CSE252B_WI26_assignment_5

March 11, 2026

1 CSE 252B: Computer Vision II, Winter 2026 – Assignment 5

Instructor: Ben Ochoa

Assignment due: Wed, Mar 18, 11:59 PM

Name:

PID:

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 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.
- Your code should be well written with sufficient comments to understand, but there is no need to write extra markdown to describe your solution if it is not explictly asked for.
- This notebook contains skeleton code, which should not be modified (this is important for standardization to facilate efficient grading).
- You may use python packages for basic linear algebra, but you may not use functions that directly solve the problem. If you are uncertain about using a specific package, function, or method, then please ask the instructional staff whether it is allowable.
- **You must submit this notebook as an .ipynb file, a .py file, and a .pdf file on Gradescope.**
  - You may directly export the notebook as a .py file. You may use nbconvert to convert the .ipynb file to a .py file using the following command `jupyter nbconvert --to script filename.ipynb`
  - There are two methods to convert the notebook to a .pdf file.
    - You may first export the notebook as a .html file, then print the web page as a .pdf file.
    - If you have XeTeX installed, then you may directly export the notebook as a .pdf file. You may use nbconvert to convert a .ipynb file to a .pdf file using the following command `jupyter nbconvert --allow-chromium-download --to webpdf filename.ipynb`
  - **You must ensure the contents in each cell (e.g., code, output images, printed results, etc.) are clearly visible, and are not cut off or partially cropped in the .pdf file.**

  - Your code and results must remain inline in the .pdf file (do not move your code to an appendix).
  - **While submitting on gradescope, you must assign the relevant pages in the .pdf file submission for each problem.**
- It is highly recommended that you begin working on this assignment early.

1.2 Problem 1 (Math): Point on Line Closest to the Origin (5 points)

Given a line $= (a, b, c)^\top$, show that the point on that is closest to the origin is the point $\mathbf{x} = (-ac, -bc, a^2 + b^2)^\top$ (Hint: this calculation is needed in the two-view optimal triangulation method used below).

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

Download input data from the course website. The file Sport0_OG0.bmp contains image 1 and the file Sport1_OG0.bmp contains image 2.

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

$$\mathbf{M} = \begin{bmatrix} \sum_x \sum_y \left(R_x^2 + G_x^2 + B_x^2\right) & \sum_x \sum_y \left(R_x R_y + G_x G_y + B_x B_y\right) \\ \sum_x \sum_y \left(R_x R_y + G_x G_y + B_x B_y\right) & \sum_x \sum_y \left(R_y^2 + G_y^2 + B_y^2\right) \end{bmatrix}$$

$$\text{where } R_x = \frac{\partial R}{\partial x}, \quad R_y = \frac{\partial R}{\partial y}, \quad G_x = \frac{\partial G}{\partial x}, \quad G_y = \frac{\partial G}{\partial y}, \quad B_x = \frac{\partial B}{\partial x}, \quad B_y = \frac{\partial B}{\partial y}.$$

where R, G and B refer to the color channels of the input image, and x and y represent the horizontal and vertical gradient directions respectively. The gradient matrix M is computed over a local window of size w. Calculate the gradient images using the five-point central difference operator. Set resulting values that are below a specified threshold value to zero (hint: calculating the mean instead of the sum in M 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 450–550 features are detected in each image. For resulting nonzero pixel values, determine the subpixel feature coordinate using the Förstner corner point operator.

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

Note: You must use the color images (not grayscale) for feature detection; otherwise you will lose points.

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 M)
- 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:

- minor eigenvalue images before thresholding
- minor eigenvalue images after thresholding
- original images with detected features

A typical implementation takes around 30 seconds to run. If yours takes more than 120 seconds, you may lose points.

