國立清華大學

計算機視覺

Computer Vision

國立清華大學

NATIONAL TSING HUA UNIVERSITY

Homework 2

系所級:電子所二年級

學號:111063548

姓名:蕭方凱

指導老師:孫民教授

目錄

1. Implementation

1.1 Camera Pose from Essential Matrix

define Estimate_initial_RT ():

```python
def estimate_initial_RT(E):
    # 步驟1: 奇異值分解 ( SVD )
    U, D, Vt = np.linalg.svd(E)

    # 定義 W 和 Z 矩陣
    W = np.array([[0, -1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])

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

    # 計算旋轉矩陣 R
    Q1 = U @ W @ Vt
    Q2 = U @ W.T @ Vt
    # print("Matrix Q1:\n", Q1)
    # print("Matrix Q2:\n", Q2)
    det_Q1 = np.linalg.det(Q1)
    det_Q2 = np.linalg.det(Q2)
    R1 = det_Q1 * Q1
    R2 = det_Q2 * Q2
    # print("R1:\n", R1)            # shape:(3, 3)
    # print("R2:\n", R2)            # shape:(3, 3)

    # 步驟2: 估算平移向量 T
    u3 = U[:, 2]
    T1 = u3                         # shape:(3,)
    T2 = -u3                        # shape:(3,)
    T1_add = T1[:, np.newaxis]      # shape:(3,1)
    T2_add = T2[:, np.newaxis]      # shape:(3,1)

    # 步驟3: 返回四種可能的組合
    RT1 = np.hstack((R1, T1_add))
    RT2 = np.hstack((R1, T2_add))
    RT3 = np.hstack((R2, T1_add))
    RT4 = np.hstack((R2, T2_add))

    return np.array([RT1, RT2, RT3, RT4])
```

Fig 1 estimate initial RT ()

U and matrix V, and Q1 = UWV^T; Q2 = UW^TV^T. Through Q1 and Q2, rotation matrices R1 and R2 can be obtained. The translation matrix T is the last column array of matrix U. It has two possibilities, positive and negative, namely T1 and T2, which represent as T1 = U[:, 2], T2 = -U[:, 2].Therefore, there are four RT combinations, all of which will be returned in this question.

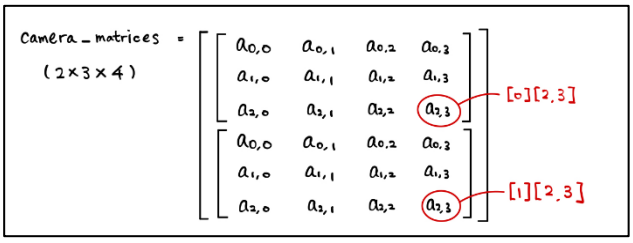
1.2 Linear 3D Points Estimation

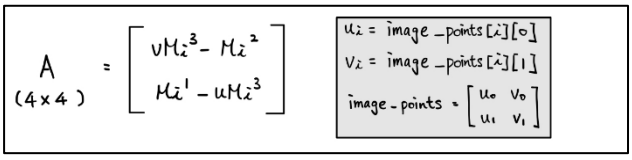
define linear estimate 3d point ():

```
def linear_estimate_3d_point(image_points, camera_matrices):
    M = len(image_points)
    A = np.zeros((2 * M, 4))
    # print(image_points.shape)
    # print(image_points)
    # print(camera_matrices.shape)
    # print(camera_matrices)
    # print(camera_matrices[0][2, 0])
    # print("")
    for i in range(M):
        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]
        # print('A', A)
        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]
        # print('A', A)


    # Perform SVD
    _, _, Vt = np.linalg.svd(A)

    # The 3D point is the last column of Vt
    point_3d = Vt[-1, :-1] / Vt[-1, -1]

    return point_3d
```

Fig 2 linear estimate 3d point ()

This question is to find the positions of 3D points through direct linear transformation. First, input image points and camera matrices. The matrix size is as follows:





According to the formula $\begin{bmatrix} vM_i^3 - M_i^2 \\ M_i^1 - uM_i^3 \end{bmatrix} P = 0$，P is the 3D point to be found。First calculate $A = \begin{bmatrix} vM_i^3 - M_i^2 \\ M_i^1 - uM_i^3 \end{bmatrix}$，then perform singular value decomposition operation on matrix A to obtain matrix Vt, and finally add the last column of matrix Vt (fourth column)，take out a 3x1 array from row 1 to row 3, and divide it by the last column of the last column.

