CSE252B_WI23_assignment_2

January 25, 2023

1 CSE 252B: Computer Vision II, Winter 2023 – Assignment 2

1.0.1 Instructor: Ben Ochoa

1.0.2 Due: Wednesday, February 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.
- This assignment contains both math and programming problems.
- 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 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 (Math): Line-plane intersection (5 points)

The line in 3D defined by the join of the points $\boldsymbol{X}_1 = (X_1, Y_1, Z_1, T_1)^\top$ and $\boldsymbol{X}_2 = (X_2, Y_2, Z_2, T_2)^\top$ can be represented as a Plucker matrix $\boldsymbol{L} = \boldsymbol{X}_1\boldsymbol{X}_2^\top - \boldsymbol{X}_2\boldsymbol{X}_1^\top$ or pencil of points $\boldsymbol{X}(\lambda) = \lambda\boldsymbol{X}_1 + (1-\lambda)\boldsymbol{X}_2$ (i.e., $\boldsymbol{X}$ is a function of λ). The line intersects the plane $\boldsymbol{\pi} = (a, b, c, d)^\top$ at the point $\boldsymbol{X_L} = \boldsymbol{L\pi}$ or $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$, where $\lambda_{\boldsymbol{\pi}}$ is determined such that $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})^\top\boldsymbol{\pi} = 0$ (i.e., $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ is the point on $\boldsymbol{\pi}$). Show that $\boldsymbol{X_L}$ is equal to $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ up to scale.

Your solution here

1.3 Problem 2 (Math): Line-quadric intersection (5 points)

In general, a line in 3D intersects a quadric $\boldsymbol{Q}$ at zero, one (if the line is tangent to the quadric), or two points. If the pencil of points $\boldsymbol{X}(\lambda) = \lambda\boldsymbol{X}_1 + (1-\lambda)\boldsymbol{X}_2$ represents a line in 3D, the (up to two) real roots of the quadratic polynomial $c_2\lambda_{\boldsymbol{Q}}^2 + c_1\lambda_{\boldsymbol{Q}} + c_0 = 0$ are used to solve for the intersection point(s) $\boldsymbol{X}(\lambda_{\boldsymbol{Q}})$. Show that $c_2 = \boldsymbol{X}_1^\top\boldsymbol{Q}\boldsymbol{X}_1 - 2\boldsymbol{X}_1^\top\boldsymbol{Q}\boldsymbol{X}_2 + \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2$, $c_1 = 2(\boldsymbol{X}_1^\top\boldsymbol{Q}\boldsymbol{X}_2 - \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2)$, and $c_0 = \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2$.

Your solution here

1.4 Problem 3 (Programming): Linear Estimation of the Camera Projection Matrix (15 points)

Download input data from the course website. The file hw2_points3D.txt contains the coordinates of 50 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 hw2_points2D.txt contains the coordinates of the 50 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 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.

Estimate the camera projection matrix $\boldsymbol{P}_{\text{DLT}}$ using the direct linear transformation (DLT) algorithm (with data normalization). You must express $\boldsymbol{x}_i = \boldsymbol{P}\boldsymbol{X}_i$ as $[\boldsymbol{x}_i]^\perp\boldsymbol{P}\boldsymbol{X}_i = \boldsymbol{0}$ (not $\boldsymbol{x}_i \times \boldsymbol{P}\boldsymbol{X}_i = \boldsymbol{0}$), where $[\boldsymbol{x}_i]^\perp\boldsymbol{x}_i = \boldsymbol{0}$, when forming the solution. Return $\boldsymbol{P}_{\text{DLT}}$, scaled such that $||\boldsymbol{P}_{\text{DLT}}||_{\text{Fro}} = 1$

The following helper functions may be useful in your DLT function implementation. You are welcome to add any additional helper functions.

