Computer Vision 1

Stereo Vision Report

Department of EECE, Northeastern University

Taoran Liu and Jianpeng Chen

Apr 14, 2023

1 Abstract

Our project can be divided in two six steps. The first part uses Harris corner detection on the left and right images, and extracts 100 corner points as feature points. The second step uses the NCC feature matching algorithm to find matching feature points in the two images. In the third part, for each pair of matching feature points, randomly select eight pairs of points to calculate the fundamental matrix, and then use the Ransac algorithm to remove outliers to calculate the optimal f matrix. The fourth step is to solve the epipolarline for each pixel according to the F matrix. The fifth step is to perform NCC matching for each pixel along the direction of the epipolarline to find the corresponding point. The sixth step is to calculate the disparity in the horizontal and vertical directions for each pixel point and its matching point, and generate a disparity image and a disparity vector map.

2 Algorithm Description

2.1 Conner detection

2.1.1 Introduction

Corner is an important feature of an image, which plays an important role in the understanding and analysis of image graphics. While retaining the important features of the image, the corner point can effectively reduce the amount of information data, make the information content very high, effectively improve the calculation speed, and facilitate reliable image matching and real-time processing.

2.1.2 The function of extracting point features

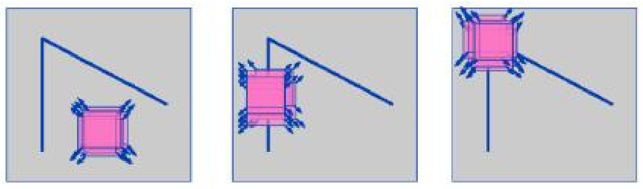
The point feature of the image is the basis of many computer vision algorithms: using feature points to represent the content of the image has many uses in the directions of moving target tracking, object recognition, image registration, panoramic image stitching and 3D reconstruction.

There is an important class of point features: corner points. Corner points: Points where the local window moves in all directions, resulting in significant changes, and points where the local curvature of the image changes suddenly Typical corner detection algorithms: Harris corner detection, CSS corner detection, etc.

2.1.3 Harris Basic idea

Basic idea: observe image features from a small window in the local image.

Corner definition: The movement of the window to any direction will cause obvious changes in the gray level of the image.



2.1.4 Harris corner detection: mathematical description

Translate the image window in $[u,v]$ to produce grayscale changes $E(u,v)$

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+u) - I(x,y)]^2$$

$$E(u,v) = \sum_{x,y} w(x,y)\left[I_x u + I_y v + O(u^2,v^2)\right]^2$$

$$[I_x u + I_y v]^2 = [u,v]\begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}$$

Therefore, for a small local movement $[u,v]$, the following expression can be approximated:

$$E(u,v) \cong [u,\ v] \quad M \quad \begin{bmatrix} u \\ v \end{bmatrix}$$

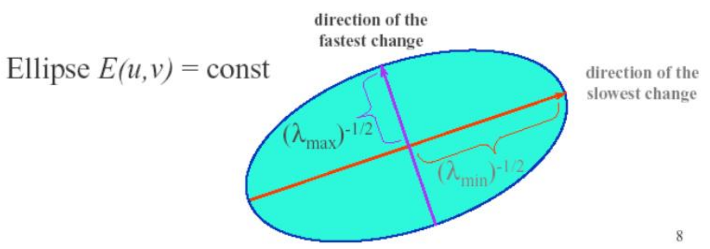
Among them, M is an 2X2 matrix, which can be obtained by the derivative of the image:

$$M = \sum_{x,y} w(x,y)\begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

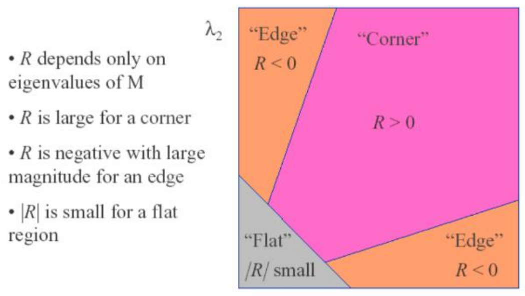
Image changes due to window movement: eigenvalue analysis of real symmetric matrices

$$E(u,v) \cong [u,\ v] \quad M \quad \begin{bmatrix} u \\ v \end{bmatrix}$$

Among them, the eigenvalues of λmax λmin


direction of the fastest change
Ellipse E(u,v) = const
direction of the slowest change
(λmax)^-1/2
(λmin)^-1/2
8

$$R = \det M - k\ \mathrm{trace}^2(M)$$


• R depends only on eigenvalues of M
• R is large for a corner
• R is negative with large magnitude for an edge
• $|R|$ is small for a flat region
λ_2
"Edge" $R < 0$
"Corner"
$R > 0$
"Flat" $|R|$ small
"Edge" $R < 0$