[ ]:
```
%matplotlib inline
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy.signal import convolve2d as conv2d
import scipy.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')

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)
```

[ ]:
```
import time

# input images
I1 = np.array(Image.open('Sport0_OG0.bmp'), dtype='float')/255.
I2 = np.array(Image.open('Sport1_OG0.bmp'), 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 corners on original images
ax = plt.subplot(1,2,1)
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,␣
 ↪color='yellow'))
# plt.plot(pts1[0,:], pts1[1,:], '.b') # display subpixel corners
plt.title('Found %d Corners'%C1)

ax = plt.subplot(1,2,2)
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,␣
 ↪color='yellow'))
# plt.plot(pts2[0,:], pts2[1,:], '.b')
plt.title('Found %d Corners'%C2)
```

```
plt.show()

# final values for parameters
print(f'w = {w}')
print(f't = {t}')
print(f'w_nms = {w_nms}')
print(f'C1 = {C1}')
print(f'C2 = {C2}')
```

2.1 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. The proximity is calculated using the subpixel coordinates of the detected feature coordinates in image 1 and image 2. Given (x_1, y_1) in image 1 and (x_2, y_2) in image 2, you can think of a square with side length p, centered at (x_2, y_2). Then, (x_1, y_1) is within the proximity window if it lies inside that square.

Use the following formula to calculate the correlation coefficient (normalized cross correlation) between two image windows w in image 1 and w' in image 2:

$$\gamma = \frac{\sum_x \sum_y \left(w'(x,y) - \bar{w}'\right)\left(w(x,y) - \bar{w}\right)}{\sqrt{\left(\sum_x \sum_y \left(w'(x,y) - \bar{w}'\right)^2\right)\left(\sum_x \sum_y \left(w(x,y) - \bar{w}\right)^2\right)}}, \quad \gamma \in [-1, 1]$$

For color images:

$$\gamma = \min(\gamma_R, \gamma_G, \gamma_B)$$

where $w(x, y)$ is the pixel value of image 1 at (x, y) and $\bar{w}$ is the mean value of the window w. Similarly, $w'(x, y)$ and $\bar{w}'$ correspond to pixel values and mean of the window in image 2.

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; Also, perform normalized cross correlation using the color image channels (not grayscale); otherwise, you will lose points.

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.

[ ]:
```
from scipy.interpolate import RegularGridInterpolator

def bilinear_interpolation(pts,I_single_channel,w):
    # inputs:
    # pts: center points
    # I_single_channel: single channel input image (mxn)
    # w: window size
    #
    # output:
    # Interpolated pixel values for the corner windows

    half_win = w//2
    I_single_channel = np.pad(I_single_channel,pad_width=half_win)
    x = np.linspace(0,I_single_channel.shape[1]-1,I_single_channel.shape[1])
    y = np.linspace(0,I_single_channel.shape[0]-1,I_single_channel.shape[0])
    interp = RegularGridInterpolator((y, x),I_single_channel,␣
 ↪bounds_error=False, fill_value=None)

    windows = []

    for c in range(pts.shape[1]):
      xx = np.linspace(pts[0][c]-half_win,pts[0][c]+half_win+1,2*half_win+1)
      yy = np.linspace(pts[1][c]-half_win,pts[1][c]+half_win+1,2*half_win+1)
      X, Y = np.meshgrid(xx, yy, indexing='ij')
      w1 = interp((Y,X))
      windows.append(w1)

    return windows

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()

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
print(f'w = {w}')
print(f't = {t}')
print(f'd = {d}')
print(f'p = {p}')
```

2.2 Problem 4 (Programming): Outlier Rejection (20 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 7-point algorithm (as described in lecture) to estimate the essential matrix, resulting in 1 or 3 solutions. Note that the 7-point algorithm requires the 2D points in normalized coordinates, not in pixel coordinates. Calculate the (squared) Sampson error as a first order approximation to the geometric error (hint: the error tolerance is simpler to calculate in pixel coordinates using the fundamental matrix $\mathtt{F} = \mathtt{K}'^{-\top}\mathtt{E}\mathtt{K}^{-1}$ than in normalized coordinates using $\mathtt{E}$. You can avoid doing covariance propagation).

Hint: this problem has codimension 1

Also: fix a random seed in your MSAC. If the instructional team cannot reproduce your results, then you will lose points. But do not try to find a good seed as the instructional team will run your code using different seeds. Try to get 130-200 inliers every time you run MSAC.

