HW2

February 2, 2022

1 CSE 252B: Computer Vision II, Winter 2022 – Assignment 2

1.0.1 Instructor: Ben Ochoa

1.0.2 Due: Wednesday, February 2, 2022, 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 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.

1.2 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.

Solution-
From Plucker matrix representation, following is the point of intersection.
$X_L = \mathrm{L}\pi = (X_1X_2^T\text{-}X_2X_1^T)\pi = X_1X_2^T\pi$ - $X_2X_1^T\pi$

$X_L = X_1 X_2^T \pi$ - $X_2 X_1^T \pi$

From pencil of point representation,
$X(\lambda_\pi)^T \pi = 0$
$(\lambda_\pi X_1 + (1-\lambda_\pi) X_2)^T \pi = 0$
$X_2^T \pi = \lambda_\pi (X_1^T \pi - X_2^T \pi)$

Solving for λ_π
$\lambda_\pi = \frac{X_2^T \pi}{X_1^T \pi - X_2^T \pi}$
Substituting in $X(\lambda_\pi)$
$X(\lambda_\pi) = \frac{X_1 X_2^T \pi - X_2 X_1^T \pi}{X_2^T \pi - X_1^T \pi}$

Comparing the expressions for $X(\lambda_\pi)$ and X_L, it is evident that the 2 represent the same point upto scale.

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 $c_0 = \boldsymbol{X}_2^\top \boldsymbol{Q} \boldsymbol{X}_2$.

Solution-
$X(\lambda) = \lambda X_1 + (1-\lambda) X_2$
The point of intersection correspond to the values of λ that satisfy the following equation-
$X(\lambda)^T Q X(\lambda) = 0$
$(\lambda X_1 + (1-\lambda) X_2)^T Q (\lambda X_1 + (1-\lambda) X_2) = 0$

$\lambda^2 X_1^T Q X_1 + (1-\lambda)\lambda X_2^T Q X_1 + (1-\lambda)\lambda X_1^T Q X_2 + (1-\lambda)^2 X_2^T Q X_2 = 0$

$\lambda^2 (X_1^T Q X_1 - X_2^T Q X_1 - X_1^T Q X_2 + X_2^T Q X_2) + \lambda (X_2^T Q X_1 + X_1^T Q X_2 - 2X_2^T Q X_2) + X_2^T Q X_2 = 0$

Using the fact $X_1^T Q X_2 = X_2^T Q X_1$ for a symmetric Q

$\lambda^2 (X_1^T Q X_1 - 2X_1^T Q X_2 + X_2^T Q X_2) + \lambda (2X_1^T Q X_2 - 2X_2^T Q X_2) + X_2^T Q X_2 = 0$

comparing with $c_2 \lambda_Q^2 + c_1 \lambda_Q + c_0 = 0$

$c_2 = X_1^T Q X_1 - 2X_1^T Q X_2 + X_2^T Q X_2$
$c_1 = 2(X_1^T Q X_2 - X_2^T Q X_2)$
$c_0 = X_2^T Q X_2$

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 Normalize(pts):
    # data normalization of n dimensional pts
    #
    # Input:
    #    pts - is in inhomogeneous coordinates
    # Outputs:
    #    pts - data normalized points
    #    T - corresponding transformation matrix

    # Compute mean and variance of each dimension
    m = np.mean(pts,1).reshape(-1,1)
    v = np.var(pts,1).reshape(-1,1)
    s = np.sqrt(m.shape[0]/np.sum(v))

    # Create Transformation matrix
    T = np.eye(pts.shape[0]+1)
    for i in range(T.shape[0]-1):
        T[i,i] = s
        T[i,-1] = -m[i]*s

    # Normalize each set of points
    pts_h = Homogenize(pts)
```

```
    for i in range(pts.shape[1]):
        pts_h[:,i] = np.matmul(T, pts_h[:,i])
    return pts_h, T

