HW3

February 6, 2023

1 CSE 252B: Computer Vision II, Winter 2023 – Assignment 3

1.0.1 Instructor: Ben Ochoa

1.0.2 Due: Wednesday, February 22, 2022, 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.
- 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 effecient 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.
- Your code and results should remain inline in the pdf (Do not move your code to an appendix).
- **You must submit 3 files on Gradescope - .pdf , .ipynb and .py file where the .py file is the conversion of your .ipynb to .py file . You must mark each problem on Gradescope in the pdf. You can convert you .ipynb to .py file using the following command:**

jupyter nbconvert –to script filename.ipynb –output output_filename.py

- It is highly recommended that you begin working on this assignment early.

1.2 Problem 1 (Programming): Estimation of the Camera Pose - Outlier rejection (20 points)

Download input data from the course website. The file `hw3_points3D.txt` contains the coordinates of 60 scene points in 3D (each line of the file gives the $\tilde{X}_i$, $\tilde{Y}_i$, and $\tilde{Z}_i$ inhomogeneous coordinates of a point). The file `hw3_points2D.txt` contains the coordinates of the 60 corresponding image points in 2D (each line of the file gives the $\tilde{x}_i$ and $\tilde{y}_i$ inhomogeneous coordinates of a point). The corresponding 3D scene and 2D image points contain both inlier and outlier correspondences. For

the inlier correspondences, the scene points have been randomly generated and projected to image points under a camera projection matrix (i.e., $\boldsymbol{x}_i = \boldsymbol{P}\boldsymbol{X}_i$), then noise has been added to the image point coordinates.

The camera calibration matrix was calculated for a 1280×720 sensor and 45° horizontal field of view lens. The resulting camera calibration matrix is given by

$$\boldsymbol{K} = \begin{bmatrix} 1545.0966799187809 & 0 & 639.5 \\ 0 & 1545.0966799187809 & 359.5 \\ 0 & 0 & 1 \end{bmatrix}$$

For each image point $\boldsymbol{x} = (x, y, w)^\top = (\tilde{x}, \tilde{y}, 1)^\top$, calculate the point in normalized coordinates $\hat{\boldsymbol{x}} = \boldsymbol{K}^{-1}\boldsymbol{x}$.

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, use the 3-point algorithm of Finsterwalder (as described in the paper by Haralick et al.) to estimate the camera pose (i.e., the rotation $\boldsymbol{R}$ and translation $\boldsymbol{t}$ from the world coordinate frame to the camera coordinate frame), resulting in up to 4 solutions, and calculate the error and cost for each solution. Note that the 3-point algorithm requires the 2D points in normalized coordinates, not in image coordinates. Calculate the projection error, which is the (squared) distance between projected points (the points in 3D projected under the normalized camera projection matrix $\hat{\boldsymbol{P}} = [\boldsymbol{R}|\boldsymbol{t}]$) and the measured points in normalized coordinates (hint: the error tolerance is simpler to calculate in image coordinates using $\boldsymbol{P} = \boldsymbol{K}[\boldsymbol{R}|\boldsymbol{t}]$ than in normalized coordinates using $\hat{\boldsymbol{P}} = [\boldsymbol{R}|\boldsymbol{t}]$. You can avoid doing covariance propagation). There must be at least **40 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

[1]:
```
import numpy as np
import time

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 normalize(K, x):
    # map the 2D points in pixel coordinates to the 2D points in normalized␣
 ↪coordinates
```

```
    # Inputs:
    #   K - camera calibration matrix
    #   x - 2D points in pixel coordinates
    # Output:
    #   pts - 2D points in normalized coordinates

    """your code here"""

    pts = np.zeros_like(x)
    return pts


# load data
x0 = np.loadtxt('hw3_points2D.txt').T
X0 = np.loadtxt('hw3_points3D.txt').T
print('x is', x0.shape)
print('X is', X0.shape)

K = np.array([[1545.0966799187809, 0, 639.5],
              [0, 1545.0966799187809, 359.5],
              [0, 0, 1]])

print('K =')
print(K)
```

```
x is (2, 60)
X is (3, 60)
K =
[[1.54509668e+03 0.00000000e+00 6.39500000e+02]
 [0.00000000e+00 1.54509668e+03 3.59500000e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

[2]:
```
from scipy.stats import chi2

def compute_MSAC_cost(P, x, X, K, tol):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #    tol - reprojection error tolerance
    #
    # Output:
    #    cost - total projection error