2.1.5 Conclusion and Program Application

The Harris corner detector is divided into three steps: gradient calculation, matrix formation and eigenvalue calculation.

First, smooth gradients in the x and y directions are computed (using first derivative of the Gaussian) to detect corners in a given grayscale image I(x,y), given by:

$$g_x(x,y) = \frac{-x}{2\pi\tau_g^4}\exp\left(-\frac{x^2+y^2}{2\tau_g^2}\right)$$

$$g_y(x,y) = \frac{-y}{2\pi\tau_g^4}\exp\left(-\frac{x^2+y^2}{2\tau_g^2}\right)$$

In our code, we used fx = [5 0 -5;8 0 -8;5 0 -5] and fy = [5 8 5;0 0 0;-5 -8 -5] as first derivative of the Gaussian mask.

Calculate the smooth gradient I(x,y) of the image as:

$$I_x = g_x(x,y) \otimes I(x,y)$$

$$I_y = g_y(x,y) \otimes I(x,y)$$

Calculate the M matrix corresponding to the picture:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Use the det and trace of the M matrix to calculate the R value corresponding to each pixel, and use the regional maximum value as the corner pixel coordinates

$$R = \det A - k[\operatorname{trace}(A)]^2$$

After we got R value for every pixel, we set use a window with 3X3 size to find local maximum R value as identified corners.

In our code, we didn’t use any constant value of R as threshold to judge whether one result is a good conner. We sort all the results from largest to smallest, and use largest 100 R value as the best output conner set.

2.2 Feature matching

2.2.1 Concepts of NCC

NCC (normalized cross correlation) algorithm, normalized cross-correlation matching method, is a matching method based on image grayscale information. It is a common image processing method to compare the similarity of two images.

There are three main methods of image matching: grayscale-based, feature-based, and transform domain-based.

The NCC algorithm can effectively reduce the influence of light on the image comparison results. Moreover, the final result of NCC is between 0 and 1, so it is particularly easy to quantify the comparison results, as long as a threshold is given to judge whether the result is good or bad. The traditional NCC comparison method is time-consuming. Although it can be optimized by adjusting the window size and the step rectangle of each detection, it cannot meet the real-time requirements for industrial production detection. Pre-calculation is realized by integrating images, and the template image is compared with the produced one. Subtle differences between electronic versions can help companies improve product quality, reduce the rate of defective products, and control quality.

2.2.2 Basic principle of NCC algorithm

Construct an n×n neighborhood as a matching window for any pixel (px, py) in the original image. Then, for the target pixel position (px+d,py), a matching window of size n×n is also constructed to measure the similarity between the two windows. Using following equation.