1.3 Non-Linear 3D Points Estimation

Since the linear estimation from SVD is not robust to noise, we need to further apply non-linear optimization to our estimated points. Following are the non-linear 3D points estimation methods.

define reprojection error ():

```
def reprojection_error(point_3d, image_points, camera_matrices):
    num_images = len(camera_matrices)   # len(camera_matrices) = 2
    error = np.array([])

    for i in range(num_images):
        Mi = camera_matrices[i]
        u, v = image_points[i]

        # 3D location of a point 3D點的坐標
        P = np.hstack((point_3d, 1))
        # print(matrix)
        # print("")

        # 將3D點P投影到圖像平面，計算重投影點
        projected_point = Mi @ P          # (3, 4)*(4, 1) = (3, 1)
        projected_u = projected_point[0] / projected_point[2]
        projected_v = projected_point[1] / projected_point[2]

        # 計算重投影誤差
        error = np.append(error, projected_u - u)
        error = np.append(error, projected_v - v)

    return error
```

Fig 3 reprojection error ()

Reprojection error is the distance between a 3D point projected onto a 2D image and the actual measured 2D point.

How to Calculate the reprojection error?

First, calculate y = MiP. This Y is a 3x1 matrix. Using this Y matrix, we can calculate the projected image coordinate pi' of P. The formula is as follows:

$$p_i' = \begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{y3} \begin{bmatrix} y1 \\ y2 \end{bmatrix}$$

The ground truth projected image coordinate pi can be obtained through the input image points. Subtract pi from pi', which is the matrix error of the above code is the reprojection error.

define Jacobian():

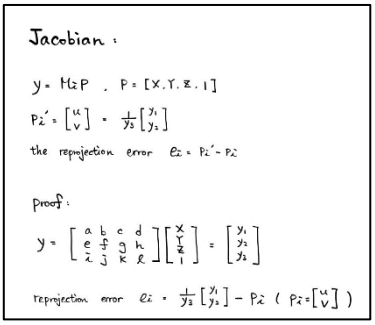
```
def jacobian(point_3d, camera_matrices):
    num_views = len(camera_matrices)            # M = num_views = 2
    num_projections = 2 * num_views             # num_projections = 4
    jacobian = np.zeros((num_projections, 3))   # (4, 3)

    # Iterate through each view
    for i in range(num_views):
        Mi = camera_matrices[i]                 # (3, 4)
        # print(Mi)
        P = np.append(point_3d, 1)              # P = [X Y Z 1] (4, 1)
        y = Mi @ P                              # y = MiP
        # print("MiP:\n", y)

        jacobian[2 * i, :] = Mi[0, 0:3] / y[2] - Mi[2, 0:3] * y[0] / (y[2] ** 2)
        jacobian[2 * i + 1, :] = Mi[1, 0:3] / y[2] - Mi[2, 0:3] * y[1] / (y[2] ** 2)
    # print("(jacobian)jacobian result:\n", jacobian)
    return jacobian
```

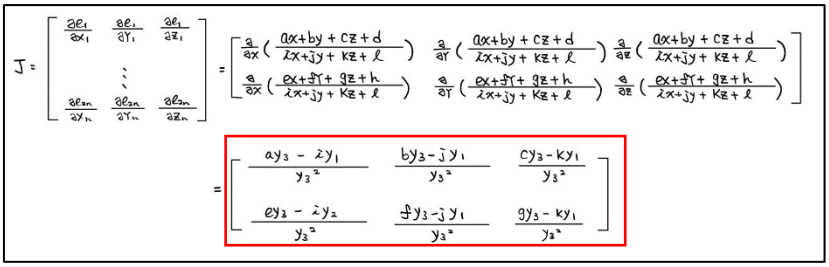
Fig 4 Jacobian

According to the Jacobian formula:

$$J = \begin{bmatrix} \frac{\partial e1}{\partial x1} & \frac{\partial e1}{\partial y1} & \frac{\partial e1}{\partial z1} \\ \vdots & \vdots & \vdots \\ \frac{\partial e2n}{\partial xn} & \frac{\partial e2n}{\partial yn} & \frac{\partial e2n}{\partial zn} \end{bmatrix}$$

Calculate the input point3d(P) and camera matrices(Mi) to obtain y = MiP, and use the formula in the red box below to find the Jacobian matrix. The guidance process is as follows:





def nonlinear estimate 3d point:

```
def nonlinear_estimate_3d_point(image_points, camera_matrices):
    point_3d = linear_estimate_3d_point(image_points, camera_matrices)
    for i in range(10):
        error = reprojection_error(point_3d, image_points, camera_matrices)
        # print("(nonlinear_estimate_3d_point)error:\n", error)

        jacobian_result = jacobian(point_3d, camera_matrices)
        # print("(nonlinear_estimate_3d_point)jacobian result:\n", jacobian_result)

        J_T = jacobian_result.T
        JTJ = np.dot(J_T, jacobian_result)

        J_inv = np.linalg.inv(JTJ)
        # print(point_3d)
        point_3d = point_3d - np.dot(np.dot(J_inv , J_T), error)
        # print("(iteraion ", i, ")", "point_3d: ", point_3d)
    return point_3d
```

Fig 5 nonlinear estimate 3d point

Use the linearly estimated points from def linear estimate 3d point as the initial guess, and the Gauss-Newton optimization can be simply implemented by:

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

which P is point_3d in the code, e is error in the code.

