計算機視覺

Computer Vision

Homework II

系所:電子所碩二

中文姓名:李聖謙

學號:111063517

授課老師:Min Sun, 孫民

Contents

I. Camera Pose from Essential Matrix

1. estimate_initial_RT

```
def estimate_initial_RT(E):
    U, S, VT = np.linalg.svd(E)

    W = np.array([[0, -1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])

    Z = np.array([[0, 1, 0],
                  [-1, 0, 0],
                  [0, 0, 0]])

    M = np.dot(U, np.dot(Z, U.T))

    Q1 = np.dot(U, np.dot(W, VT))
    Q2 = np.dot(U, np.dot(W.T, VT))

    R1 = (np.linalg.det(Q1)) * Q1
    R2 = (np.linalg.det(Q2)) * Q2

    u3 = U[:, 2]
    T1 = u3
    T2 = -u3

    RT1 = np.hstack((R1, T1[:, np.newaxis]))
    RT2 = np.hstack((R1, T2[:, np.newaxis]))
    RT3 = np.hstack((R2, T1[:, np.newaxis]))
    RT4 = np.hstack((R2, T2[:, np.newaxis]))

    RTs = []
    RTs.append(RT1)
    RTs.append(RT2)
    RTs.append(RT3)
    RTs.append(RT4)

    return RTs
```

Part A result

```
--------------------------------------------------------------------------------
Part A: Check your matrices against the example R,T
--------------------------------------------------------------------------------
Example RT:
 [[ 0.9736 -0.0988 -0.2056  0.9994]
 [ 0.1019  0.9948  0.0045 -0.0089]
 [ 0.2041 -0.0254  0.9786  0.0331]]

Estimated RT:
 [array([[ 0.98305251, -0.11787055, -0.14040758,  0.99941228],
       [-0.11925737, -0.99286228, -0.00147453, -0.00886961],
       [-0.13923158,  0.01819418, -0.99009269,  0.03311219]]), array([[ 0.98305251, -0.11787055, -0.14040758, -0.99941228],
       [-0.11925737, -0.99286228, -0.00147453,  0.00886961],
       [-0.13923158,  0.01819418, -0.99009269, -0.03311219]]), array([[ 0.97364135, -0.09878708, -0.20558119,  0.99941228],
       [ 0.10189204,  0.99478508,  0.00454512, -0.00886961],
       [ 0.2040601 , -0.02537241,  0.97862951,  0.03311219]]), array([[ 0.97364135, -0.09878708, -0.20558119, -0.99941228],
       [ 0.10189204,  0.99478508,  0.00454512,  0.00886961],
       [ 0.2040601 , -0.02537241,  0.97862951, -0.03311219]])]
```

Coding method

The purpose of this question is to use the essential matrix, decomposed through Singular Value Decomposition (SVD), to derive four possibilities (R1T1, R2T1, R1T2, R2T2) using rotation matrices and translation matrices. From the results, it can be determined that the correct camera pose corresponds to R2T1.

II. Linear 3D Points Estimation

2. linear_estimate_3d_point

```python
def linear_estimate_3d_point(image_points, camera_matrices):
    M = len(image_points)
    A = np.zeros((2 * M, 4))

    for i in range(M):
```

$v*M^3_i$ M^2_i

```python
        # v
        A[2 * i, 0] = image_points[i][1] * camera_matrices[i][2, 0] - camera_matrices[i][1, 0]
        A[2 * i, 1] = image_points[i][1] * camera_matrices[i][2, 1] - camera_matrices[i][1, 1]
        A[2 * i, 2] = image_points[i][1] * camera_matrices[i][2, 2] - camera_matrices[i][1, 2]
        A[2 * i, 3] = image_points[i][1] * camera_matrices[i][2, 3] - camera_matrices[i][1, 3]
        # u
        A[2 * i + 1, 0] = camera_matrices[i][0, 0] - image_points[i][0] * camera_matrices[i][2, 0]
        A[2 * i + 1, 1] = camera_matrices[i][0, 1] - image_points[i][0] * camera_matrices[i][2, 1]
        A[2 * i + 1, 2] = camera_matrices[i][0, 2] - image_points[i][0] * camera_matrices[i][2, 2]
        A[2 * i + 1, 3] = camera_matrices[i][0, 3] - image_points[i][0] * camera_matrices[i][2, 3]
```