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
- random seed

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

[ ]:
```
## Camera Calibration Matrices for the two cameras

K1 = np.array([[933.506409, 0, 377.685547],
               [0, 907.118286, 287.696991],
               [0, 0, 1]])

K2 = np.array([[934.712585, 0, 375.183105],
               [0, 903.909607, 290.008118],
               [0, 0, 1]])
```

[ ]:
```
import sympy as sp
from scipy.stats import chi2

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(x1, x2, F, tol):
    # Input:
    # x1, x2: (3 x n) Homogeneous points in pixel coordinates
    # F: 3x3 fundamental matrix
    # tol: Reprojection error tolerance

    # Output:
    # cost, number of inliers

    num_inliers = 0
    total_cost = 0
    """your code here"""

    return total_cost, num_inliers
```

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

def determine_inliers(pts1, pts2, thresh, tol, p, K1, K2):
    # 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
    #    K1 - the camera calibration matrix associated with image 1
    #    K2 - the camera calibration matrix associated with image 2
    #
    # Output:
    #    consensus_min_cost - final cost from MSAC
    #    consensus_min_cost_model - essential matrix E
    #    inliers - list of indices of the inliers corresponding to input data
    #    trials - number of attempts taken to find consensus set

    np.random.seed(seed)
    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)
    """your code here"""

    return consensus_min_cost, consensus_min_cost_model, inliers, trials


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

tic=time.time()

cost_MSAC, E_MSAC, inliers, trials = determine_inliers(match1, match2, thresh,␣
 ↪tol, p, K1, K2)

# 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(E_MSAC, 'E_MSAC')

# display the figures
plt.figure(figsize=(14,8))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)

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
print(f'random seed = {seed}')
print(f'p = {p}')
print(f'alpha = {alpha}')
print(f'tolerance = {tol}')
print(f'threshold = {thresh}')
print(f'num_inliers = {len(inliers)}')
print(f'num_attempts = {trials}')
print(f'consensus min cost = {cost_MSAC}')
```

2.3 Problem 5 (Programming): Linear Estimation of the Essential Matrix (15 points)

Estimate the essential matrix $\mathtt{E}_{\mathrm{DLT}}$ from the resulting set of inlier correspondences using the direct linear transformation (DLT) algorithm. Include the numerical values of the resulting $\mathtt{E}_{\mathrm{DLT}}$, scaled such that $\|\mathtt{E}_{\mathrm{DLT}}\|_{\mathrm{Fro}} = 1$

[ ]:
```
def estimate_essential_matrix_linear(x1, x2, K1, K2):
    # Inputs:
    #    x1 - inhomogeneous inlier correspondences in image 1
    #    x2 - inhomogeneous inlier correspondences in image 2
    #
    # Outputs:
    #    E - the DLT estimate of the essential matrix

    E = np.zeros((3,3))
    """your code here"""

    return E


# compute the linear estimate
time_start=time.time()
E_DLT = estimate_essential_matrix_linear(xin1, xin2, K1, K2)
time_total=time.time()-time_start

# display the resulting E_DLT, scaled with its frobenius norm
display_results(E_DLT, 'E_DLT')
```

2.4 Problem 6 (Programming): Nonlinear Estimation of the Essential Matrix (70 points)

Retrieve the normalized camera projection matrices $\hat{\mathtt{P}} = [\mathtt{I} \,|\, \mathbf{0}]$ and $\hat{\mathtt{P}}' = [\mathtt{R} \,|\, \mathbf{t}]$ from $\mathtt{E}_{\mathrm{DLT}}$. Use the resulting camera rotation and translation associated with the second camera and the triangulated 3D points as an initial estimate to an iterative estimation method, specifically the sparse Levenberg-

Marquardt algorithm, to determine the Maximum Likelihood estimate of the essential matrix that minimizes the reprojection error. The initial estimate of the 3D points must be determined using the two-view optimal triangulation method described in lecture (algorithm 12.1 in the Hartley & Zisserman book, but use the ray-plane intersection method for the final step instead of the homogeneous method). Additionally, you must parameterize the second camera rotation using the angle-axis representation ω (where $[\omega]_\times = \ln \mathtt{R}$) of a 3D rotation, the second camera unit translation vector $\mathbf{t}$ using the parameterization of the n-sphere, and the homogeneous 3D 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 second camera rotation ω_{LM} and $\mathtt{R}_{\text{LM}}$, second camera unit translation vector $\mathbf{t}_{\text{LM}}$, and the essential matrix $\mathtt{E}_{\text{LM}} = [\mathbf{t}_{\text{LM}}]_\times \mathtt{R}_{\text{LM}}$.

[ ]:
```
def apply_optimal_correction(x1, x2, E):
    # Input:
    #    x1 - Homogeneous point in image 1 in normalized coordinates (3 x 1)
    #    x2 - Homogeneous point in image 2 in normalized coordinates (3 x 1)
    #    E - DLT estimate of the essential matrix (3 x 3)
    # Output:
    #    x1_corrected - Homogeneous corrected point in image 1 in normalized␣
 ↪coordinates (3 x 1)
    #    x2_corrected - Homogeneous corrected point in image 2 in normalized␣
 ↪coordinates (3 x 1)

    x1_corrected = x1.copy()
    x2_corrected = x2.copy()