$$\hat{f} = \frac{f}{||f||} = \frac{f}{\sqrt{\sum_{[i,j]\in R} f^2(i,j)}}$$

$$\hat{g} = \frac{g}{||g||} = \frac{g}{\sqrt{\sum_{[i,j]\in R} g^2(i,j)}}$$

$$N_{fg} = C_{\hat{f}\hat{g}} = \sum_{[i,j]\in R} \hat{f}(i,j)\hat{g}(i,j)$$

$$\hat{f} = \frac{f}{||f||} \quad \hat{g} = \frac{g}{||g||}$$

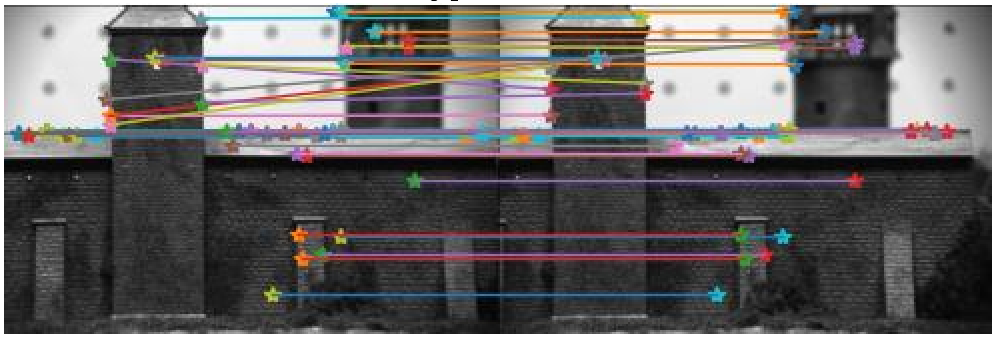
2.2.3 Program Application

In Harris conner detection part, we find 100 corner with largest R value in each image. In this step, we used these conners as feature points to do NCC matching algorism.

The main idea of our matching function is to do one-way searching for matching points from one image to another. And in main function we run matching function twice times, used first-time output point set as input run matching function again inversely. It will make sure every pair of matches is bidirectional.

In the matching function, first, we pick one conner pixel from image1 and use it's nXn neighbor as a matching window f(i, j). Then use this window and each corner point in picture 2 and the corresponding nXn neighbor g(i, j) to calculate the NCC value, sort all the NCC result from largest to smallest, and use the g(i,j) with largest NCC value as the matching pair with f(i ,j). Last repeat this process for each conner pixels in image1, get all matching pairs from image1 to image2.

Here is the results of the matching pairs.



2.3 Fundamental matrix

The fundamental matrix is a mathematical representation of the epipolar geometry of a pair of images. It describes the relationship between corresponding points in two images taken from different viewpoints using a calibrated camera.

To calculate Fundamental matrix, we used eight-point algorithm. First constructs the matrix A using the coordinates of the corresponding points, and then computes the singular value decomposition (SVD) of A to obtain the least squares solution for the fundamental matrix F. The function then constrains F by setting the smallest singular value of F to zero, and returns the normalized fundamental matrix.

Our algorithm for computing the fundamental matrix involves the following steps:

Construct a coefficient matrix A of size Nx9, where each row of A is the outer product (also known as the Kronecker product) of the corresponding rows of xs and xss.Compute the singular value decomposition (SVD) of A, which provides the basis for the subsequent steps.

Approximate the fundamental matrix Fa by taking the last column of V, which corresponds to the smallest singular value. This approximation may be regularized to improve the accuracy of the computed fundamental matrix.

Compute the SVD of Fa, which provides the singular values and singular vectors of Fa.
Compute the algebraically best fundamental matrix F by setting the smallest singular value of Da to zero and then reconstructing F using the singular vectors Ua and Va.

Finally, the function returns the computed fundamental matrix F.

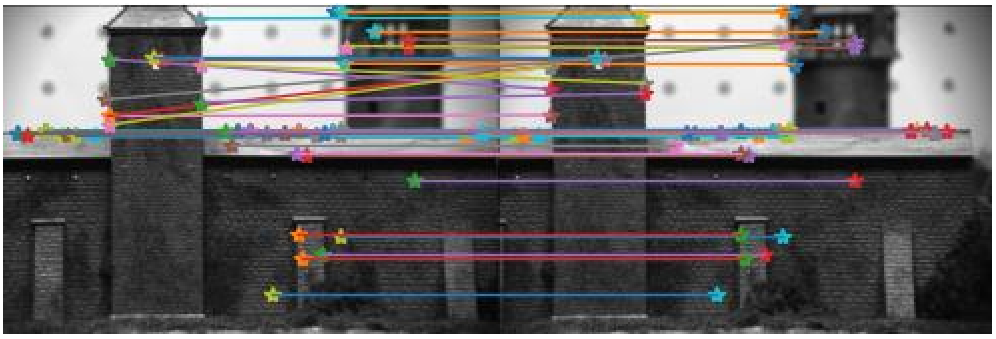
2.4 Ransac

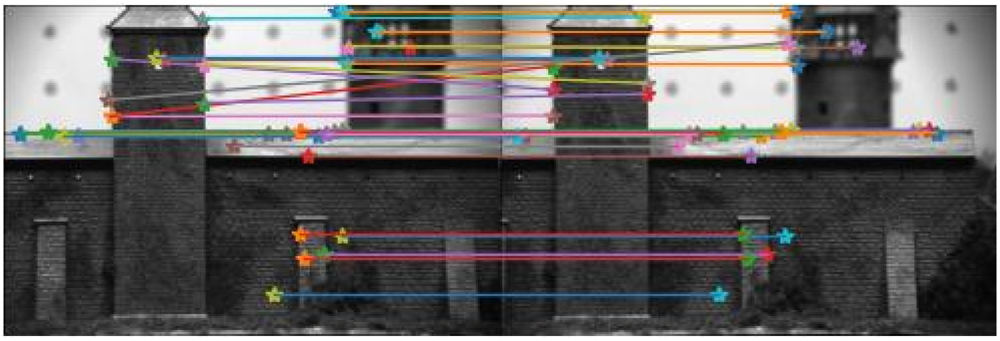
The function first sets up some parameters for the RANSAC algorithm, such as the number of iterations to run (ransactimes), the number of points to randomly select for each iteration (ransanc_num), and a variable to keep track of the best fundamental matrix found so far (F_result).

The function then runs the RANSAC algorithm for ransactimes iterations. In each iteration, ransanc_num corresponding points are randomly selected from the ncc_match list, and the fundamental matrix is computed using the findFmatrix function. The quality of the fundamental matrix is then evaluated by computing the algebraic error between the corresponding points and the epipolar lines. The fundamental matrix with the lowest error is kept as the current best result.

Finally, the function returns the best fundamental matrix found during the RANSAC iterations.

Here is the contrast of all pairings and inliers pairings results.





2.5 Calculate epipolar line Using Fundamental Matrix

Given a point (x, y) in one image, and the fundamental matrix F between the two images, we first compute the epipolar line corresponding to the point using the formula:

e_line = F * [x, y, 1]^T

Next, we want to calculate the points on this line that intersect with the other image. To do this, we take a set of 'linepoints' equally spaced along the width of the other image, and compute the y-coordinates of these points using the equation of the epipolar line:

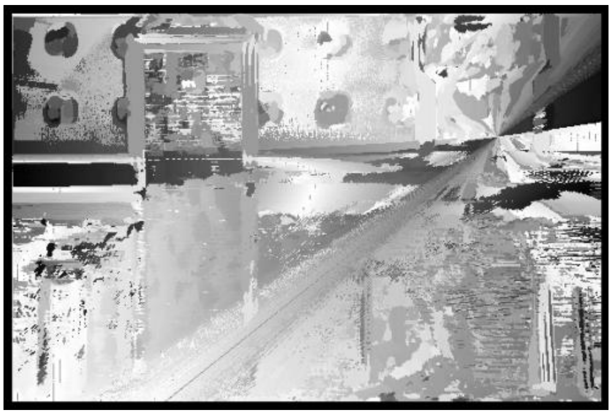
y = (-e_line[2] - e_line[0]*x) / e_line[1].

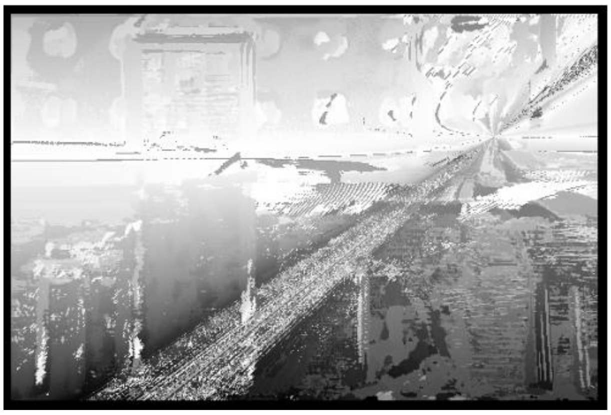
Here, x is the x-coordinate of each linepoint. We then filter out the linepoints that lie outside the height of the other image. Finally, we return the x and y coordinates of the linepoints that lie within the height of the other image.

