HW4

February 18, 2023

1 CSE 252B: Computer Vision II, Winter 2023 – Assignment 4

1.0.1 Instructor: Ben Ochoa

1.0.2 Due: Wednesday, March 8, 2023, 11:59 PM

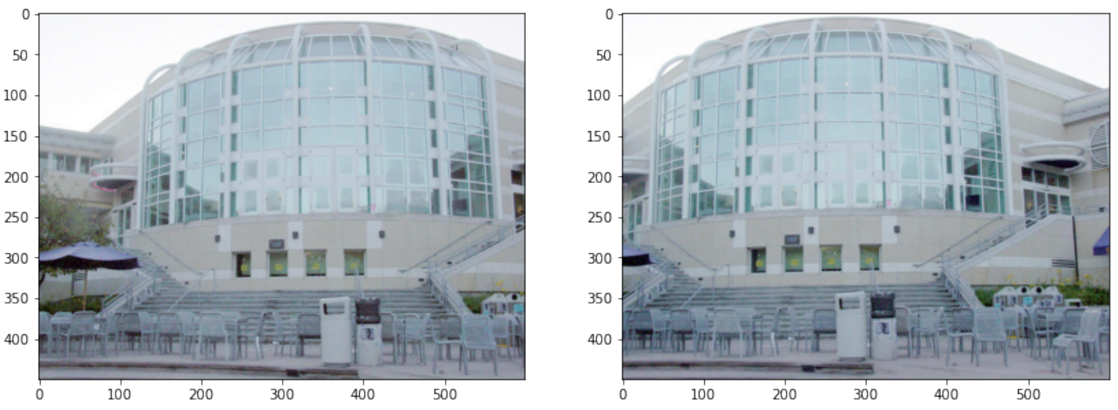
1.1 Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- All solutions must be written in this notebook
- Math problems must be done in Markdown/LATEX.
- You must show your work and describe your solution.
- Programming aspects of this assignment must be completed using Python in this notebook.
- This notebook contains skeleton code, which should not be modified (this is important for standardization to facilate effeciant grading).
- You may use python packages for basic linear algebra, but you may not use packages that directly solve the problem. If you are uncertain about using a specific package, then please ask the instructional staff whether or not it is allowable.
- You must submit this notebook exported as a pdf. You must also submit this notebook as an .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- It is highly recommended that you begin working on this assignment early.

[1]:
```
%matplotlib inline
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import time

# open the input images
I1 = np.array(Image.open('price_center20.JPG'), dtype='float')/255.
I2 = np.array(Image.open('price_center21.JPG'), dtype='float')/255.

# Display the input images
plt.figure(figsize=(14,8))
plt.subplot(1,2,1)
plt.imshow(I1)
plt.subplot(1,2,2)
plt.imshow(I2)
plt.show()
```



1.2 Problem 1 (Programming): Feature detection (20 points)

Download input data from the course website. The file price_center20.jpeg contains image 1 and the file price_center21.jpeg contains image 2.

For each input image, calculate an image where each pixel value is the minor eigenvalue of the gradient matrix

$$N = \begin{bmatrix} \sum_w I_x^2 & \sum_w I_x I_y \\ \sum_w I_x I_y & \sum_w I_y^2 \end{bmatrix}$$

where w is the window about the pixel, and I_x and I_y are the gradient images in the x and y direction, respectively. Calculate the gradient images using the fivepoint central difference operator. Set resulting values that are below a specified threshold value to zero (hint: calculating the mean instead of the sum in N allows for adjusting the size of the window without changing the threshold value). Apply an operation that suppresses (sets to 0) local (i.e., about a window) nonmaximum pixel values in the minor eigenvalue image. Vary these parameters such that 600–650 features are detected in each image. For resulting nonzero pixel values, determine the subpixel feature coordinate using the Forstner corner point operator.

You may use scipy.signal.convolve to perform convolution operation and scipy.ndimage.maximum_filter for NMS operation.

You may either directly use the color images for feature detection, or use the color to grayscale mapping $Y = 0.21263903 * R + 0.71516871 * G + 0.072192319 * B$ to convert the images to grayscale first.

Report your final values for:

- the size of the feature detection window (i.e. the size of the window used to calculate the elements in the gradient matrix N)
- the minor eigenvalue threshold value
- the size of the local nonmaximum suppression window
- the resulting number of features detected (i.e. corners) in each image.