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

    x_projected = Dehomogenize(np.matmul(P,Homogenize(X)))
    cost = np.sum(np.square(x-x_projected))
    return cost
```

[2]:
```
def computeKronickerTerm(x, X):
    '''
    Computes the left null-space of x and
    its kronicker product with X
    '''
    x = x.reshape(-1,1)
    X = X.reshape(-1,1)
    u, s, vh = np.linalg.svd(x, full_matrices=True)
    lns = u[:,s.shape[0]:x.shape[0]].T
    A = np.kron(lns,X.T)
    return A
```

[3]:
```
def DLT(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 = Normalize(x)
        X, U = Normalize(X)
    else:
        x = Homogenize(x)
        X = Homogenize(X)

    N = x.shape[1]
```

```
    A = []
    for i in range(N):
        A.append(computeKronickerTerm(x[:,i], X[:,i]))
    A = np.concatenate(A)
    _, _, vh = np.linalg.svd(A, full_matrices=True)
    P = vh[-1,:].reshape(3,4)

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

    return P

def displayResults(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 = DLT(x, X, normalize=False)
cost = ComputeCost(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 = DLT(x, X, normalize=True)
cost = ComputeCost(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 0.004465 secs
Cost=97.053719142
Running DLT with data normalization
took 0.004495 secs
Cost=84.104680130

==Correct outputs==
Cost=97.053718991 without data normalization
Cost=84.104680130 with data normalization
```

[4]:
```
# Report your P_LM final value here!
displayResults(P_DLT, x, X, 'P_DLT')
```

```
P_DLT =
[[ 6.04350846e-03 -4.84282446e-03  8.82395315e-03  8.40441373e-01]
 [ 9.09666810e-03 -2.30374203e-03 -6.18060233e-03  5.41657305e-01]
 [ 5.00625470e-06  4.47558354e-06  2.55223773e-06  1.25160752e-03]]
```

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.

[5]:
```
# Note that np.sinc is different than defined in class
def Sinc(x):
    # Returns a scalar valued sinc value
    if x == 0:
        y = 1
    else:
        y =  np.sin(x)/x
    return y

def Jacobian(P,p,X):
```

```
    # compute the jacobian matrix
    #
    # Input:
    #    P - 3x4 projection matrix
    #    p - 11x1 homogeneous parameterization of P
    #    X - 3n 3D scene points
    # Output:
    #    J - 2nx11 jacobian matrix

    P_ = Deparameterize(p).reshape(1,-1)

    # Compute derivative of derivative of deparametrized p wrt to
    # parameterized p

    if (np.linalg.norm(p) == 0):
        # Norm of p is 0
        da_dv = np.zeros((1,11))
        db_dv = np.eye(11)
    else:
        # da/dv
        da_dv = -0.5*P_[0,1:]
        # ||v|| term
        v_norm = np.linalg.norm(p)

        # derivative of sinc
        d_sinc = np.cos(v_norm/2)/(v_norm/2) -\
        np.sin(v_norm/2)/np.square(v_norm/2)

        # db/dv
        db_dv = Sinc(v_norm/2)/2*np.eye(11) + \
         d_sinc * np.matmul(p,p.T)/(4*v_norm)


    dp_bar_dp = np.vstack((da_dv, db_dv))
    assert dp_bar_dp.shape[0] == 12 and dp_bar_dp.shape[1] == 11 , "second term"

    J = []
    for i in range(X.shape[1]):
        X_ = X[:,i].reshape(-1,1)

        w = np.matmul(P[2,:].reshape(1,-1),X_)
        x_ = Dehomogenize(np.matmul(P, X_))

        dx_dp_1 = np.hstack((X_.T, np.zeros((1,4)),X_.T*-x_[0]))
        dx_dp_2 = np.hstack((np.zeros((1,4)), X_.T, -x_[1]*X_.T))

        dx_dp = np.vstack((dx_dp_1, dx_dp_2))/w
```

```
        assert dx_dp.shape[0] == 2 and dx_dp.shape[1] == 12, "first term"
        J.append(np.matmul(dx_dp, dp_bar_dp))