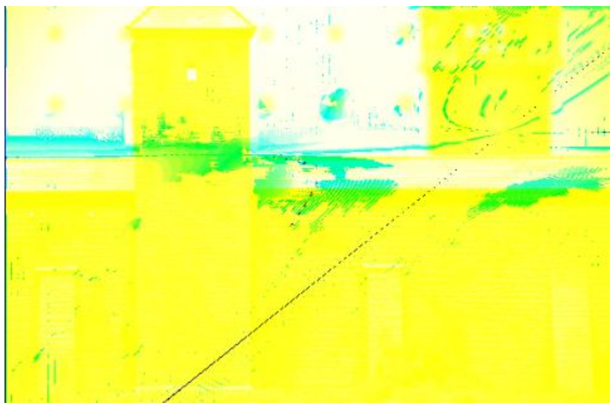
2.6 Compute the dense disparity maps using epipolar geometry and normalized cross correlation matching.

We first convert the full image to grayscale and compute the value channel of its HSV representation and initialize horizontal and vertical disparity maps and a disparity vector map. For each pixel in the left image, compute its corresponding epipolar line in the right image using the fundamental matrix F and the epipole number. Extract a small NCC window around the pixel in the left image and search for a matching window along the epipolar line in the right image using NCC. Compute the horizontal and vertical displacement between the matched pixels, and store them in the corresponding disparity maps and compute the hue and saturation of the disparity vector using the horizontal and vertical displacement values.

Finally we just need to normalize the disparity maps and the disparity vector map.