M^1_i $u*M^3_i$

```python
    U, S, VT = np.linalg.svd(A)
    # 取最後一行
    point_3d = VT[-1, :-1] / VT[-1, -1]

    return point_3d
```

Part B result

```
--------------------------------------------------------------------------------
Part B: Check that the difference from expected point
is near zero
--------------------------------------------------------------------------------
Difference:  0.0029243053036643873
```

Algorithm

After linear combination, we can get the matrix equation AP=0.

$$AP = 0 \Rightarrow \begin{bmatrix} vM_i^3 - M_i^2 \\ M_i^1 - uM_i^3 \end{bmatrix} \cdot P = 0$$

Performing SVD decomposition on matrix A, we obtain A = USVT. Since the equation is homogeneous, the rightmost row of VT is the solution for P.

Coding method

First create a zero matrix A of size (2M, 4), then use a for loop to calculate the linearly simplified matrix of the image_points and camera_matrices, fill in the A matrix one by one, and finally do SVD on the A matrix. And take the rightest row of VT as the solution of P.

III. Non-Linear 3D Points Estimation

3. reprojection_error

```python
def reprojection_error(point_3d, image_points, camera_matrices):
    M = len(image_points)
    error = np.empty((2 * M,))

    for i in range(M):
        Mi = camera_matrices[i]
        pi = image_points[i]

        P = np.append(point_3d, 1)

        y = np.dot(Mi, P)

        y3_inv = 1.0 / y[2]
        p_prime_i = y[:2] * y3_inv
        ei = p_prime_i - pi
        error[2 * i:2 * (i + 1)] = ei

    return error
```

$$P'_i = \begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{y_3}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

$$e = P'_i - P_i$$

4. jacobian

```python
def jacobian(point_3d, camera_matrices):

    M = camera_matrices.shape[0]
    jacobian = np.zeros((2 * M, 3))

    for i in range(M):
        P = camera_matrices[i]
        projected_point = np.dot(P, np.append(point_3d, 1))

        u, v, w = projected_point[0], projected_point[1], projected_point[2]

        dZ = P[2, 0:3]

        du_dX = P[0, 0] / w
        du_dY = P[0, 1] / w
        du_dZ = P[0, 2] / w

        dv_dX = P[1, 0] / w
        dv_dY = P[1, 1] / w
        dv_dZ = P[1, 2] / w

        jacobian[2 * i, :] = [du_dX, du_dY, du_dZ] - u * dZ / (w ** 2)
        jacobian[2 * i + 1, :] = [dv_dX, dv_dY, dv_dZ] - v * dZ / (w ** 2)

    return jacobian
```

Partial derivatives

Part C result

```
--------------------------------------------------------------------------------
Part C: Check that the difference from expected error/Jacobian
is near zero
--------------------------------------------------------------------------------
Error Difference:  8.301299988565727e-07
Jacobian Difference:  1.817113215452082e-08
```

Algorithm

The purpose of Reprojection Error is to calculate the different between the linear estimate of point_3d and its actual value. On the other hand, the Jacobian is computed by finding the partial derivatives of each value within the matrix, with the goal of refining the Reprojection Error. It is the combination of these two definitions that enables the calculation of Gauss-Newton optimization in the next definition.