    J = np.concatenate(J,0)
    assert J.shape[0] == 2*X.shape[1] and J.shape[1] == 11 \
                      , "Jacobioan shape incorrect"
    return J


def Parameterize(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 ParameterizeHomog(P.reshape(-1,1))


def Deparameterize(p):
    # Deparameterize all optimization variables
    return DeParameterizeHomog(p).reshape(3,4)


def ParameterizeHomog(V):
    # Given a homogeneous vector V return its minimal parameterization

    v_hat = 2*V[1:]/Sinc(np.arccos(V[0]))

    if np.linalg.norm(v_hat) > np.pi:
        v_hat = (1 - 2*np.pi/np.linalg.norm(v_hat)*np.ceil((np.linalg.
↪norm(v_hat)-np.pi)/(2*np.pi)))*v_hat

    return v_hat


def DeParameterizeHomog(v):
    # Given a parameterized homogeneous vector return its deparameterization

    a = np.array([[np.cos(np.linalg.norm(v)/2)]])
    b = v*Sinc(np.linalg.norm(v)/2)/2
    v_bar = np.vstack((a,b))
    return v_bar

def Normalize_withCov(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

    # Compute mean and variance of each dimension
    m = np.mean(pts,1).reshape(-1,1)
    v = np.var(pts,1).reshape(-1,1)
    s = np.sqrt(m.shape[0]/np.sum(v))

    # Create Transform matrix
    T = np.eye(pts.shape[0]+1)
    for i in range(T.shape[0]-1):
        T[i,i] = s
        T[i,-1] = -m[i]*s

    # Normalize each set of points
    pts_h = Homogenize(pts)
    for i in range(pts.shape[1]):
        pts_h[:,i] = np.matmul(T, pts_h[:,i])

    # Covariance propagation
    covarx = (s*s)*covarx

    return pts_h, T, covarx

def ComputeCost_withCov(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

    x_hat = Dehomogenize(np.matmul(P,Homogenize(X)))
    m = x.ravel('F')
    m_hat = x_hat.ravel('F')
    cost = np.matmul(np.matmul((m-m_hat).T,np.linalg.inv(covarx)),(m-m_hat))
    return cost
```

[6]:
```
def LM(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 = Normalize_withCov(x, covarx)
    X, U = Normalize(X)

    # Data normalize P
    P = T @ P @ np.linalg.inv(U)

    # Initialize previous cost
    cost_prev = ComputeCost_withCov(P, Dehomogenize(x), Dehomogenize(X), covarx)

    # Initialize iterations and cost
    i = 0
    cost = 0

    while(i < max_iters):

        # Normalizie the camera projection matrix
        P = P/np.linalg.norm(P)*np.sign(P[-1,-1])

        # Parametrize p
        p = Parameterize(P)

        # Calculate Jacobian
        J = Jacobian(P,p,X)

        # Solve augmented normal equation
        J_T_Sigma = np.matmul(J.T, np.linalg.inv(covarx))

        epsilon = Dehomogenize(x) - Dehomogenize(np.matmul(P,X))
        epsilon = epsilon.ravel('F')

        # L*Delta = Q (Normal Equation)
        Q = np.matmul(J_T_Sigma, epsilon.T)
        L = np.matmul(J_T_Sigma,J) + lam*np.eye(11)
        delta = np.matmul(np.linalg.inv(L),Q)
        p_c = p + delta.reshape(-1,1)  # Candidate

        # De-parametrize P
        P_c = Deparameterize(p_c)
```

```
        cost = ComputeCost_withCov(P_c, Dehomogenize(x), Dehomogenize(X),␣
↪covarx)

        if (1 - cost/cost_prev < 1e-12):
            # Terminate if change in cost less than
            # threshold
            break

        if cost < cost_prev:
            # Valid iteration
            lam = lam/10
            P = P_c
            print ('iter %03d Cost %.9f'%(i+1, cost))
            i = i + 1
            cost_prev = cost
        else:
            # Update lambda
            lam = lam*10

    # 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 = LM(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 84.104680130
iter 001 Cost 82.791336044
iter 002 Cost 82.790238006
iter 003 Cost 82.790238005
took 0.051673 secs

==Correct outputs==
```

```
Begins at 84.104680130; ends at 82.790238005
```

[7]:
```
# Report your P_LM final value here!
displayResults(P_LM, x, X, 'P_LM')
```

```
P_LM =
[[ 6.09434291e-03 -4.72647758e-03  8.79023503e-03  8.43642842e-01]
 [ 9.02017241e-03 -2.29290824e-03 -6.13330068e-03  5.36660248e-01]
 [ 4.99088611e-06  4.45205073e-06  2.53705045e-06  1.24348254e-03]]
```