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

```
    cost = np.inf
    return cost

def determine_inliers(x, X, K, thresh, tol, p):
    # Inputs:
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #    K - camera calibration matrix
    #    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 - camera projection matrix P
    #    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, 4))
    inliers = np.random.randint(0, 59, size=10)
    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 100
tol = 0
p = 0
alpha = 0

tic = time.time()

cost_MSAC, P_MSAC, inliers, trials = determine_inliers(x0, X0, K, thresh, tol,␣
 ↪p)

# choose just the inliers
x = x0[:, inliers]
X = X0[:, inliers]

toc = time.time()
```

```
time_total = toc-tic

# display the results
print(f'took {time_total} secs')
print(f'iterations: {trials}')
print(f'inlier count: {len(inliers)}')
print(f'MSAC Cost: {cost_MSAC:.9f}')
print('P = ')
print(P_MSAC)
print('inliers: ', inliers)
```

```
took 0.0 secs
iterations: 0
inlier count: 10
MSAC Cost: inf
P =
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
inliers:  [31 39  7 53 48  8 42 17 20  9]
```

Final values for parameters

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

1.3 Problem 2 (Programming): Estimation of the Camera Pose - Linear Estimate (30 points)

Estimate the normalized camera projection matrix $\hat{\boldsymbol{P}}_{\text{linear}} = [\boldsymbol{R}_{\text{linear}}|\boldsymbol{t}_{\text{linear}}]$ from the resulting set of inlier correspondences using the linear estimation method (based on the EPnP method) described in lecture. Report the resulting $\boldsymbol{R}_{\text{linear}}$ and $\boldsymbol{t}_{\text{linear}}$.

[3]:
```
def sum_of_square_projection_error(P, x, X, K):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #
    # Output:
    #    cost - Sum of squares of the reprojection error

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

```
    cost = np.inf
    return cost

def estimate_camera_pose_linear(x, X, K):
    # Inputs:
    #    x - 2D inlier points
    #    X - 3D inlier points
    # Output:
    #    P - normalized camera projection matrix

    """your code here"""


    R = np.eye(3)
    t = np.array([[1, 0, 0]]).T
    P = np.concatenate((R, t), axis=1)
    return P

tic = time.time()
P_linear = estimate_camera_pose_linear(x, X, K)
toc = time.time()
time_total = toc - tic

# display the results
print(f'took {time_total} secs')
print('R_linear = ')
print(P_linear[:, 0:3])
print('t_linear = ')
print(P_linear[:, -1])
```

```
took 0.0 secs
R_linear =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
t_linear =
[1. 0. 0.]
```

1.4 Problem 3 (Programming): Estimation of the Camera Pose - Nonlinear Estimate (30 points)

Use $\boldsymbol{R}_{\text{linear}}$ and $\boldsymbol{t}_{\text{linear}}$ as an initial estimate to an iterative estimation method, specifically the Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the camera pose that minimizes the projection error under the normalized camera projection matrix $\hat{\boldsymbol{P}} = [\boldsymbol{R}|\boldsymbol{t}]$. You must parameterize the camera rotation using the angle-axis representation $\boldsymbol{\omega}$ (where $[\boldsymbol{\omega}]_\times = \ln \boldsymbol{R}$) of a 3D rotation, which is a 3-vector.

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 camera rotation $\boldsymbol{\omega}_{\text{LM}}$ and $\boldsymbol{R}_{\text{LM}}$, and the camera translation $\boldsymbol{t}_{\text{LM}}$.

[4]:
```
from scipy.linalg import block_diag

# Note that np.sinc is different than defined in class
def sinc(x):
    """your code here"""


    y = x
    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 data_normalize(pts):
    # Input:
    #    pts - 3D 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 normalize_with_cov(K, x, covarx):
    # Inputs:
    #    K - camera calibration matrix
    #    x - 2D points in pixel coordinates
    #    covarx - covariance matrix
    #
    # Outputs:
    #    pts - 2D points in normalized coordinates
    #    covarx - normalized covariance matrix

    """your code here"""
    pts = np.zeros_like(x)
    covarx = np.zeros_like(covarx)
    return pts, covarx


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 - 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 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 - 2D projected point in normalized coordinates
    #    X - 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 compute_cost(P, x, X, covarx):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D ground truth image points in normalized coordinates
    #    X - 3D groundtruth scene points
    #    covarx - covariance matrix
    #
    # Output:
    #    cost - total projection error

    """your code here"""

    cost = np.inf
    return cost
```

[5]:
```
# Unit Tests (Do not change)

# parameterize and deparameterize unit test
def check_values_parameterize():
    eps = 1e-8  # Floating point error threshold
    w = np.load('unit_test/omega.npy')
    R = np.load('unit_test/rotation.npy')

    w_param, _ = parameterize_rotation_matrix(R)
    R_deparam = deparameterize_rotation_matrix(w)

    param_valid = np.all(np.abs(w_param - w) < eps)
    deparam_valid = np.all(np.abs(R_deparam - R) < eps)

    print(f'Parameterized rotation matrix is equal to the given value +/- {eps}:
 ↪ {param_valid}')
    print(f'Deparameterized rotation matrix is equal to the given value +/-␣
 ↪{eps}: {deparam_valid}')