Jacobian schematic diagram

$$y = M_i P$$

$$y = \begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & L \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad J = \begin{bmatrix} \frac{\partial e_1}{\partial X_1} & \frac{\partial e_2}{\partial Y_1} & \frac{\partial e_3}{\partial Z_1} \\ \vdots & \vdots & \vdots \\ \frac{\partial e_{2n}}{\partial X_n} & \frac{\partial e_{2n}}{\partial Y_n} & \frac{\partial e_{2n}}{\partial Z_n} \end{bmatrix} \qquad \begin{aligned} e_i &= P_i' - P_i \\ p' &= \frac{1}{y_3}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \end{aligned}$$

$$\frac{\partial}{\partial x}\left(\frac{aX+by+gZ+j}{cX+fy+iz+L}\right) = \frac{a(cX+fy+iz+L) - c(aX+by+gz+j)}{(cX+fy+iz+L)^2}$$

$$= \underbrace{\frac{a}{cX+fy+iz+L}}_{du\_dX,\ dv\_dx} - \underbrace{\frac{c(aX+by+gz+j)}{(cX+fy+iz+L)^2}}_{u \cdot dZ,\ v \cdot dZ}$$

$$\frac{\partial}{\partial y}\left(\frac{aX+by+gZ+j}{cX+fy+iz+L}\right) = \frac{b(cX+fy+iz+L) - f(aX+by+gz+j)}{(cX+fy+iz+L)^2}$$

$$= \underbrace{\frac{b}{cX+fy+iz+L}}_{du\_dY,\ dv\_dx} - \underbrace{\frac{f(aX+by+gz+j)}{(cX+fy+iz+L)^2}}_{u \cdot dZ,\ v \cdot dZ}$$

$$\frac{\partial}{\partial z}\left(\frac{aX+by+gZ+j}{cX+fy+iz+L}\right) = \frac{g(cX+fy+iz+L) - i(aX+by+gz+j)}{(cX+fy+iz+L)^2}$$

$$= \underbrace{\frac{g}{cX+fy+iz+L}}_{du\_dz,\ dv\_dx} - \underbrace{\frac{i(aX+by+gz+j)}{(cX+fy+iz+L)^2}}_{u \cdot dZ,\ v \cdot dZ}$$

Coding method

For Reprojection Error, start by creating an empty error matrix of size (2M,). Then, use a for loop to scan through all the image points. For each point, take the inner product of Mi and P, and substitute them into $e = P'_i - P_i$, and finally populate the results into the error empty matrix.

For jacobian, create a (2M, 3) matrix and use a for loop to calculate du/dX, du/dY and du/dZ, which represent the partial derivatives of u with respect to the x, y and z coordinates respectively. dv/dX, dv/dY and dv/dZ, which represent the partial derivatives of v with respect to the x, y and z coordinates of the three-dimensional point respectively, and finally fill in the empty matrix with these calculation results.

5. nonlinear_estimate_3d_point

```
def nonlinear_estimate_3d_point(image_points, camera_matrices):

    linear_ppoint = linear_estimate_3d_point(image_points, camera_matrices)
    point_3d = linear_ppoint

    for iteration in range(10):

        error = reprojection_error(point_3d, image_points, camera_matrices)
        J = jacobian(point_3d, camera_matrices)

        Hessian = np.dot(J.T, J)

        Hessian_inv = np.linalg.inv(Hessian)
        gradient = np.dot(J.T, error)
        step = np.dot(Hessian_inv, gradient)

        point_3d -= step

    return point_3d
```

Import 3d point

Import Reprojection error and jacobian definition

$P = P - (J^T J)^{-1} J^T e$

Part D result

```
--------------------------------------------------------------------------------
Part D: Check that the reprojection error from nonlinear method
is lower than linear method
--------------------------------------------------------------------------------
Linear method error: 98.7354235689419
Nonlinear method error: 95.59481784846034
```

Algorithm

The non-linear calculations utilize the Gauss-Newton approach to optimize the estimated values obtained from linear calculations. The results show that the non-linear errors are indeed lower compared to the linear ones.

```
--------------------------------------------------------------------------------
Part D: Check that the reprojection error from nonlinear method
is lower than linear method
--------------------------------------------------------------------------------
[-0.9415765  -4.6160174   4.42533523]
[-0.92042469 -4.6399826   4.47362616]
[-0.92056391 -4.63993193  4.47303332]
[-0.9205578  -4.63993565  4.47305161]
[-0.92055795 -4.63993556  4.47305115]
[-0.92055794 -4.63993556  4.47305116]
[-0.92055794 -4.63993556  4.47305116]
[-0.92055794 -4.63993556  4.47305116]
[-0.92055794 -4.63993556  4.47305116]
[-0.92055794 -4.63993556  4.47305116]
```

10 iterations result

After 10 iterations, the non-linear calculations have effectively removed excess noise and eventually converged to a constant value.