Display figures for:

- original images with detected features

A typical implementation takes around 30 seconds to run. If yours takes more than 60 seconds, you may lose points.

[ ]:
```python
from scipy import signal
from scipy import ndimage

def image_gradient(I):
    # inputs:
    # I is the input image (may be mxn for Grayscale or mxnx3 for RGB)
    #
    # outputs:
    # Ix is the derivative of the magnitude of the image w.r.t. x
    # Iy is the derivative of the magnitude of the image w.r.t. y

    m, n = I.shape[:2]

    """your code here"""


    return Ix, Iy


def minor_eigenvalue_image(Ix, Iy, w):
    # Calculate the minor eigenvalue image J
    #
    # inputs:
    # Ix is the derivative of the magnitude of the image w.r.t. x
    # Iy is the derivative of the magnitude of the image w.r.t. y
    # w is the size of the window used to compute the gradient matrix N
    #
    # outputs:
    # J0 is the mxn minor eigenvalue image of N before thresholding

    m, n = Ix.shape[:2]
    J0 = np.zeros((m,n))

    #Calculate your minor eigenvalue image J0.
    """your code here"""
    return J0

def nms(J, w_nms):
    # Apply nonmaximum supression to J using window w_nms
    #
    # inputs:
    # J is the minor eigenvalue image input image after thresholding
    # w_nms is the size of the local nonmaximum suppression window
    #
    # outputs:
    # J2 is the mxn resulting image after applying nonmaximum suppression
    #

    J2 = J.copy()
    """your code here"""


    return J2

def forstner_corner_detector(Ix, Iy, w, t, w_nms):
    # Calculate the minor eigenvalue image J
    # Threshold J
    # Run non-maxima suppression on the thresholded J
    # Gather the coordinates of the nonzero pixels in J
    # Then compute the sub pixel location of each point using the Forstner␣
 ↪operator
    #
    # inputs:
    # Ix is the derivative of the magnitude of the image w.r.t. x
    # Iy is the derivative of the magnitude of the image w.r.t. y
    # w is the size of the window used to compute the gradient matrix N
    # t is the minor eigenvalue threshold
    # w_nms is the size of the local nonmaximum suppression window
    #
    # outputs:
    # C is the number of corners detected in each image
    # pts is the 2xC array of coordinates of subpixel accurate corners
    #     found using the Forstner corner detector
    # J0 is the mxn minor eigenvalue image of N before thresholding
    # J1 is the mxn minor eigenvalue image of N after thresholding
    # J2 is the mxn minor eigenvalue image of N after thresholding and NMS

    m, n = Ix.shape[:2]
    J0 = np.zeros((m,n))
    J1 = np.zeros((m,n))
    #Calculate your minor eigenvalue image J0 and its thresholded version J1.
    """your code here"""


    #Run non-maxima suppression on your thresholded minor eigenvalue image.
    J2 = nms(J1, w_nms)

    #Detect corners.
    """your code here"""


    return C, pts, J0, J1, J2


# feature detection
def run_feature_detection(I, w, t, w_nms):
    Ix, Iy = image_gradient(I)
    C, pts, J0, J1, J2 = forstner_corner_detector(Ix, Iy, w, t, w_nms)
    return C, pts, J0, J1, J2
```

[ ]:
```
# ImageGradient() unit test
def check_values(I, target):
    eps = 1e-8  # Floating point error threshold
    I = I[2:-2, 2:-2]  # Ignore border values
    valid = np.all((I < target + eps) & (I > target - eps))
    print(f'Image is all equal to {target} +/- {eps}: {valid}')

def gray_to_RGB(I):
    h, w = I.shape
    I = np.expand_dims(I, axis=-1)
    return np.broadcast_to(I, (h, w, 3))

rampx = np.array(Image.open('rampx.png'), dtype='float')
rampy = np.array(Image.open('rampy.png'), dtype='float')

# If you are using grayscale images in ImageGradient(), comment out these lines
rampx = gray_to_RGB(rampx)
rampy = gray_to_RGB(rampy)

# rampx_Ix should be all ones, rampx_Iy should be all zeros (to floating point␣
 ↪error)
rampx_Ix, rampx_Iy = image_gradient(rampx)
check_values(rampx_Ix, 1)
check_values(rampx_Iy, 0)
```

```
# rampy_Ix should be all zeros, rampx_Iy should be all ones (to floating point␣
↪error)
rampy_Ix, rampy_Iy = image_gradient(rampy)
check_values(rampy_Ix, 0)
check_values(rampy_Iy, 1)
```

[ ]:
```
# input images
I1 = np.array(Image.open('price_center20.JPG'), dtype='float')/255.
I2 = np.array(Image.open('price_center21.JPG'), dtype='float')/255.

# parameters to tune
w = 1
t = 1
w_nms = 1

tic = time.time()
# run feature detection algorithm on input images
C1, pts1, J1_0, J1_1, J1_2 = run_feature_detection(I1, w, t, w_nms)
C2, pts2, J2_0, J2_1, J2_2 = run_feature_detection(I2, w, t, w_nms)
toc = time.time() - tic

print('took %f secs'%toc)

# display results
plt.figure(figsize=(14,24))

# show pre-thresholded minor eigenvalue images
plt.subplot(3,2,1)
plt.imshow(J1_0, cmap='gray')
plt.title('pre-thresholded minor eigenvalue image')
plt.subplot(3,2,2)
plt.imshow(J2_0, cmap='gray')
plt.title('pre-thresholded minor eigenvalue image')

# show thresholded minor eigenvalue images
plt.subplot(3,2,3)
plt.imshow(J1_1, cmap='gray')
plt.title('thresholded minor eigenvalue image')
plt.subplot(3,2,4)
plt.imshow(J2_1, cmap='gray')
plt.title('thresholded minor eigenvalue image')

# show corners on original images
ax = plt.subplot(3,2,5)
plt.imshow(I1)
for i in range(C1): # draw rectangles of size w around corners
    x,y = pts1[:,i]
    ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
# plt.plot(pts1[0,:], pts1[1,:], '.b') # display subpixel corners
plt.title('found %d corners'%C1)

ax = plt.subplot(3,2,6)
plt.imshow(I2)
for i in range(C2):
    x,y = pts2[:,i]
    ax.add_patch(patches.Rectangle((x-w/2,y-w/2),w,w, fill=False))
# plt.plot(pts2[0,:], pts2[1,:], '.b')
plt.title('found %d corners'%C2)

plt.show()
```

Final values for parameters

- w =
- t =
- w_nms =
- C1 =
- C2 =

1.3 Problem 2 (Programming): Feature matching (15 points)

Determine the set of one-to-one putative feature correspondences by performing a brute-force search for the greatest correlation coefficient value (in the range [-1, 1]) between the detected features in image 1 and the detected features in image 2. Only allow matches that are above a specified correlation coefficient threshold value (note that calculating the correlation coefficient allows for adjusting the size of the matching window without changing the threshold value). Further, only allow matches that are above a specified distance ratio threshold value, where distance is measured to the next best match for a given feature. Vary these parameters such that 160-240 putative feature correspondences are established. Optional: constrain the search to coordinates in image 2 that are within a proximity of the detected feature coordinates in image 1.

Use the following formula to calculate the correlation coefficient (normalized cross correlation) between two image windows I_1 and I_2:

$$\frac{\sum_{x,y}\left[I_1(x,y)-\overline{I_1}\right]\left[I_2(x,y)-\overline{I_2}\right]}{\sqrt{\sum_{x,y}\left[I_1(x,y)-\overline{I_1}\right]^2\cdot\sum_{x,y}\left[I_2(x,y)-\overline{I_2}\right]^2}}$$

where $I(x,y)$ is the pixel value of I at (x,y) and $\overline{I}$ is the mean value of I.

Note: You must center each window at the sub-pixel corner coordinates while computing normalized cross correlation, i.e., you must use bilinear interpolation to compute the pixel values at non-integer coordinates; otherwise, you will lose points.

You may use scipy.interpolate.RegularGridInterpolator to perform bilinear interpolation.

Report your final values for:

- the size of the matching window
- the correlation coefficient threshold
- the distance ratio threshold
- the size of the proximity window (if used)
- the resulting number of putative feature correspondences (i.e., matched features)

Display figures for:

- pair of images, where the matched features in each of the images are indicated by a square window about the feature.

(You must use original (color) images to the draw boxes and correspondence lines)

A typical implementation takes around 10 seconds to run. If yours takes more than 120 seconds, you may lose points.

[ ]:
```
def compute_ncc(I1, I2, pts1, pts2, w, p):
    # compute the normalized cross correlation between image patches I1, I2
    # result should be in the range [-1,1]
    #
    # Do ensure that windows are centered at the sub-pixel co-ordinates
    #      while computing normalized cross correlation.
    #
    # inputs:
    # I1, I2 are the input images
    # pts1, pts2 are the point to be matched
    # w is the size of the matching window to compute correlation coefficients
    # p is the size of the proximity window
    #
    # output:
    # normalized cross correlation matrix of scores between all windows in
    #    image 1 and all windows in image 2
    #

    """your code here"""

    return scores


def perform_match(scores, t, d):
    # perform the one-to-one correspondence matching on the correlation␣
 ↪coefficient matrix
    #
    # inputs:
    # scores is the NCC matrix
    # t is the correlation coefficient threshold
    # d distance ration threshold
    #
    # output:
    # 2xM array of the feature coordinates in image 1 and image 2,
    # where M is the number of matches.

    """your code here"""
    inds = []

    return inds



def run_feature_matching(I1, I2, pts1, pts2, w, t, d, p):
    # inputs:
    # I1, I2 are the input images
    # pts1, pts2 are the point to be matched
    # w is the size of the matching window to compute correlation coefficients
    # t is the correlation coefficient threshold
    # d distance ration threshold
    # p is the size of the proximity window
    #
    # outputs:
    # inds is a 2xk matrix of matches where inds[0,i] indexs a point pts1
    #     and inds[1,i] indexs a point in pts2, where k is the number of matches

    scores = compute_ncc(I1, I2, pts1, pts2, w, p)
    inds = perform_match(scores, t, d)
    return inds
```

[ ]:
```
# parameters to tune
w = 1
t = 1
d = 1
p = np.inf

tic = time.time()
# run the feature matching algorithm on the input images and detected features
inds = run_feature_matching(I1, I2, pts1, pts2, w, t, d, p)
toc = time.time() - tic

print('took %f secs'%toc)

# create new matrices of points which contain only the matched features
match1 = pts1[:,inds[0,:].astype('int')]
match2 = pts2[:,inds[1,:].astype('int')]

# display the results
```

```
plt.figure(figsize=(14,24))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)
plt.title('found %d putative matches'%match1.shape[1])
for i in range(match1.shape[1]):
    x1,y1 = match1[:,i]
    x2,y2 = match2[:,i]
    ax1.plot([x1, x2],[y1, y2],'-r')
    ax1.add_patch(patches.Rectangle((x1-w/2,y1-w/2),w,w, fill=False))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w/2,y2-w/2),w,w, fill=False))

plt.show()

# test 1-1
print('unique points in image 1: %d'%np.unique(inds[0,:]).shape[0])
print('unique points in image 2: %d'%np.unique(inds[1,:]).shape[0])
```

Final values for parameters

- w =
- t =
- d =
- p =
- num_matches =

1.4 Problem 3 (Programming): Outlier Rejection (15 points)

The resulting set of putative point correspondences should contain both inlier and outlier correspondences (i.e., false matches). Determine the set of inlier point correspondences using the M-estimator Sample Consensus (MSAC) algorithm, where the maximum number of attempts to find a consensus set is determined adaptively. For each trial, you must use the 4-point algorithm (as described in lecture) to estimate the planar projective transformation from the 2D points in image 1 to the 2D points in image 2. Calculate the (squared) Sampson error as a first order approximation to the geometric error. There must be at least **150 inlier correspondences**.

hint: this problem has codimension 2

Report your values for:

- the probability p that as least one of the random samples does not contain any outliers
- the probability α that a given point is an inlier
- the resulting number of inliers
- the number of attempts to find the consensus set
- the tolerance for inliers
- the cost threshold

Display figures for:

- pair of images, where the inlier features in each of the images are indicated by a square window about the feature and a line segment is drawn from the feature to the coordinates of the corresponding feature in the other image

```
[ ]: from scipy.stats import chi2

     def display_results(H, title):
         print(title+' =')
         print (H/np.linalg.norm(H)*np.sign(H[-1,-1]))

     def homogenize(x):
         # converts points from inhomogeneous to homogeneous coordinates
         return np.vstack((x,np.ones((1,x.shape[1]))))

     def dehomogenize(x):
         # converts points from homogeneous to inhomogeneous coordinates
         return x[:-1]/x[-1]

     def compute_MSAC_cost(H,x1,x2,tol):
         # Inputs:
         # x1 - 2D points in image 1
         # x2 - 2D points in image 2
         # H - planar projective transformation matrix
         # tol - tolerance for establishing datum/model compatibility
         #
         # Output:
         # cost - Squared sampson error
         """your code here"""

         cost = np.inf
         return cost

     def determine_inliers(pts1, pts2, thresh, tol, p):
         # Inputs:
         #    pts1 - matched feature correspondences in image 1
         #    pts2 - matched feature correspondences in image 2
         #    thresh - cost threshold
         #    tol - reprojection error tolerance
         #    p - probability that as least one of the random samples does not␣
      ↪contain any outliers
         #
         # Output:
         #    consensus_min_cost - final cost from MSAC
         #    consensus_min_cost_model - planar projective transformation matrix H
         #    inliers - list of indices of the inliers corresponding to input data
         #    trials - number of attempts taken to find consensus set
```

```
    """your code here"""



    trials = 0
    max_trials = np.inf
    consensus_min_cost = np.inf
    consensus_min_cost_model = np.zeros((3,3))
    inliers = np.random.randint(0, 200, size=100)
    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 0
tol = 0
p = 0
alpha = 0

tic=time.time()

cost_MSAC, H_MSAC, inliers, trials = determine_inliers(match1, match2, thresh,␣
 ↪tol, p)

# choose just the inliers
xin1 = match1[:,inliers]
xin2 = match2[:,inliers]

toc=time.time()
time_total=toc-tic

# display the results
print('took %f secs'%time_total)
print('%d iterations'%trials)
print('inlier count: ',len(inliers))
print('inliers: ',inliers)
print('MSAC Cost=%.9f'%cost_MSAC)
display_results(H_MSAC, 'H_MSAC')

# display the figures
plt.figure(figsize=(14,24))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)
plt.title('found %d inliers'%xin1.shape[1])
for i in range(xin1.shape[1]):
    x1,y1 = xin1[:,i]
    x2,y2 = xin2[:,i]
    ax1.plot([x1, x2],[y1, y2],'-r')
    ax1.add_patch(patches.Rectangle((x1-w/2,y1-w/2),w,w, fill=False))
    ax2.plot([x2, x1],[y2, y1],'-r')
    ax2.add_patch(patches.Rectangle((x2-w/2,y2-w/2),w,w, fill=False))

plt.show()
```

Final values for parameters

- $p =$
- $\alpha =$
- tolerance =
- threshold =
- num_inliers =
- num_attempts =

1.5 Problem 4 (Programming): Linear Estimate (15 points)

Estimate the planar projective transformation $\boldsymbol{H}_{\text{DLT}}$ from the resulting set of inlier correspondences using the direct linear transformation (DLT) algorithm (with data normalization). You must express $\boldsymbol{x}'_i = \boldsymbol{H}\boldsymbol{x}_i$ as $[\boldsymbol{x}'_i]^{\perp}\boldsymbol{H}\boldsymbol{x}_i = \boldsymbol{0}$ (not $\boldsymbol{x}'_i \times \boldsymbol{H}\boldsymbol{x}_i = \boldsymbol{0}$), where $[\boldsymbol{x}'_i]^{\perp}\boldsymbol{x}'_i = \boldsymbol{0}$, when forming the solution. Return $\boldsymbol{H}_{\text{DLT}}$, scaled such that $||\boldsymbol{H}_{\text{DLT}}||_{\text{Fro}} = 1$

[ ]:
```
def data_normalize(pts):
    # Input:
    # pts - 2D scene points
    # Outputs:
    # pts - data normalized points
    # T - corresponding transformation matrix

    """your code here"""

    T = np.eye(pts.shape[0] + 1)
    return pts, T

def compute_sampson_cost(H,x1,x2):
    # Inputs:
    # x1 - 2D points in image 1
    # x2 - 2D points in image 2
    # H - planar projective transformation matrix
    #
    # Output:
    # cost - Squared sampson error
    """your code here"""

    cost = np.inf
    return cost

def estimate_2D_projective_transformation_linear(x1, x2, normalize=True):
    # Inputs:
    #    x1 - inhomogeneous inlier correspondences in image 1
    #    x2 - inhomogeneous inlier correspondences in image 1
    #    normalize - if True, apply data normalization to x1 and x2
    #
    # Outputs:
    #    H - the DLT estimate of the planar projective transformation
    #    cost - Sampson cost for the above DLT Estimate H. Assume points in␣
 ↪image 1 as scene points.

    """your code here"""

    # data normalization
    if normalize:
        print('normalize')


    # data denormalize
    if normalize:
        print('denormalize')


    H = np.eye(3)/np.sqrt(3)
    cost = np.inf
    return H, cost


# compute the linear estimate without data normalization
print('Running DLT without data normalization')
time_start=time.time()
H_DLT,cost = estimate_2D_projective_transformation_linear(x1, x2,␣
 ↪normalize=False)
time_total=time.time()-time_start
# display the results
print('took %f secs'%time_total)
print('Cost=%.9f'%cost)


# compute the linear estimate with data normalization
print ('Running DLT with data normalization')
time_start=time.time()
H_DLT, cost = estimate_2D_projective_transformation_linear(xin1, xin2,␣
 ↪normalize=True)
time_total=time.time()-time_start
```

```
# display the results
print('took %f secs'%time_total)
print('Cost=%.9f'%cost)
```