    """your code here"""

    return x1_corrected, x2_corrected


def triangulate_scene_point(x1, x2, E, Pp):
    # Input:
    #    x1 - Homogeneous point in image 1 in normalized coordinates (3 x 1)
    #    x2 - Homogeneous point in image 2 in normalized coordinates (3 x 1)
    #    E - DLT estimate of the essential matrix (3 x 3)
    #    Pp - the second normalized camera projection matrix
    # Output:
    #    X_scene = Triangulated homogeneous 3D scene point (4 x 1)

    X_scene = np.zeros((4, 1))

    """your code here"""

    return X_scene
```

```
def decompose_essential_matrix(E, pt1, pt2):
    #
    # This function decomposes the essential matrix into a rotation
    # matrix and translation vector such that we have normalized cameras P1 =␣
 ↪[I | 0]
    # and P2 = [R | t].  t is a unit vector indicating the direction of␣
 ↪translation.
    #
    # Input:
    #    E - DLT estimate of the essential matrix (3 x 3)
    #    pt1, pt2 - One point correspondence in normalized coordinates (3x1)
    # Output:
    #    R - The rotation matrix
    #    t - The translation vector

    R = np.eye(3)
    t = np.zeros((3, 1))

    """your code here"""

    return R,t
```

[ ]:
```
# Unit tests for optimal triangulation (Do not change)

def check_values_triangulation():
    eps = 1e-5  # Floating point error threshold

    # Givens
    # The fundamental matrix (3 x 3)
    E = np.load('unit_test/E.npy')

    # Homogeneous image points in normalized coordinates (3 x 1) each
    x1 = np.load('unit_test/x1.npy')
    x2 = np.load('unit_test/x2.npy')

    # Second normalized projection matrix P' (3 x 4)
    P_target = np.load('unit_test/P_target.npy')

    # Targets
    x1_corr_target = np.load('unit_test/x1_corr.npy')
    x2_corr_target = np.load('unit_test/x2_corr.npy')

    # Compute
    R,t = decompose_essential_matrix(E, x1[:,0].reshape(3,1), x2[:,0].
 ↪reshape(3,1))
    P = np.hstack((R,t.reshape(3,1)))
    x1_corr, x2_corr = np.empty((3,0)), np.empty((3,0))
```

```
    X_scene = np.empty((4,0))
    for i in range(x1.shape[1]):
      x1c, x2c = apply_optimal_correction(x1[:,i], x2[:,i], E)
      x1_corr = np.concatenate((x1_corr,x1c),axis=1)
      x2_corr = np.concatenate((x2_corr,x2c),axis=1)
      X_scene = np.concatenate((X_scene,triangulate_scene_point(x1[:,i], x2[:
 ↪,i], E, P)), axis=1)

    # Verify optimal corrected points
    P_valid = np.all(np.abs(P - P_target) < eps)
    x1_corr_valid = np.all(np.abs(dehomogenize(x1_corr_target) -␣
 ↪dehomogenize(x1_corr)) < eps)
    x2_corr_valid = np.all(np.abs(dehomogenize(x2_corr_target) -␣
 ↪dehomogenize(x2_corr)) < eps)

    # Verify triangulated 3D scene point
    x1_proj = dehomogenize(np.hstack((np.eye(3),np.zeros((3,1)))) @ X_scene)
    x2_proj = dehomogenize(P @ X_scene)

    x1_proj_valid = np.all(np.abs(x1_proj - dehomogenize(x1_corr)) < eps)
    x2_proj_valid = np.all(np.abs(x2_proj - dehomogenize(x2_corr)) < eps)
    X_scene_valid = x1_proj_valid and x2_proj_valid

    print(f'Computed second normalized camera projection matrix is equal to the␣
 ↪given value +/- {eps}: {P_valid}')
    print(f'Computed optimal corrected point in image 1 is equal to the given␣
 ↪value +/- {eps}: {x1_corr_valid}')
    print(f'Computed optimal corrected point in image 2 is equal to the given␣
 ↪value +/- {eps}: {x2_corr_valid}')
    print(f'Computed triangulated 3D scene point is equal to the given value +/
 ↪- {eps}: {X_scene_valid}')

check_values_triangulation()
```

[ ]:
```
# Note that np.sinc is different than defined in class
def sinc(x):

    # Returns a scalar valued sinc value
    """your code here"""

    return y

def d_sinc(x):

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

```
    return y

def skew(w):
    # Returns the skew-symmetrix represenation of a vector
    """your code here"""

    w_skew = np.zeros((3, 3))
    return w_skew


def parameterize_rotation_matrix(R):
    # Parameterizes rotation matrix into its axis-angle representation
    """your code here"""

    w = np.array([[1, 0, 0]]).T
    theta = 0
    return w, theta


def deparameterize_rotation_matrix(w):
    # Deparameterizes to get rotation matrix
    """your code here"""

    R = np.zeros((3, 3))
    return R


def unitize(x):
    return x / np.linalg.norm(x)


def parameterize_homog(v_bar):
    # Given a homogeneous vector v_bar (n x 1) return its minimal␣
 ↪parameterization ((n - 1) x 1)
    """your code here"""

    return v


def deparameterize_homog(v):
    # Given a parameterized homogeneous vector (n x 1) return its␣
 ↪deparameterization ((n + 1) x 1)
    """your code here"""

    return v_bar
```

```python
def partial_vbar_partial_v(v):
    # Input:
    # v - homogeneous parameterized vector (n x 1)
    # Output:
    # d_vbar_d_v - derivative of vbar w.r.t v ((n + 1) x n)

    """your code here"""

    return d_vbar_d_v


def partial_x_hat_partial_w(R, w, t, X):
    # Compute the (partial x_hat) / (partial omega) component of the jacobian
    # Inputs:
    #    R - 3x3 rotation matrix
    #    w - 3x1 axis-angle parameterization of R
    #    t - 3x1 translation vector
    #    X - 4x1 3D inlier point
    #
    # Output:
    #    dx_hat_dw -  matrix of size 2x3

    dx_hat_dw = np.zeros((2, 3))

    """your code here"""

    return dx_hat_dw

def householder_n_sphere(x):
    # Compute the householder matrix for vector x
    # Inputs:
    #    x - n x 1 vector
    # Output:
    #    H - n x n Householder matrix

    H = np.eye(x.shape[0])

    """your code here"""

    return H

def deparameterize_n_sphere(y):
    # Compute f(y) for the n-sphere parameterization
    # Inputs:
    #    y - n x 1 vector
    # Output:
    #    fy - (n+1) x 1 vector
```

```
    fy = np.zeros((y.shape[0]+1,1))

    """your code here"""

    return fy


def partial_x_hat_partial_t(R, t, x_norm, X):
    # Compute the (partial x_hat) / (partial t) component of the jacobian
    # Inputs:
    #    R - 3x3 rotation matrix
    #    t - 3x1 translation vector
    #    x_norm - 3x1 2D projected point in normalized coordinates
    #    X - 4x1 3D inlier point
    #
    # Output:
    #    dx_hat_dt -  matrix of size 2x3

    dx_hat_dt = np.zeros((2, 3))

    """your code here"""

    return dx_hat_dt


def partial_x_hat_partial_X_hat(P, X):
    # compute the dx_hat_dX_hat component for the Jacobian
    #
    # Input:
    #    P - 3x4 normalized projection matrix
    #    X - Homogenous 3D scene point (4x1)
    # Output:
    #    dx_hat_dX_hat - 2x3 matrix

    dx_hat_dX_hat = np.zeros((2, 3))

    """your code here"""

    return dx_hat_dX_hat
```

[ ]:
```
# Unit tests for Jacobian (Do not change)

def check_values_jacobian():
    eps = 1e-5  # Floating point error threshold

    # Givens
```

```
    # The parameterized second camera projection matrix (11 x 1)
    p_hat_prime = np.load('unit_test/p_hat_prime.npy')

    # The parameterized scene point (3 x 1)
    X_hat = np.load('unit_test/X_hat.npy')

    # The first camera projection matrix (3 x 4)
    P = np.load('unit_test/P.npy')

    # Targets
    dp_bar_prime_dp_hat_prime_target = np.load('unit_test/
↪dp_bar_prime_dp_hat_prime.npy')
    dX_bar_dX_hat_target = np.load('unit_test/dX_bar_dX_hat.npy')
    dx_hat_prime_dX_hat_target = np.load('unit_test/dx_hat_prime_dX_hat.npy')
    dx_hat_dX_hat_target = np.load('unit_test/dx_hat_dX_hat.npy')

    # Compute
    dp_bar_prime_dp_hat_prime = partial_vbar_partial_v(p_hat_prime)
    dX_bar_dX_hat = partial_vbar_partial_v(X_hat)

    P_prime = deparameterize_homog(p_hat_prime).reshape((3, 4))
    X = deparameterize_homog(X_hat)  # (4 x 1) vector

    dx_hat_prime_dX_hat = partial_x_hat_partial_X_hat(P_prime, X)
    dx_hat_dX_hat = partial_x_hat_partial_X_hat(P, X)

    # Check
    dp_bar_prime_dp_hat_prime_valid = np.all(np.
↪abs(dp_bar_prime_dp_hat_prime_target - dp_bar_prime_dp_hat_prime) < eps)
    dX_bar_dX_hat_valid = np.all(np.abs(dX_bar_dX_hat_target - dX_bar_dX_hat) <␣
↪eps)
    dx_hat_prime_dX_hat_valid = np.all(np.abs(dx_hat_prime_dX_hat_target -␣
↪dx_hat_prime_dX_hat) < eps)
    dx_hat_dX_hat_valid = np.all(np.abs(dx_hat_dX_hat_target - dx_hat_dX_hat) <␣
↪eps)

    print(f'Computed partial_pbar\'_partial_phat\' is equal to the given value␣
↪+/- {eps}: {dp_bar_prime_dp_hat_prime_valid}')
    print(f'Computed partial_Xbar_partial_Xhat is equal to the given value +/-␣
↪{eps}: {dX_bar_dX_hat_valid}')
    # print(f'Computed partial_xhat\'_partial_phat\' is equal to the given␣
↪value +/- {eps}: {dx_hat_prime_dp_hat_prime_valid}')
    print(f'Computed partial_xhat\'_partial_Xhat is equal to the given value +/
↪- {eps}: {dx_hat_prime_dX_hat_valid}')
    print(f'Computed partial_xhat_partial_Xhat is equal to the given value +/-␣
↪{eps}: {dx_hat_dX_hat_valid}')
```

```
check_values_jacobian()
```

[ ]:
```
def estimate_essential_matrix_nonlinear(E, x1, x2, max_iters, lam, K1, K2):
    # Input:
    #    E - DLT estimate of the essential 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:
    #    w - Final angle-axis representation of second camera rotation after␣
 ↪convergence
    #    t - Final second camera unit translation vector after convergence


    """your code here"""

    cost = np.inf
    print ('iter %03d Cost %.9f'%(0, cost))

    for i in range(max_iters):
        print ('iter %03d Cost %.9f'%(i+1, cost))
    return w, t

# LM hyperparameters
lam = .001
max_iters = 10

# Run LM initialized by DLT estimate
print ('Sparse LM')
time_start=time.time()
w_LM, t_LM = estimate_essential_matrix_nonlinear(E_DLT, xin1, xin2, max_iters,␣
 ↪lam, K1, K2)
time_total=time.time()-time_start
print('took %f secs'%time_total)
R_LM = deparameterize_rotation_matrix(w_LM)
E_LM = skew(t_LM) @ R_LM

# display the results
print('w_LM = ')
print(w_LM)
print('R_LM = ')
print(R_LM)
print('t_LM = ')
print(t_LM)
display_results(E_LM, 'E_LM')
```

2.5 Problem 7 (Programming): Point to Line Mapping (10 points)

Qualitatively determine the accuracy of $\mathtt{E}_{\mathrm{LM}}$ by mapping points in image 1 to epipolar lines in image 2. Identify three distinct corners distributed in image 1 that are not in the set of inlier correspondences, selected manually by visual inspection as clear corner features and well distributed across the image, visually approximate their pixel coordinates $\mathbf{x}_{\{1,2,3\}}$, and map them to epipolar lines $\ell'_{\{1,2,3\}} = \mathtt{K}'^{-\top}\mathtt{E}_{\mathrm{LM}}\mathtt{K}^{-1}\mathbf{x}_{\{1,2,3\}}$ in the second image under the essential matrix $\mathtt{E}_{\mathrm{LM}}$.

Include a figure containing the pair of images, where the three points in image 1 are indicated by a square (or circle) about the feature and the corresponding epipolar lines are drawn in image 2. Comment on the qualitative accuracy of the mapping. (Hint: each line ℓ'_i should pass through the point $\mathbf{x}'_i$ in image 2 that corresponds to the point $\mathbf{x}_i$ in image 1).

[ ]:
```
# Store your three points in image 1 in variable xchosen1
# Store the corresponding epipolar lines in variable epi_lines

# You can modify the code to display the figures, to highlight the␣
 ↪corresponding point in image 2.
#    You will have to find the pixel co-ordinates of the
#    corresponding point in image 2 manually, as we are explicitly not choosing␣
 ↪inliers (find the real matching point
#    and not the one your code outputs). The epipolar lines should
#    pass close by or through these points.
#

"""your code here"""


# display the figures
plt.figure(figsize=(28,16))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)
ax1.imshow(I1)
ax2.imshow(I2)
im_height, im_width = I1.shape[:2]
x_ax = np.linspace(0, im_width, im_width*10)
colors = ['red', 'blue', 'yellow']
for i in range(xchosen1.shape[1]):
    a, b, c = epi_lines[:, i]
    xx, yy = [], []
    for xval in x_ax:
        yval = -(a/b)*xval - c/b
        if yval > 0 and yval < im_width:
            xx.append(xval)
            yy.append(yval)
    x1,y1 = xchosen1[:,i]
    ax1.add_patch(patches.Rectangle((x1-w/2,y1-w/2),w,w, fill=True,␣
 ↪color=colors[i]))
```

```
    ax2.plot(xx,yy,'-r', color=colors[i])
plt.show()
```

“ “ “Comment on your results here.” “ ”

3D Point Cloud Visualization from Nonlinear Essential Matrix Refinement

This section visualizes the 3D scene structure recovered from the nonlinear estimation of the essential matrix using Levenberg Marquardt optimization.

[ ]:
```
import numpy as np
import open3d as o3d

R_LM = deparameterize_rotation_matrix(w_LM)
t_LM = np.asarray(t_LM).reshape(3,)

P0 = np.hstack((np.eye(3), np.zeros((3,1))))
P1 = np.hstack((R_LM, t_LM.reshape(3,1)))

x1 = homogenize(xin1)
x2 = homogenize(xin2)

K1inv = np.linalg.inv(K1)
K2inv = np.linalg.inv(K2)
x1n = K1inv @ x1
x2n = K2inv @ x2

n = x1n.shape[1]
X_list = []

for i in range(n):
    X = triangulate_scene_point(x1n[:, i], x2n[:, i], E_LM, P1)
    X_list.append(X)

X_h = np.hstack(X_list)
X = X_h[:3, :] / X_h[3:4, :]
pts = X.T.astype(np.float64)

C0 = np.zeros(3)
C1 = -R_LM.T @ t_LM

pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(pts)
pcd.paint_uniform_color([0.7, 0.7, 0.7])

baseline = np.linalg.norm(C1 - C0)
axis_size = 0.2 * baseline
sphere_radius = 0.02 * baseline
```

```
cam0_frame = o3d.geometry.TriangleMesh.create_coordinate_frame(size=axis_size)

cam0_center = o3d.geometry.TriangleMesh.create_sphere(radius=sphere_radius)
cam0_center.translate(C0)
cam0_center.paint_uniform_color([1, 1, 1])
cam0_center.compute_vertex_normals()

Rcw1 = R_LM.T

T1 = np.eye(4)
T1[:3, :3] = Rcw1
T1[:3,  3] = C1

cam1_frame = o3d.geometry.TriangleMesh.create_coordinate_frame(size=axis_size)
cam1_frame.transform(T1)

cam1_center = o3d.geometry.TriangleMesh.create_sphere(radius=sphere_radius)
cam1_center.translate(C1)
cam1_center.paint_uniform_color([1, 0.5, 0])
cam1_center.compute_vertex_normals()

o3d.visualization.draw_geometries(
    [pcd, cam0_frame, cam1_frame, cam0_center, cam1_center],
    window_name="Triangulated 3D Point Cloud + Camera Axes (No Scaling)",
    width=1400,
    height=900
)
```