A) horizontal disparity


B) vertical disparity


C) disparity vector using color

Appendix

code

```
import numpy as np
import cv2
from scipy.ndimage import filters
from scipy import ndimage
import matplotlib.pyplot as plt
import random
from scipy import linalg
import sys
import math

#########################
NCC_window_size = 8
epipole_number = 100
#########################

def harris_corner_detector(image, threshold, min_distance):
    sigma = 1.4
    # first compute derivatives
    img_x = np.zeros(image.shape)
    filters.gaussian_filter(image, (sigma, sigma), (0, 1), img_x)
    img_y = np.zeros(image.shape)
    filters.gaussian_filter(image, (sigma, sigma), (1, 0), img_y)

    # compute Wxx, Wxy, Wyy
    Wxx = filters.gaussian_filter(img_x * img_x, sigma)
    Wxy = filters.gaussian_filter(img_x * img_y, sigma)
    Wyy = filters.gaussian_filter(img_y * img_y, sigma)

    Wdet = Wxx * Wyy - Wxy ** 2
    Wtr = Wxx + Wyy
    harrisim = Wdet / Wtr

    corner_threshold = harrisim.max() * threshold
    harrisim_t = (harrisim > corner_threshold) * 1
    coords = np.array(harrisim_t.nonzero()).T
    candidate_values = [harrisim[c[0], c[1]] for c in coords]
    index = np.argsort(candidate_values)
    allowed_locations = np.zeros(harrisim.shape)

    # non-maximum suppression
    allowed_locations[min_distance:-min_distance, min_distance:-min_distance] = 1
    results_coord = []
    for i in index:
        if allowed_locations[coords[i, 0], coords[i, 1]] == 1:
            results_coord.append(coords[i])
            allowed_locations[(coords[i, 0] - min_distance):(coords[i, 0] + min_distance),
            (coords[i, 1] - min_distance):(coords[i, 1] + min_distance)] = 0
    return results_coord

def Calculate_NCC(window_left, window_right):
    N_L = sum(sum(window_left**2))**(1/2)
    N_R = sum(sum(window_right**2))**(1/2)
    ncc = sum(sum((window_left/N_L)*(window_right/N_R)))
    return ncc

def NCC_match_points(gray_left,gray_right,leftcorner,rightcorner,windowsize,threshold):
    m = len(leftcorner)
    n = len(rightcorner)
    gray_left = np.array(gray_left,dtype='uint32')
    gray_right = np.array(gray_right,dtype='uint32')
    NCC_value = np.zeros((m,n))
    for i in range (m):
        leftwindow = gray_left[(leftcorner[i][0] - windowsize):(leftcorner[i][0] + windowsize + 1),
(leftcorner[i][1] - windowsize):(leftcorner[i][1] + windowsize + 1)]
        for j in range(n):
            rightwindow = gray_right[(rightcorner[j][0]-windowsize):(rightcorner[j][0]+windowsize+1),
(rightcorner[j][1]-windowsize):(rightcorner[j][1]+windowsize+1)]
            NCC_value[i,j] = Calculate_NCC(leftwindow, rightwindow)
    ncc = NCC_value.tolist()
    matches = []
    for i in range (m*n):
        m_max = []
        for j in range(m):
            m_max.extend([max(ncc[j])])
        temp = max(m_max)
        if temp < threshold:
            break
        for a in range (m):
            for b in range(n):
                if ncc[a][b]==temp:
                    matches += [[a,b]]
                    ncc[a][b] = 0
                    a=m+1
                    b=n+1
```

```python
                    break
    return matches

def draw_match_points(image_left, image_right, left_keypoints, right_keypoints, matches, linenumber):
    """
    Draws keypoints and their matches between two images.

    Args:
    - image_left (numpy array): The first image
    - image_right (numpy array): The second image
    - left_keypoints (list): Keypoint list of the left image
    - right_keypoints (list): Keypoint list of the right image
    - matches (list): List of matches between keypoints
    - linenumber (int): Number of matches to be drawn

    Returns:
    - None
    """
    # Create an empty image to hold the combined images
    width = image_left.shape[1] + image_right.shape[1]
    height = max(image_left.shape[0], image_right.shape[0])
    image = np.zeros([height, width, 3], dtype='uint8')

    # Copy the first image to the left side of the combined image
    for i in range(0, width):
        for j in range(0, height):
            if i < ((image.shape[1] / 2) - 1):
                image[j][i] = image_left[j][i]
                x = i
            else:
                image[j][i] = image_right[j][i - x - 2]

    # Draw the matches on the combined image
    plt.ion()
    plt.imshow(image, cmap='gray')
    for a in range(0, linenumber):
        temp_l = matches[a][0]
        temp_r = matches[a][1]
        plt.plot(left_keypoints[temp_l][1], left_keypoints[temp_l][0], '*')
        plt.plot(right_keypoints[temp_r][1] + image_left.shape[1], right_keypoints[temp_r][0], '*')
        plt.plot([left_keypoints[temp_l][1], right_keypoints[temp_r][1] + image_left.shape[1]],
                 [left_keypoints[temp_l][0], right_keypoints[temp_r][0]], linewidth=1)
    plt.ioff()
    plt.axis('off')
    plt.show()

    return 0

def rgbtohsv(img):
    m, n, k = img.shape
    r, g, b = cv2.split(img)
    r, g, b = r / 255.0, g / 255.0, b / 255.0
    #hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    H = np.zeros((m, n), np.float32)
    S = np.zeros((m, n), np.float32)
    V = np.zeros((m, n), np.float32)
    HSV = np.zeros((m, n, 3), np.float32)
    for i in range(0, m):
        for j in range(0, n):
            mx = max((b[i, j], g[i, j], r[i, j]))
            mn = min((b[i, j], g[i, j], r[i, j]))
            V[i, j] = mx
            if V[i, j] == 0:
                S[i, j] = 0
            else:
                S[i, j] = (V[i, j] - mn) / V[i, j]
            if mx == mn:
                H[i, j] = 0
            elif V[i, j] == r[i, j]:
                if g[i, j] >= b[i, j]:
                    H[i, j] = (60 * ((g[i, j]) - b[i, j]) / (V[i, j] - mn))
                else:
                    H[i, j] = (60 * ((g[i, j]) - b[i, j]) / (V[i, j] - mn)) + 360
            elif V[i, j] == g[i, j]:
                H[i, j] = 60 * ((b[i, j]) - r[i, j]) / (V[i, j] - mn) + 120
            elif V[i, j] == b[i, j]:
                H[i, j] = 60 * ((r[i, j]) - g[i, j]) / (V[i, j] - mn) + 240
            H[i, j] = H[i, j] / 2
    HSV[:, :, 0] = H[:, :]
    HSV[:, :, 1] = S[:, :]
    HSV[:, :, 2] = V[:, :]
    return HSV

def draw_inliers(image1, image2, cor_l, cor_r, ncc_match, F):
    threshold = 1
    inliers = []
    draw_p = min(len(ncc_match), 40)
    for c in range(draw_p):
        p_l = np.hstack([cor_l[ncc_match[c][0]], 1])
        p_r = np.hstack([cor_r[ncc_match[c][1]], 1])

        if np.dot(np.dot(p_l, F), p_r) < threshold:
            inliers.append(ncc_match[c])

    print(len(inliers))
    draw_match_points(image1, image2, cor_l, cor_r, inliers, len(inliers))
    return 0

def findFmatrix(x1,x2,n):
    x1=np.array(x1)
    x2=np.array(x2)
    A = np.zeros((n, 9))
    for i in range(n):
        A[i] = [x1[i, 1] * x2[i, 1], x1[i, 1] * x2[i, 0], x1[i, 1],
                x1[i, 0] * x2[i, 1], x1[i, 0] * x2[i, 0], x1[i, 0],
                x2[i, 1], x2[i, 0], 1]
    # compute linear least square solution
    U, S, V = linalg.svd(A)
    F = V[-1].reshape(3, 3)
    # constrain F
    U, S, V = linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), V))
    return F / F[2, 2]

def Ransac_f_matrix(cor_l,cor_r,ncc_match):
    ransactimes = 1000
    ACC_result=sys.maxsize
    F_result=None
    ransanc_num=50
    for a in range(ransactimes):
        p1=[]
        p2=[]
        for b in range(ransanc_num):
          temp=random.randint(0, len(ncc_match)-1)
          p1.extend([cor_l[ncc_match[temp][0]]])
          p2.extend([cor_r[ncc_match[temp][1]]])
        F=findFmatrix(p1,p2,ransanc_num)
        temp1=0
        for c in range(len(ncc_match)):
            p_l=[]
            p_l.extend([cor_l[ncc_match[c][0]]])
            p_l=np.append(p_l,1)
            p_r = []
            p_r.extend([cor_r[ncc_match[c][1]]])
            p_r = np.append(p_r, 1)
            temp1+=abs(np.dot(np.dot(p_l.T,F),p_r))
        if temp1<ACC_result:
            ACC_result=temp1
            F_result=F
            #print([a,ACC_result])
    return F_result

def calculateepipole(point,F,m,n,linepoints):
    e_line=np.dot(F,point.T)
    # print(e_line)
    t = np.linspace(NCC_window_size, n-NCC_window_size-1, linepoints)
    # print(t)
    lt = np.array([(e_line[2] + e_line[0] * tt) / (-e_line[1]) for tt in t])
    # print(lt)
    ndx = (lt >= 0) & (lt < m)
    # print(ndx)
    t = np.reshape(t, (linepoints, 1))
    return t[ndx],lt[ndx]

def Compute_dense_map(leftimage,rightimage,F,fullimage):
    v=rgbtohsv(fullimage)[:,:,2]
    m, n= leftimage.shape
    horizontal_map=np.zeros((m,n))
    vertical_map=np.zeros((m,n))
    vector_map=np.zeros((m,n,3))
    leftimage = np.array(leftimage, dtype='uint32')
    rightimage = np.array(rightimage, dtype='uint32')
    temp_right=np.zeros((m+2*NCC_window_size,n))
    temp_right[NCC_window_size:-NCC_window_size,:]=rightimage
    max_h=0
    max_v=0
    max_sat=0
    # calculate epipole line