[ ]:
```
# display your H_DLT, scaled with its frobenius norm
display_results(H_DLT, 'H_DLT')
```

1.6 Problem 5 (Programming): Nonlinear Estimate (45 points)

Use $\boldsymbol{H}_{\mathrm{DLT}}$ and the Sampson corrected points (in image 1) as an initial estimate to an iterative estimation method, specifically the sparse Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the planar projective transformation that minimizes the reprojection error. You must parameterize the planar projective transformation matrix and the homogeneous 2D scene points that are being adjusted using the parameterization of homogeneous vectors.

Report the initial cost (i.e. cost at iteration 0) and the cost at the end of each successive iteration. Show the numerical values for the final estimate of the planar projective transformation matrix $\boldsymbol{H}_{\mathrm{LM}}$, scaled such that $||\boldsymbol{H}_{\mathrm{LM}}||_{\mathrm{Fro}} = 1$.

[ ]:
```
# Note that np.sinc is different than defined in class
def sinc(x):
    """your code here"""
    y = x
    return y

def parameterize_matrix(P):
    # wrapper function to interface with LM
    # takes all optimization variables and parameterizes all of them

    return parameterize_homog(P.reshape(-1,1))

def deparameterize_matrix(m,rows,columns):
    # Deparameterize all optimization variables
    # Input:
    #   m - matrix to be deparameterized
    #   rows - number of rows of the deparameterized matrix
    #   columns - number of rows of the deparameterized matrix
    #
    # Output:
    #    deparameterized matrix

    return deparameterize_homog(m).reshape(rows,columns)

def parameterize_homog(v_bar):
    # Given a homogeneous vector v_bar return its minimal parameterization
    """your code here"""

    return v
```

```
def deparameterize_homog(v):
    # Given a parameterized homogeneous vector return its deparameterization
    """your code here"""

    return v_bar

def partial_vbar_partial_v(v):
    # Input:
    # v - homogeneous parameterization vector
    # Output:
    # d_vbar_d_v - derivative of vbar w.r.t v

    """your code here"""

    return d_vbar_d_v

def partial_xhat_partial_hbar(H,x):
    # Mapping deparameterized scene point to the second image under the␣
 ↪deparameterized second 2D projective transformation matrix
    # Input:
    # H - 2D projective transformation matrix (3x3)
    # x - deparameterized scene point (3x1)
    # Output:
    # d_xhat_d_hbar - derivative of xhat w.r.t hbar (2x9)

    """your code here"""

    return d_xhat_d_hbar

def partial_xhat_partial_xbar(H,x):
    #Map the deparameterized scene point to the first/second image under the␣
 ↪first/second 2D projective transformation matrix
    # Input:
    # H - 2D projective transformation matrix (3x3)
    # x - deparameterized scene point (3x1)
    # Output:
    # d_xhat_d_xbar - derivative of xhat w.r.t xbar (2x3)

    """your code here"""

    return d_xhat_d_xbar

def data_normalize_with_cov(pts, covarx):
    # data normalization of n dimensional pts
    #
    # Input:
```

```
    # pts - is in inhomogeneous coordinates
    # covarx - covariance matrix associated with x. Has size 2n x 2n, where n␣
↪is number of points.
    # Outputs:
    # pts - data normalized points
    # T - corresponding transformation matrix
    # covarx - normalized covariance matrix
    """your code here"""

    T = np.eye(pts.shape[0]+1)
    return pts, T, covarx

def compute_cost(x1,x2,scene_pts,H,cov1,cov2):
    # Inputs:
    # x1 - 2D points in image 1
    # x2 - 2D points in image 2
    # scene_pts - 3D scene points
    # H - 2D projective transformation matrix (3x3)
    # cov1 - covariance matrix associated with points in image 1
    # cov2 - covariance matrix associated with points in image 2
    #
    # Output:
    # cost - total error
    """your code here"""

    cost = np.inf
    return cost
```

[2]:
```
# Unit Tests (Do not change)

def check_values_jacobian():
    eps = 1e-5  # Floating point error threshold
    x_scene_hat = np.load('unit_test/scene_point_parameterized.npy')
    h_hat = np.load('unit_test/H_parameterized.npy')

    d_hbar_d_hhat_target  = np.load('unit_test/d_hbar_d_hhat.npy')
    d_xhat_d_hbar_target  = np.load('unit_test/d_xhat_d_hbar.npy')
    d_xhat1_d_xbar_target = np.load('unit_test/d_xhat1_d_xbar.npy')
    d_xhat2_d_xbar_target = np.load('unit_test/d_xhat2_d_xbar.npy')

    d_hbar_d_hhat  = partial_vbar_partial_v(h_hat)
    d_xhat_d_hbar  =␣
↪partial_xhat_partial_hbar(deparameterize_matrix(h_hat,3,3),deparameterize_homog(x_scene_hat
    d_xhat1_d_xbar = partial_xhat_partial_xbar(np.
↪eye(3),deparameterize_homog(x_scene_hat))
    d_xhat2_d_xbar =␣
↪partial_xhat_partial_xbar(deparameterize_matrix(h_hat,3,3),deparameterize_homog(x_scene_hat
    d_hbar_d_hhat_valid = np.all(np.abs(d_hbar_d_hhat - d_hbar_d_hhat_target) <␣
↪eps)
    d_xhat_d_hbar_valid = np.all(np.abs(d_xhat_d_hbar - d_xhat_d_hbar_target) <␣
↪eps)
    d_xhat1_d_xbar_valid = np.all(np.abs(d_xhat1_d_xbar -␣
↪d_xhat1_d_xbar_target) < eps)
    d_xhat2_d_xbar_valid = np.all(np.abs(d_xhat2_d_xbar -␣
↪d_xhat2_d_xbar_target) < eps)

    print(f'Computed partial_hbar_partial_h is equal to the given value +/-␣
↪{eps}: {d_hbar_d_hhat_valid}')
    print(f'Computed partial_xhat_partial_hbar is equal to the given value +/-␣
↪{eps}: {d_xhat_d_hbar_valid}')
    print(f'Computed partial_xhat_partial_xbar is equal to the given value +/-␣
↪{eps}: {d_xhat1_d_xbar_valid}')
    print(f'Computed partial_xhat\'_partial_xbar is equal to the given value +/
↪- {eps}: {d_xhat2_d_xbar_valid}')

check_values_jacobian()
```

[ ]:
```
def estimate_2D_projective_transformation_non_linear(H, x1, x2, max_iters, lam):
    # Input:
    #    H - Initial estimate of planar projective transformation matrix
    #    x1 - inhomogeneous inlier points in image 1
    #    x2 - inhomogeneous inlier points in image 2
    #    max_iters - maximum number of iterations
    #    lam - lambda parameter
    # Output:
    #    H - Final H (3x3) obtained after convergence

    # data normalization

    n_points = x1.shape[1]
    """your code here"""


    for i in range(max_iters):
        cost = np.inf
        print('iter %03d Cost %.9f Avg cost per point: %.9f'%(it+1, cost, cost/
↪n_points))

    # data denormalization

    return H
```

```
# LM hyperparameters
lam = .001
max_iters = 100

# Run LM initialized by DLT estimate with data normalization
print ('Running sparse LM with data normalization')
print('iter %03d Cost %.9f Avg cost per point: %.9f'%(0, cost, cost/xin1.
 ↪shape[1]))
time_start=time.time()
H_LM = estimate_2D_projective_transformation_non_linear(H_DLT, xin1, xin2,␣
 ↪max_iters, lam)
time_total=time.time()-time_start
print('took %f secs'%time_total)
```

```
[ ]: # display your converged H_LM, scaled with its frobenius norm
     display_results(H_LM, 'H_LM')
```