[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 data_normalize(pts):
    # data normalization of n dimensional pts
    #
    # Input:
    #    pts - is in inhomogeneous coordinates
    # Outputs:
```

```
    #    pts - data normalized points
    #    T - corresponding transformation matrix

    """your code here"""


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

def sum_of_square_projection_error(P, x, X):
    # Inputs:
    #    P - the camera projection matrix
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    # Output:
    #    cost - Sum of squares of the reprojection error

    """your code here"""


    cost = np.inf
    return cost
```

[2]:
```
def estimate_camera_projection_matrix_linear(x, X, normalize=True):
    # Inputs:
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #    normalize - if True, apply data normalization to x and X
    #
    # Output:
    #    P - the (3x4) DLT estimate of the camera projection matrix
    P = np.eye(3,4)+np.random.randn(3,4)/10

    # data normalization
    if normalize:
        x, T = data_normalize(x)
        X, U = data_normalize(X)
    else:
        x = homogenize(x)
        X = homogenize(X)

    """your code here"""


    # data denormalize
    if normalize:
        P = np.linalg.inv(T) @ P @ U
```

```
    return P

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

# load the data
x=np.loadtxt('hw2_points2D.txt').T
X=np.loadtxt('hw2_points3D.txt').T

assert x.shape[1] == X.shape[1]
n = x.shape[1]

# compute the linear estimate without data normalization
print ('Running DLT without data normalization')
time_start=time.time()
P_DLT = estimate_camera_projection_matrix_linear(x, X, normalize=False)
cost = sum_of_square_projection_error(P_DLT, x, X)
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()
P_DLT = estimate_camera_projection_matrix_linear(x, X, normalize=True)
cost = sum_of_square_projection_error(P_DLT, x, X)
time_total=time.time()-time_start
# display the results
print('took %f secs'%time_total)
print('Cost=%.9f'%cost)

print("\n==Correct outputs==")
print("Cost=%.9f without data normalization"%97.053718991)
print("Cost=%.9f with data normalization"%84.104680130)
```

```
Running DLT without data normalization
took 0.000302 secs
Cost=inf
Running DLT with data normalization
took 0.008690 secs
Cost=inf

==Correct outputs==
Cost=97.053718991 without data normalization
```

```
Cost=84.104680130 with data normalization
```

[3]:
```
# Report your P_DLT (estimated camera projection matrix linear) value here!
display_results(P_DLT, x, X, 'P_DLT')
```

```
P_DLT =
[[ 5.61439661e-01 -4.31364493e-04  5.07262374e-02  3.74830054e-02]
 [-4.10862482e-03  6.12866520e-01  1.43106214e-02 -4.33078895e-02]
 [ 2.46814609e-02  2.97301747e-02  5.48968300e-01  1.56627803e-02]]
```

1.5 Problem 4 (Programming): Nonlinear Estimation of the Camera Projection Matrix (30 points)

Use $\boldsymbol{P}_{\text{DLT}}$ as an initial estimate to an iterative estimation method, specifically the Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the camera projection matrix that minimizes the projection error. You must parameterize the camera projection matrix as a parameterization of the homogeneous vector $\boldsymbol{p} = vec(\boldsymbol{P}^\top)$. It is highly recommended to implement a parameterization of homogeneous vector method where the homogeneous vector is of arbitrary length, as this will be used in following assignments.

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 projection matrix $\boldsymbol{P}_{\text{LM}}$, scaled such that $||\boldsymbol{P}_{\text{LM}}||_{\text{Fro}} = 1$.

The following helper functions may be useful in your LM function implementation. You are welcome and encouraged to add any additional helper functions.

Hint: LM has its biggest cost reduction after the 1st iteration. You'll know if you are implementing LM correctly if you experience this.

[4]:
```
# Note that np.sinc is different than defined in class
def sinc(x):
    # Returns a scalar valued sinc value
    """your code here"""


    y = x
    return y


def partial_x_partial_p(P,X,x):
    # compute the dx_dp component for the Jacobian
    #
    # Input:
    #    P - 3x4 projection matrix
    #    X - Homogenous 3D scene point
    #    x - inhomogenous 2D point
    # Output:
    #    dx_dp - 2x12 matrix
```