IV. Decide the Correct RT

6. estimate_RT_from_E

```
def estimate_RT_from_E(E, image_points, K):

    U, S, Vt = np.linalg.svd(E)

    W = np.array([[0, -1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])

    Q1 = np.dot(U, np.dot(W, Vt))
    Q2 = np.dot(U, np.dot(W.T, Vt))

    R1 = (np.linalg.det(Q1)) * Q1
    R2 = (np.linalg.det(Q2)) * Q2

    t = U[:, 2]
    T1 = t
    T2 = -t

    RT1 = np.dot(-(R1.T) , T1)
    RT2 = np.dot(-(R1.T) , T2)
    RT3 = np.dot(-(R2.T) , T1)
    RT4 = np.dot(-(R2.T) , T2)

    RT5 = np.hstack((R1.T, RT1[:, np.newaxis]))
    RT6 = np.hstack((R1.T, RT2[:, np.newaxis]))
    RT7 = np.hstack((R2.T, RT3[:, np.newaxis]))
    RT8 = np.hstack((R2.T, RT4[:, np.newaxis]))

    M1= np.dot(np.linalg.inv(K) , RT5)
    M2= np.dot(np.linalg.inv(K) , RT6)
    M3= np.dot(np.linalg.inv(K) , RT7)
    M4= np.dot(np.linalg.inv(K) , RT8)

    X, Y, Z = image_points.shape
    count_R1_T1 = 0
    count_R1_T2 = 0
    count_R2_T1 = 0
    count_R2_T2 = 0

    for i in range(X):
        for j in range(Y):
            M = np.append(image_points[i, j, :], 1)
            R1T1 = M1[:, :-1]
            R1_T1 = M - M1[:, -1]
            R1T2 = M2[:, :-1]
            R1_T2 = M - M2[:, -1]
            R2T1 = M3[:, :-1]
            R2_T1 = M - M3[:, -1]
            R2T2 = M4[:, :-1]
            R2_T2 = M - M4[:, -1]

            Z1= np.linalg.lstsq(R1T1, R1_T1, rcond=None)[0]
            Z2 = np.linalg.lstsq(R1T2, R1_T2, rcond=None)[0]
            Z3 = np.linalg.lstsq(R2T1, R2_T1, rcond=None)[0]
            Z4 = np.linalg.lstsq(R2T2, R2_T2, rcond=None)[0]
```

```python
            if Z1[2] > 0:
                count_R1_T1 += 1
            if Z2[2] > 0:
                count_R1_T2 += 1
            if Z3[2] > 0:
                count_R2_T1 += 1
            if Z4[2] > 0:
                count_R2_T2 += 1

    max_z = max(count_R1_T1, count_R1_T2, count_R2_T1, count_R2_T2)

    if max_z == count_R1_T1 :
        R = R1
        T = T2
    elif max_z == count_R1_T2 :
        R = R1
        T = T2
    elif max_z == count_R2_T1 :
        R = R2
        T = T1
    elif max_z == count_R2_T2 :
        R = R2
        T = T2

    RT = np.hstack((R, T.reshape(3, 1)))
    return RT
```

Part E result

```
--------------------------------------------------------------------------------
Part E: Check your matrix against the example R,T
--------------------------------------------------------------------------------
Example RT:
 [[ 0.9736 -0.0988 -0.2056  0.9994]
 [ 0.1019  0.9948  0.0045 -0.0089]
 [ 0.2041 -0.0254  0.9786  0.0331]]
!
Estimated RT:
 [[ 0.97364135 -0.09878708 -0.20558119  0.99941228]
 [ 0.10189204  0.99478508  0.00454512 -0.00886961]
 [ 0.2040601  -0.02537241  0.97862951  0.03311219]]
```