# partial_x_hat_partial_w and partial_x_hat_partial_t unit test
def check_values_jacobian():
    eps = 1e-8  # Floating point error threshold
```

```
    w = np.load('unit_test/omega.npy')
    R = np.load('unit_test/rotation.npy')
    x = np.load('unit_test/point_2d.npy')
    X = np.load('unit_test/point_3d.npy')
    t = np.load('unit_test/translation.npy')
    dx_hat_dw_target = np.load('unit_test/partial_x_partial_omega.npy')
    dx_hat_dt_target = np.load('unit_test/partial_x_partial_t.npy')

    dx_hat_dw = partial_x_hat_partial_w(R, w, t, X)
    dx_hat_dt = partial_x_hat_partial_t(R, t, x, X)

    w_valid = np.all(np.abs(dx_hat_dw - dx_hat_dw_target) < eps)
    t_valid = np.all(np.abs(dx_hat_dt - dx_hat_dt_target) < eps)

    print(f'Computed partial_x_hat_partial_w is equal to the given value +/-␣
↪{eps}: {w_valid}')
    print(f'Computed partial_x_hat_partial_t is equal to the given value +/-␣
↪{eps}: {t_valid}')

check_values_parameterize()
check_values_jacobian()
```

```
Parameterized rotation matrix is equal to the given value +/- 1e-08: False
Deparameterized rotation matrix is equal to the given value +/- 1e-08: False
Computed partial_x_hat_partial_w is equal to the given value +/- 1e-08: False
Computed partial_x_hat_partial_t is equal to the given value +/- 1e-08: False
```

[6]:
```
def estimate_camera_pose_nonlinear(P, x, X, K, max_iters, lam):
    # Inputs:
    #    P - initial estimate of camera pose
    #    x - 2D inliers
    #    X - 3D inliers
    #    K - camera calibration matrix
    #    max_iters - maximum number of iterations
    #    lam - lambda parameter
    #
    # Output:
    #    P - Final camera pose obtained after convergence

    n_points = X.shape[1]
    covarx = np.eye(2 * n_points)
    """your code here"""


    for i in range(max_iters):
```

```
        cost = compute_cost(P, x, X, covarx)
        print(f'iter: {i + 1:03d}  Cost: {cost:.09f}  Avg cost per point: {cost␣
↪/ n_points}')

    return P

# LM hyperparameters
lam = .001
max_iters = 100

tic = time.time()
P_LM = estimate_camera_pose_nonlinear(P_linear, x, X, K, max_iters, lam)
w_LM, _ = parameterize_rotation_matrix(P_LM[:, 0:3])
toc = time.time()
time_total = toc-tic

# display the results
print('took %f secs'%time_total)
print('w_LM = ')
print(w_LM)
print('R_LM = ')
print(P_LM[:,0:3])
print('t_LM = ')
print(P_LM[:,-1])
```

```
iter: 001  Cost: inf  Avg cost per point: inf
iter: 002  Cost: inf  Avg cost per point: inf
iter: 003  Cost: inf  Avg cost per point: inf
iter: 004  Cost: inf  Avg cost per point: inf
iter: 005  Cost: inf  Avg cost per point: inf
iter: 006  Cost: inf  Avg cost per point: inf
iter: 007  Cost: inf  Avg cost per point: inf
iter: 008  Cost: inf  Avg cost per point: inf
iter: 009  Cost: inf  Avg cost per point: inf
iter: 010  Cost: inf  Avg cost per point: inf
iter: 011  Cost: inf  Avg cost per point: inf
iter: 012  Cost: inf  Avg cost per point: inf
iter: 013  Cost: inf  Avg cost per point: inf
iter: 014  Cost: inf  Avg cost per point: inf
iter: 015  Cost: inf  Avg cost per point: inf
iter: 016  Cost: inf  Avg cost per point: inf
iter: 017  Cost: inf  Avg cost per point: inf
iter: 018  Cost: inf  Avg cost per point: inf
iter: 019  Cost: inf  Avg cost per point: inf
iter: 020  Cost: inf  Avg cost per point: inf
iter: 021  Cost: inf  Avg cost per point: inf
iter: 022  Cost: inf  Avg cost per point: inf
iter: 023  Cost: inf  Avg cost per point: inf
```