```
    dx_dp = np.zeros((2,12))

    """your code here"""

    return dx_dp


def parameterize_matrix(P):
    # wrapper function to interface with LM
    # takes all optimization variables and parameterizes all of them
    # in this case it is just P, but in future assignments it will
    # be more useful
    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
    #
    # For the camera projection, deparameterize it using␣
 ↪deparameterize_matrix(p,3,4)
    # where p is the parameterized camera projection 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 deparameterize_homog_with_Jacobian(v):
    # Input:
    #    v - homogeneous parameterization vector (11x1 in case of p)
    # Output:
    #    v_bar - deparameterized homogeneous vector
    #    partial_vbar_partial_v - derivative of v_bar w.r.t v


    partial_vbar_partial_v = np.zeros((12,11))
    v_bar = np.zeros((12,1))

    """your code here"""

    return v_bar, partial_vbar_partial_v

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(P, x, X, covarx):
    # Inputs:
    #    P - the camera projection matrix
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #    covarx - covariance matrix associated with x. Has size 2n x 2n, where␣
 ↪n is number of points.
    # Output:
    #    cost - Total reprojection error

    """your code here"""


    cost = np.inf
```

```
    return cost
```

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

# partial_x_partial_p unit test
def check_values_partial_x_partial_p():
    eps = 1e-8  # Floating point error threshold
    x_2d = np.load('Unit_test/x_2d.npy')
    P = np.load('Unit_test/Projection.npy')
    X = np.load('Unit_test/X.npy')
    target = np.load('Unit_test/dx_dp.npy')
    dx_dp = partial_x_partial_p(P,X,x_2d)
    valid = np.all((dx_dp < target + eps) & (dx_dp > target - eps))
    print(f'Computed partial_x_partial_p is all equal to ground truth +/- {eps}:
 ↪ {valid}')

# deparameterize_homog_with_Jacobian unit test
def check_values_partial_vbar_partial_v():
    eps = 1e-8  # Floating point error threshold
    p = np.load('Unit_test/p.npy')
    dp_dp_target = np.load('Unit_test/dp_dp.npy')
    p_bar_target = np.load('Unit_test/Projection.npy').reshape(12,1)
    p_bar,dp_dp = deparameterize_homog_with_Jacobian(p)
    valid_dp_dp = np.all((dp_dp < dp_dp_target + eps) & (dp_dp > dp_dp_target -␣
 ↪eps))
    valid_p_bar = np.all((p_bar < p_bar_target + eps) & (p_bar > p_bar_target -␣
 ↪eps))
    valid = valid_dp_dp & valid_p_bar
    print(f'Computed v_bar,partial_vbar_partial_v is all equal to ground truth␣
 ↪+/- {eps}: {valid}')

check_values_partial_x_partial_p()
check_values_partial_vbar_partial_v()
```

```
Computed partial_x_partial_p is all equal to ground truth +/- 1e-08: False
Computed v_bar,partial_vbar_partial_v is all equal to ground truth +/- 1e-08:
False
```

[6]:
```
def estimate_camera_projection_matrix_nonlinear(P, x, X, max_iters, lam):
    # Input:
    #    P - initial estimate of P
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #    max_iters - maximum number of iterations
    #    lam - lambda parameter
    # Output:
    #    P - Final P (3x4) obtained after convergence
```

```
    # data normalization
    covarx = np.eye(2*X.shape[1])
    x, T, covarx = data_normalize_with_cov(x, covarx)
    X, U = data_normalize(X)


    """your code here"""


    # you may modify this so long as the cost is computed
    # at each iteration
    for i in range(max_iters):









        cost = compute_cost(P, x, X, covarx)
        print ('iter %03d Cost %.9f'%(i+1, cost))

    # data denormalization
    P = np.linalg.inv(T) @ P @ U
    return P



# LM hyperparameters
lam = .001
max_iters = 100

# Run LM initialized by DLT estimate with data normalization
print ('Running LM with data normalization')
print ('iter %03d Cost %.9f'%(0, cost))
time_start=time.time()
P_LM = estimate_camera_projection_matrix_nonlinear(P_DLT, x, X, max_iters, lam)
time_total=time.time()-time_start
print('took %f secs'%time_total)

print("\n==Correct outputs==")
print("Begins at %.9f; ends at %.9f"%(84.104680130, 82.790238005))
```

```
Running LM with data normalization
iter 000 Cost inf
iter 001 Cost inf
iter 002 Cost inf
iter 003 Cost inf
```

```
iter 004 Cost inf
iter 005 Cost inf
iter 006 Cost inf
iter 007 Cost inf
iter 008 Cost inf
iter 009 Cost inf
iter 010 Cost inf
iter 011 Cost inf
iter 012 Cost inf
iter 013 Cost inf
iter 014 Cost inf
iter 015 Cost inf
iter 016 Cost inf
iter 017 Cost inf
iter 018 Cost inf
iter 019 Cost inf
iter 020 Cost inf
iter 021 Cost inf
iter 022 Cost inf
iter 023 Cost inf
iter 024 Cost inf
iter 025 Cost inf
iter 026 Cost inf
iter 027 Cost inf
iter 028 Cost inf
iter 029 Cost inf
iter 030 Cost inf
iter 031 Cost inf
iter 032 Cost inf
iter 033 Cost inf
iter 034 Cost inf
iter 035 Cost inf
iter 036 Cost inf
iter 037 Cost inf
iter 038 Cost inf
iter 039 Cost inf
iter 040 Cost inf
iter 041 Cost inf
iter 042 Cost inf
iter 043 Cost inf
iter 044 Cost inf
iter 045 Cost inf
iter 046 Cost inf
iter 047 Cost inf
iter 048 Cost inf
iter 049 Cost inf
iter 050 Cost inf
iter 051 Cost inf
```

```
iter 052 Cost inf
iter 053 Cost inf
iter 054 Cost inf
iter 055 Cost inf
iter 056 Cost inf
iter 057 Cost inf
iter 058 Cost inf
iter 059 Cost inf
iter 060 Cost inf
iter 061 Cost inf
iter 062 Cost inf
iter 063 Cost inf
iter 064 Cost inf
iter 065 Cost inf
iter 066 Cost inf
iter 067 Cost inf
iter 068 Cost inf
iter 069 Cost inf
iter 070 Cost inf
iter 071 Cost inf
iter 072 Cost inf
iter 073 Cost inf
iter 074 Cost inf
iter 075 Cost inf
iter 076 Cost inf
iter 077 Cost inf
iter 078 Cost inf
iter 079 Cost inf
iter 080 Cost inf
iter 081 Cost inf
iter 082 Cost inf
iter 083 Cost inf
iter 084 Cost inf
iter 085 Cost inf
iter 086 Cost inf
iter 087 Cost inf
iter 088 Cost inf
iter 089 Cost inf
iter 090 Cost inf
iter 091 Cost inf
iter 092 Cost inf
iter 093 Cost inf
iter 094 Cost inf
iter 095 Cost inf
iter 096 Cost inf
iter 097 Cost inf
iter 098 Cost inf
iter 099 Cost inf
```

```
iter 100 Cost inf
took 0.011111 secs

==Correct outputs==
Begins at 84.104680130; ends at 82.790238005
```

[7]:
```
# Report your P_LM (estimated camera projection matrix nonlinear) final value␣
 ↪here!
display_results(P_LM, x, X, 'P_LM')
```

```
P_LM =
[[ 5.61439661e-01 -4.31364493e-04  5.07262374e-02  3.74830054e-02]
 [-4.10862482e-03  6.12866520e-01  1.43106214e-02 -4.33078895e-02]
 [ 2.46814609e-02  2.97301747e-02  5.48968300e-01  1.56627803e-02]]
```