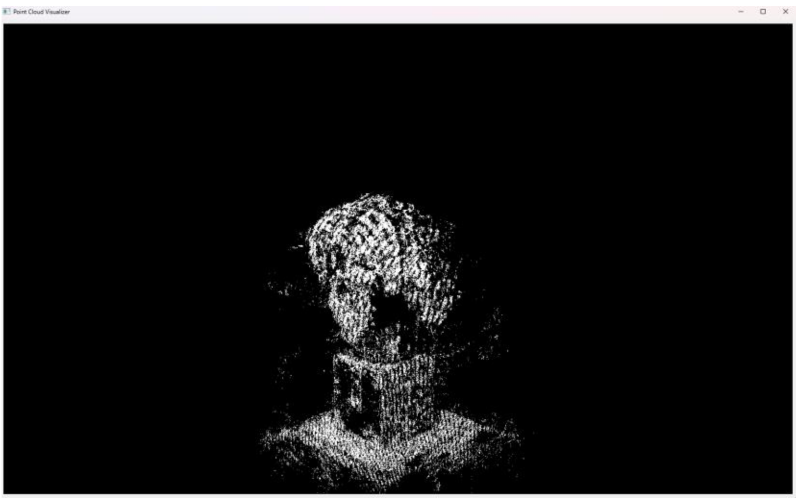
Coding method

This question is like the first question, but the method is different. It uses the Essential matrix (E) and two camera corrections (K) to predict the RT matrix of rotation and translation. According to the meaning of the question, four RT matrices can be obtained. For the correct RT, triangulated point P exists in front of both cameras, which means that it has a positive z-coordinate with respect to both camera reference systems. The equation we need is $M' = K'[R^T \quad -R^T T]$. In the program, I added one to the count of the four cameras if the Z coordinate is positive, and then finally found the maximum value among the four. The result shows that the matrix is the correct value when RT is R2T1.

V. Result

Front (zoom out, looks whiter)

Front (Zoom in, looks darker)

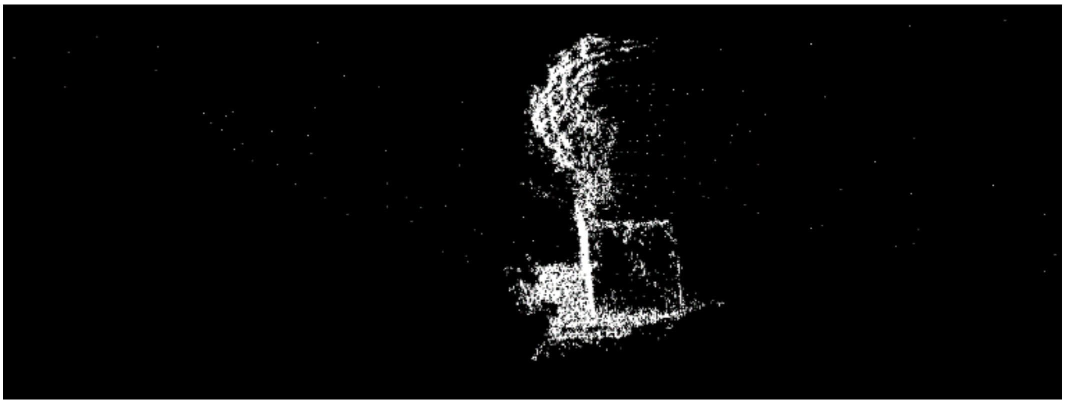


Side

VI. Discussion

This homework is considerably more challenging than the first one, as our lab has recently been involved in research related to drone. There have also been discussions about 3D modeling. However, in our lab, we had acquired depth cameras for our research. In this homework, we are directly using coding to reconstruct images of objects, which is a technology I haven't encountered before.

This assignment is more like math than writing code. If you don’t first understand the meaning of each matrix and the relationship between each definition, it is easy for some small bugs to affect the results, such as the final When I first wrote the nonlinear definition, I forgot to use the point_3d from linear definition, so I started from the empty matrix and entered the calculation. This resulted in the final point cloud diagram being exactly the opposite of the correct answer (as shown below).



Fault result