```
iter: 024  Cost: inf  Avg cost per point: inf
iter: 025  Cost: inf  Avg cost per point: inf
iter: 026  Cost: inf  Avg cost per point: inf
iter: 027  Cost: inf  Avg cost per point: inf
iter: 028  Cost: inf  Avg cost per point: inf
iter: 029  Cost: inf  Avg cost per point: inf
iter: 030  Cost: inf  Avg cost per point: inf
iter: 031  Cost: inf  Avg cost per point: inf
iter: 032  Cost: inf  Avg cost per point: inf
iter: 033  Cost: inf  Avg cost per point: inf
iter: 034  Cost: inf  Avg cost per point: inf
iter: 035  Cost: inf  Avg cost per point: inf
iter: 036  Cost: inf  Avg cost per point: inf
iter: 037  Cost: inf  Avg cost per point: inf
iter: 038  Cost: inf  Avg cost per point: inf
iter: 039  Cost: inf  Avg cost per point: inf
iter: 040  Cost: inf  Avg cost per point: inf
iter: 041  Cost: inf  Avg cost per point: inf
iter: 042  Cost: inf  Avg cost per point: inf
iter: 043  Cost: inf  Avg cost per point: inf
iter: 044  Cost: inf  Avg cost per point: inf
iter: 045  Cost: inf  Avg cost per point: inf
iter: 046  Cost: inf  Avg cost per point: inf
iter: 047  Cost: inf  Avg cost per point: inf
iter: 048  Cost: inf  Avg cost per point: inf
iter: 049  Cost: inf  Avg cost per point: inf
iter: 050  Cost: inf  Avg cost per point: inf
iter: 051  Cost: inf  Avg cost per point: inf
iter: 052  Cost: inf  Avg cost per point: inf
iter: 053  Cost: inf  Avg cost per point: inf
iter: 054  Cost: inf  Avg cost per point: inf
iter: 055  Cost: inf  Avg cost per point: inf
iter: 056  Cost: inf  Avg cost per point: inf
iter: 057  Cost: inf  Avg cost per point: inf
iter: 058  Cost: inf  Avg cost per point: inf
iter: 059  Cost: inf  Avg cost per point: inf
iter: 060  Cost: inf  Avg cost per point: inf
iter: 061  Cost: inf  Avg cost per point: inf
iter: 062  Cost: inf  Avg cost per point: inf
iter: 063  Cost: inf  Avg cost per point: inf
iter: 064  Cost: inf  Avg cost per point: inf
iter: 065  Cost: inf  Avg cost per point: inf
iter: 066  Cost: inf  Avg cost per point: inf
iter: 067  Cost: inf  Avg cost per point: inf
iter: 068  Cost: inf  Avg cost per point: inf
iter: 069  Cost: inf  Avg cost per point: inf
iter: 070  Cost: inf  Avg cost per point: inf
iter: 071  Cost: inf  Avg cost per point: inf
```

```
iter: 072  Cost: inf  Avg cost per point: inf
iter: 073  Cost: inf  Avg cost per point: inf
iter: 074  Cost: inf  Avg cost per point: inf
iter: 075  Cost: inf  Avg cost per point: inf
iter: 076  Cost: inf  Avg cost per point: inf
iter: 077  Cost: inf  Avg cost per point: inf
iter: 078  Cost: inf  Avg cost per point: inf
iter: 079  Cost: inf  Avg cost per point: inf
iter: 080  Cost: inf  Avg cost per point: inf
iter: 081  Cost: inf  Avg cost per point: inf
iter: 082  Cost: inf  Avg cost per point: inf
iter: 083  Cost: inf  Avg cost per point: inf
iter: 084  Cost: inf  Avg cost per point: inf
iter: 085  Cost: inf  Avg cost per point: inf
iter: 086  Cost: inf  Avg cost per point: inf
iter: 087  Cost: inf  Avg cost per point: inf
iter: 088  Cost: inf  Avg cost per point: inf
iter: 089  Cost: inf  Avg cost per point: inf
iter: 090  Cost: inf  Avg cost per point: inf
iter: 091  Cost: inf  Avg cost per point: inf
iter: 092  Cost: inf  Avg cost per point: inf
iter: 093  Cost: inf  Avg cost per point: inf
iter: 094  Cost: inf  Avg cost per point: inf
iter: 095  Cost: inf  Avg cost per point: inf
iter: 096  Cost: inf  Avg cost per point: inf
iter: 097  Cost: inf  Avg cost per point: inf
iter: 098  Cost: inf  Avg cost per point: inf
iter: 099  Cost: inf  Avg cost per point: inf
iter: 100  Cost: inf  Avg cost per point: inf
took 0.000000 secs
w_LM =
[[1]
 [0]
 [0]]
R_LM =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
t_LM =
[1. 0. 0.]
```

[ ]: