - offset, step_size):
            Sxx = np.sum(Ixx[y - offset:y + 1 + offset, x - offset:x + 1 + offset])
            Syy = np.sum(Iyy[y - offset:y + 1 + offset, x - offset:x + 1 + offset])
            Sxy = np.sum(Ixy[y - offset:y + 1 + offset, x - offset:x + 1 + offset])
            r = (Sxx * Syy) - (Sxy ** 2) - k * (Sxx + Syy) ** 2
            if r < 0:
                r = 0
            R[y][x] = r

    R /= np.max(R)
    R *= 255.0
    # _, R = cv2.threshold(R, 0.01 * np.max(R), 255, 0)
    return R


def non_max_suppression(img, window_size=5, thresh=2.5):
    h, w = img.shape[0], img.shape[1]

    offset = np.floor(window_size / 2).astype(np.int8)

    temp = np.zeros_like(img)
    temp[offset:h - offset, offset:w - offset] = img[offset:h - offset, offset:w - offset]
    img = temp
    del temp

    index = np.zeros((h, w, 3), dtype=np.int16)

    index[:, :, 0] = img

    for y in range(h):
        for x in range(w):
            index[y][x][1] = x
```

```
            index[y][x][2] = y

    corners = []

    global NMS_index
    NMS_index = np.zeros((h, w))

    for y in range(offset, h - offset, offset):
        for x in range(offset, w - offset, offset):
            ret = mean_shift_converge(index, (x, y), window_size, thresh)
            if ret:
                corners.append(ret)

    corners = list(set(corners))
    corners.sort()
    return corners


def mean_shift_converge(index, point, window_size=5, thresh=2.5):
    x, y = point
    offset = np.floor(window_size / 2).astype(np.int8)

    if NMS_index[y][x] == 1:
        return None

    window = index[y - offset:y + 1 + offset, x - offset:x + 1 + offset]
    if np.max(window[:, :, 0]) > thresh:
        window = np.reshape(window, (window_size ** 2, 3))
        window[::-1] = window[window[:, 0].argsort()]
        if window[0][1] == x and window[0][2] == y:
            NMS_index[y][x] = 1
            return x, y
        else:
            if mean_shift_converge(index, (window[0][1], window[0][2]), window_size, thresh):
                NMS_index[y][x] = 1
                return window[0][1], window[0][2]
            else:
                return None
    else:
        return None


def NCC(img1, img2, c1, c2, window_size=3):
    if len(img1.shape) > 2:
        img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)

    if len(img2.shape) > 2:
        img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    h, w = img1.shape[0], img1.shape[1]

    offset = np.floor(window_size / 2).astype(np.int8)

    ncc = []
    for p1 in c1:
        for p2 in c2:
            (x1, y1) = p1
            (x2, y2) = p2

            if x1 <= offset or x1 >= (w - offset) or y1 <= offset or y1 >= (h - offset) or \
                    x2 <= offset or x2 >= (w - offset) or y2 <= offset or y2 >= (h - offset):
                continue

            window1 = img1[y1 - offset:y1 + 1 + offset, x1 - offset:x1 + 1 + offset]
            window1 = window1.astype(np.float64)
```

```
            window2 = img2[y2 - offset:y2 + 1 + offset, x2 - offset:x2 + 1 + offset]
            window2 = window2.astype(np.float64)

            mean1 = np.mean(window1)
            std1 = np.std(window1)

            mean2 = np.mean(window2)
            std2 = np.std(window2)

            s = 0
            for i in range(0, window_size):
                for j in range(0, window_size):
                    s += (window1[i][j] - mean1) / std1 * (window2[i][j] - mean2) / std2
            s /= (window_size ** 2)

            ncc.append((s, p1, p2))

    ncc.sort(key=lambda x: x[0], reverse=True)
    return ncc


def remove_values_from_list(the_list, val):
    return [value for value in the_list if value[1] != val]


def filter_NCC(NCC_list, len_thresh=0, score_thresh=0):
    NCC_list.sort(key=lambda x: x[0], reverse=True)

    if score_thresh > 0:
        NCC_list = [NCC_list[i] for i in range(len(NCC_list)) if NCC_list[i][0] > score_thresh]

    if 0 < len_thresh <= len(NCC_list):
        NCC_list = NCC_list[:len_thresh]

    return NCC_list


def RANSAC(corres, max_N=10000):
    p = 0.99
    e = 0.6
    s = len(corres)
    N = np.log(1 - p) / (np.log(1 - (1 - e) ** s))
    N = np.int32(N)

    if N > max_N:
        N = max_N

    print("N =", N)

    inliers = []
    inlinecount = 0
    resF = []

    for x in range(0, N):
        pointsImg1 = []
        pointsImg2 = []
        c = 0

        while c < 8:

            r = np.random.randint(low=0, high=len(corres))
            if corres[r][0] not in pointsImg1:
                if corres[r][1] not in pointsImg2:
                    pointsImg1.append(corres[r][0])
```

```
                    pointsImg2.append(corres[r][1])
                    c += 1

        F = fundMat8Points(pointsImg1, pointsImg2)

        counter = 0
        thresh = 0.001
        temp = []

        for p in corres:
            points = np.array(p)
            pl = np.append(points[0], [1])
            pr = np.append(points[1], [1])
            plT = [[pl[0]], [pl[1]], [pl[2]]]
            res = pr @ F @ plT

            if -thresh < res < thresh:
                counter += 1
                temp.append(p)

        if counter > inlinecount:
            inlinecount = counter
            inliers = temp
            resF = F

        if counter == len(corres):
            x = N

    if counter == 0:
        print("ERROR!!!")

    print(len(corres), inlinecount)

    return np.array(inliers), resF


def fundMat8Points(pImg1, pImg2):
    A = []

    pImg1 = np.array(pImg1)
    pImg2 = np.array(pImg2)
    ones = np.ones((8, 1))
    pImg1 = np.hstack((pImg1, ones))
    pImg2 = np.hstack((pImg2, ones))

    ux1 = np.mean(pImg1[:, 0])
    uy1 = np.mean(pImg1[:, 1])
    ux2 = np.mean(pImg2[:, 0])
    uy2 = np.mean(pImg2[:, 1])

    T1 = [[1, 0, -ux1], [0, 1, -uy1], [0, 0, 1]]
    T2 = [[1, 0, -ux2], [0, 1, -uy2], [0, 0, 1]]

    for i in range(0, 8):
        pImg1[i] = pImg1[i] @ T1
        pImg2[i] = pImg2[i] @ T2

    sdvx1 = np.std(pImg1[:, 0])
    sdvy1 = np.std(pImg1[:, 1])
    sdvx2 = np.std(pImg2[:, 0])
    sdvy2 = np.std(pImg2[:, 1])

    T1 = np.array([[1 / sdvx1, 0, 0], [0, 1 / sdvy1, 0], [0, 0, 1]])
    T2 = np.array([[1 / sdvx2, 0, 0], [0, 1 / sdvy2, 0], [0, 0, 1]])
```

```
    for i in range(0, 8):
        pImg1[i] = T1 @ pImg1[i]
        pImg2[i] = T2 @ pImg2[i]

    T1 = [[1 / sdvx1, 0, -ux1], [0, 1 / sdvy1, -uy1], [0, 0, 1]]
    T2 = [[1 / sdvx2, 0, -ux2], [0, 1 / sdvy2, -uy2], [0, 0, 1]]

    for i in range(0, 8):
        A.append([pImg1[i][0] * pImg2[i][0],
                  pImg1[i][0] * pImg2[i][1],
                  pImg1[i][0],
                  pImg1[i][1] * pImg2[i][0],
                  pImg1[i][1] * pImg2[i][1],
                  pImg1[i][1],
                  pImg2[i][0],
                  pImg2[i][1],
                  1])

    _, _, v = np.linalg.svd(A)
    vT = np.transpose(v)
    F = vT[:, 8]
    F = np.reshape(F, (3, 3))

    u, d, v = np.linalg.svd(F)
    d[2] = 0
    F = (u * d[..., None, :]) @ v

    T1 = np.array(T1)
    T2 = np.array(T2)
    F = np.transpose(T2) @ F @ T1
    return F


def display_corner_pairings(img1, img2, lines, y1_shift=0, y2_shift=0):
    w1, h1 = img1.shape[1], img1.shape[0]
    w2, h2 = img2.shape[1], img2.shape[0]

    w_max, h_max = np.max((w1, w2)), np.max((h1, h2))

    out1 = np.zeros((h_max, w_max, 3), dtype=np.uint8)
    out1[y1_shift:h1 + y1_shift, 0:w1] = img1
    out2 = np.zeros((h_max, w_max, 3), dtype=np.uint8)
    out2[y2_shift:h2 + y2_shift, 0:w2] = img2
    combine = np.concatenate((out1, out2), axis=1)

    for p in lines:
        cv2.circle(combine, (p[0][0], p[0][1] + y1_shift), 2, (0, 0, 255), 1)
        cv2.circle(combine, (p[1][0] + w1, p[1][1] + y2_shift), 2, (0, 0, 255), 1)

        hsv = np.array((np.random.rand(), 1, 1))
        rgb = list((np.array(colorsys.hsv_to_rgb(hsv[0], hsv[1], hsv[2])) * 255.0).astype(np.uint16))
        rgb = [int(rgb[0]), int(rgb[1]), int(rgb[2])]
        cv2.line(combine, (p[0][0], p[0][1] + y1_shift), (p[1][0] + w1, p[1][1] + y2_shift), rgb, 1)

    plt.imshow(combine)
    plt.axis('off')
    plt.show()


def calc_disparity(img1, img2, F, c1, c2, SSD_window_size=5):
    w, h = img1.shape[1], img1.shape[0]

    if len(img1.shape) > 2:
        img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY).astype(np.float32)
    if len(img2.shape) > 2:
```

```
        img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY).astype(np.float32)

    x_d = c1[:, 0] - c2[:, 0]
    y_d = c1[:, 1] - c2[:, 1]

    x_d_min = np.min(x_d)
    x_d_max = np.max(x_d)
    y_d_min = np.min(y_d)
    y_d_max = np.max(y_d)
    x_d_mid = (x_d_max + x_d_min) / 2
    y_d_mid = (y_d_max + y_d_min) / 2

    window_size = (x_d_max - x_d_min)+4
    if window_size % 2 == 0:
        window_size += 1

    offset = np.floor(window_size / 2).astype(np.int8)
    SSD_offset = np.floor(SSD_window_size / 2).astype(np.int8)

    print("Window size:", window_size)

    x_disparity = np.full((h, w), -1, dtype=np.float32)
    y_disparity = np.full((h, w), -1, dtype=np.float32)
    disparity = np.full((h, w, 2), -1, dtype=np.float32)

    for i in range((offset+SSD_offset), h-(offset+SSD_offset), 1):
        for j in range((offset+SSD_offset), w-(offset+SSD_offset), 1):
            x_new = int(j - x_d_mid)
            y_new = int(i - y_d_mid)
            # print(x_new, y_new)

            if not ((offset+SSD_offset) <= x_new < (w-(offset+SSD_offset)) and (offset+SSD_offset) <= y_new <= (h-(offset+SSD_offset))):
                continue

            p = [(a, b) for b in range(y_new, y_new + 1) for a in range(x_new - offset, x_new + 1 + offset)]
            SSD = []

            window1 = img1[i - SSD_offset:i + 1 + SSD_offset, j - SSD_offset:j + 1 + SSD_offset]
            for (px, py) in p:
                window2 = img2[py - SSD_offset:py + 1 + SSD_offset, px - SSD_offset:px + 1 + SSD_offset]
                ssd = np.average((window1-window2) ** 2)
                SSD.append((ssd, (px, py)))

            SSD.sort(key=lambda x: x[0], reverse=False)

            x_disparity[i][j] = j - SSD[0][1][0]
            y_disparity[i][j] = i - SSD[0][1][1]

    x_disparity = np.fabs(x_disparity).astype(np.uint8)
    y_disparity = np.fabs(y_disparity).astype(np.uint8)
    disparity = np.fabs(disparity).astype(np.uint8)
    return x_disparity, y_disparity


def main():
    dataset_path = ["cast", "cones", "cast"]

    for path in dataset_path:
        print(path)

        # Open first 2 images in the specified folder and save them in imgset
        imgset = []
        for i in range(2):
            imgset.append(cv2.cvtColor(cv2.imread(path + '/' + os.listdir(path + '/')[i]), cv2.COLOR_BGR2RGB))
```

```
        w, h = imgset[0].shape[1], imgset[0].shape[0]

        # Get corners for the first image
        a = time.time()  # Timing Statements
        ret = Harris(imgset[0], 3, 3, 0.04, step_size=2)
        centroids = non_max_suppression(ret, 9)
        corners = [centroids]

        # Get corners for the second image
        ret = Harris(imgset[1], 3, 3, 0.04, step_size=2)
        centroids = non_max_suppression(ret, 9)
        corners.append(centroids)

        a = time.time()
        NCC_score = NCC(imgset[0], imgset[1], corners[0], corners[1], 5)
        # Filters the NCC list based on some parameters - sort by score with a threshold, min length
        NCC_score = filter_NCC(NCC_score, len_thresh=20)
        print("NCC: ", time.time() - a)  # Timing Statements

        # Display all corners found in red, and top NCC corners in green
        out1 = imgset[0].copy()
        out2 = imgset[1].copy()
        combine = np.concatenate((out1, out2), axis=1)
        for p in corners[0]:
            cv2.circle(combine, (p[0], p[1]), 3, (255, 0, 0), 1)
        for p in corners[1]:
            cv2.circle(combine, (p[0] + w, p[1]), 3, (255, 0, 0), 1)
        for p in NCC_score:
            cv2.circle(combine, (p[1][0], p[1][1]), 3, (0, 255, 0), 1)
            cv2.circle(combine, (p[2][0] + w, p[2][1]), 3, (0, 255, 0), 1)

        plt.imshow(combine)
        plt.axis('off')
        plt.show()

        # Display corner pairings before RANSAC
        p1 = [(p[1], p[2]) for p in NCC_score]
        display_corner_pairings(imgset[0], imgset[1], p1)

        inliers, F = RANSAC(p1)

        # Display corner pairings after RANSAC
        display_corner_pairings(imgset[0], imgset[1], inliers)

        a = time.time()
        print(imgset[0].shape, imgset[1].shape)
        x_disparity, y_disparity = calc_disparity(imgset[0], imgset[1], F, inliers[:, 0, :], inliers[:, 1, :])
        print("Disparity", time.time() - a)  # Timing Statements

        # Save disparity images
        cv2.imwrite(path + "/x.jpg", x_disparity)
        cv2.imwrite(path + "/y.jpg", y_disparity)


def display_disparity():
    for c in range(2):
        if c == 0:
            # Open cones disparity images and crop
            x_disparity = cv2.imread("cones_x.jpg", 0)
            y_disparity = cv2.imread("cones_y.jpg", 0)
            x_disparity = x_disparity[20:354, 55:429].astype(np.float32)
            y_disparity = y_disparity[20:354, 55:429].astype(np.float32)
        else:
            # Open Castle disparity images and crop
            x_disparity = cv2.imread("cast_x.jpg", 0)
```

```
            y_disparity = cv2.imread("cast_y.jpg", 0)
            x_disparity = x_disparity[21:361, 93:554].astype(np.float32)
            y_disparity = y_disparity[21:361, 93:554].astype(np.float32)

        w, h = x_disparity.shape[1], x_disparity.shape[0]

        # Calculate vector disparity image
        disparity = np.zeros((h, w, 2), dtype=np.float32)
        disparity[:, :, 0] = np.sqrt((y_disparity - x_disparity) ** 2)
        disparity[:, :, 1] = (np.arctan2(y_disparity, x_disparity) + np.pi) * 180.0 / np.pi

        # Calculate min and max bounds to normal image to 0-255
        min_x = np.min(x_disparity)
        max_x = np.max(x_disparity)

        min_y = np.min(y_disparity)
        max_y = np.max(y_disparity)

        print(min_x, min_y, max_x, max_y)

        x_disparity = 255.0 * (x_disparity - min_x) / (max_x - min_x)
        y_disparity = 255.0 * (y_disparity - min_y) / (max_y - min_y)

        x_disparity = np.fabs(x_disparity).astype(np.uint8)
        y_disparity = np.fabs(y_disparity).astype(np.uint8)

        RGB = np.zeros((h, w, 3), dtype=np.float32)

        max_mag = np.max(disparity[:, :, 0])
        min_mag = np.min(disparity[:, :, 0])

        max_ang = np.max(disparity[:, :, 1])
        min_ang = np.min(disparity[:, :, 1])

        # Claulate RGB image by taking HSI of the vector disparity
        for i in range(h):
            for j in range(w):
                h = (disparity[i][j][1] - min_ang) / (max_ang - min_ang) * 0.25 + 0.75
                s = (disparity[i][j][0] - min_mag) / (max_mag - min_mag)
                hsv = np.array((h, s, 1))
                rgb = list((np.array(colorsys.hsv_to_rgb(hsv[0], hsv[1], hsv[2])) * 255.0).astype(np.uint16))
                RGB[i][j] = [int(rgb[0]), int(rgb[1]), int(rgb[2])]

        RGB = RGB.astype(np.uint8)

        plt.imshow(x_disparity, cmap='gray')
        plt.axis('off')
        plt.show()

        plt.imshow(y_disparity, cmap='gray')
        plt.axis('off')
        plt.show()

        plt.imshow(RGB)
        plt.axis('off')
        plt.show()


if __name__ == "__main__":
    main()
    display_disparity()
```