    for i in range(NCC_window_size,m-NCC_window_size-1):
    # for i in range(30,80):
        for j in range(NCC_window_size,n-NCC_window_size-1):
        # for j in range(30,80):
            line_x,line_y=calculateepipole(np.array([j,i,1]),F,m,n,epipole_number)
            # print(line_x, line_y)
            leftwindow =
leftimage[(i-NCC_window_size):(i+NCC_window_size+1),(j-NCC_window_size):(j+NCC_window_size+1)]
            # search NCC on the line
            temp_ncc=0
            temp_match=[]
            for a in range(len(line_x)):
               pointx=int(line_x[a])
               pointy=int(line_y[a])
               for b in range(-3, 4):
                   rightwindow =
temp_right[(pointy):(pointy+2*NCC_window_size+1),(pointx-NCC_window_size):(pointx+NCC_window_si
ze+1)]
                   temp = Calculate_NCC(leftwindow,rightwindow)
                   if temp>temp_ncc and temp>0:
                          temp_ncc=temp
                          temp_match=[pointy,pointx]

            if temp_match!=[]:
                dif_x=abs(j-temp_match[1])
                dif_y=abs(i-temp_match[0])
                horizontal_map[i][j]=dif_x
                vertical_map[i][j]=dif_y
                hue=0
                if dif_x!=0:
                   hue=math.degrees(math.atan(dif_y/dif_x))
                if hue<0:
                    hue=360-abs(hue)
                sat=(dif_x**2+dif_y**2)**(1/2)
                vector_map[i][j][0] = hue
                vector_map[i][j][1] = sat
                if dif_x>max_h:
                   max_h=dif_x
                if dif_y>max_v:
                    max_v=dif_y
                if sat>max_sat:
                    max_sat=sat
        print(i)
    horizontal_map*=(255/max_h)
    vertical_map*=(255/max_v)
    #print(vertical_map[:0])
    for i in range(NCC_window_size,m-NCC_window_size-1):
        for j in range(NCC_window_size,n-NCC_window_size-1):
            if horizontal_map[i][j] == 0:
```

```
temp=horizontal_map[i-1][j-1]+horizontal_map[i-1][j]+horizontal_map[i-1][j+1]+horizontal_map[i][j-1]+horizontal_map[i][j+1]+horizontal_map[i+1][j-1]+horizontal_map[i+1][j]+horizontal_map[i+1][j+1]
                temp/=8
                if temp>=10:
                    horizontal_map[i][j]=int(255-temp)
            else:
                horizontal_map[i][j]=int(255-horizontal_map[i][j])
            if vertical_map[i][j] == 0:

temp=vertical_map[i-1][j-1]+vertical_map[i-1][j]+vertical_map[i-1][j+1]+vertical_map[i][j-1]+vertical_map[i][j+1]+vertical_map[i+1][j-1]+vertical_map[i+1][j]+vertical_map[i+1][j+1]
                temp/=8
                if temp>=10:
                    vertical_map[i][j]=int(255-temp)
            else:
                vertical_map[i][j]=int(255-vertical_map[i][j])
    vector_map[:,:,2]=v
    vector_map[:,:,1]*=(255/max_sat)

    # horizontal disparity component
    plt.ion()
    plt.imshow(horizontal_map,cmap ='gray')
    plt.ioff()
    plt.axis('off')
    plt.show()
    # vertical disparity component
    plt.ion()
    plt.imshow(vertical_map, cmap='gray')
    plt.ioff()
    plt.axis('off')
    plt.show()
    # disparity vector using color
    plt.ion()
    plt.imshow(vector_map, cmap='hsv')
    plt.ioff()
    plt.axis('off')
    plt.show()
    return 0


if __name__ == '__main__':
    # read images and convert it into gray map
    leftimage = cv2.imread("cast-left-1.jpeg")
    rightimage = cv2.imread("cast-right-1.jpeg")
    gray_left = cv2.cvtColor(leftimage, cv2.COLOR_BGR2GRAY)
    gray_right = cv2.cvtColor(rightimage, cv2.COLOR_BGR2GRAY)

    # Harris corner detection
    left_corners = harris_corner_detector(gray_left, 0.03, 8)
    right_corners = harris_corner_detector(gray_right, 0.03, 8)
    ncc_match_points = NCC_match_points(gray_left, gray_right, left_corners, right_corners,
NCC_window_size, 0.99)

    print([len(left_corners), len(right_corners)])
    print(len(ncc_match_points))

    # visualize the all correspondenc results
    draw_match_points(gray_left, gray_right, left_corners, right_corners, ncc_match_points,
len(ncc_match_points))

    # compute F matrix with RANSAC with 8 pts
    F = Ransac_f_matrix(left_corners, right_corners, ncc_match_points)
    print(f'Fundanmental Matrix is: {F}')
    draw_inliers(leftimage, rightimage, left_corners, right_corners, ncc_match_points, F)

    # compute dense disparity map and show the result
    Compute_dense_map(gray_left, gray_right, F, leftimage)
```