After calculating the nonlinear estimated 3d point, you can use it and the linear estimated 3d point as the input point_3d of the reprojection error respectively. Calculate the linear error value and nonlinear error value.

The results are as following:

```
--------------------------------------------------------------------------------
Part D: Check that the reprojection error from nonlinear method
is lower than linear method
--------------------------------------------------------------------------------
Linear method error: 98.7354235689419
Nonlinear method error: 95.59481784846032
--------------------------------------------------------------------------------
```

You will find that the nonlinear error is smaller, which represents the 3d points calculated using reprojection_error and jacobian is more accurate than direct linear transformation(DLT method).

1.4 Decide the Correct RT

def estimate_RT_from_E:

```
def estimate_RT_from_E(E, image_points, K):
    # Singular Value Decomposition of the Essential Matrix
    U, D, Vt = np.linalg.svd(E)

    # Calculate the rotation matrices R1 and R2
    W = np.array([[0, -1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])

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

    Q1 = U @ W @ Vt
    Q2 = U @ W.T @ Vt
    det_Q1 = np.linalg.det(Q1)
    det_Q2 = np.linalg.det(Q2)
    R1 = det_Q1 * Q1
    R2 = det_Q2 * Q2

    u3 = U[:, 2]
    T1 = u3                          # shape:(3,)
    T2 = -u3                         # shape:(3,)
    T1_add = T1[:, np.newaxis]       # shape:(3,1)
    T2_add = T2[:, np.newaxis]       # shape:(3,1)

    RT = np.hstack((R2, T1_add))
    # projected_points = np.dot(K, RT @ image_points.reshape(4, 1))
    # z_coordinates = projected_points[2, :]
    # print(z_coordinates)

    # print(RT)
    return RT
```

Fig 6 estimate_RT_from_E

RT has four combinations, namely R1T1, R1T2, R2T1, R2T2.The correct R, T is the one in which there are most projected 3D points having positive z-coordinate for both images. By observing the result graph can also determine which is the best RT combination.

Although I don't know if it is correct, calculating "**projected_points = np.dot(K, RT @ image_points.reshape(4, 1))**" can be used to determine which RT combination of four RTs has a positive z coordinate. Through this method, it is found that R1 will definitely make the z coordinate a negative value. **Finally, found that R2T1 is the best combination**.

2. Results

Example result:

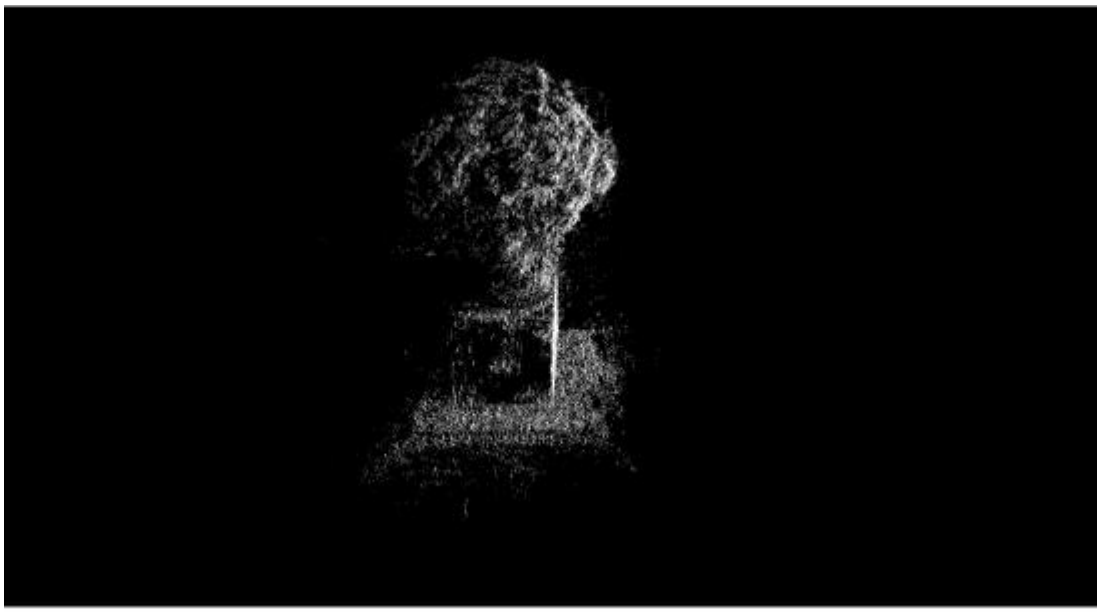
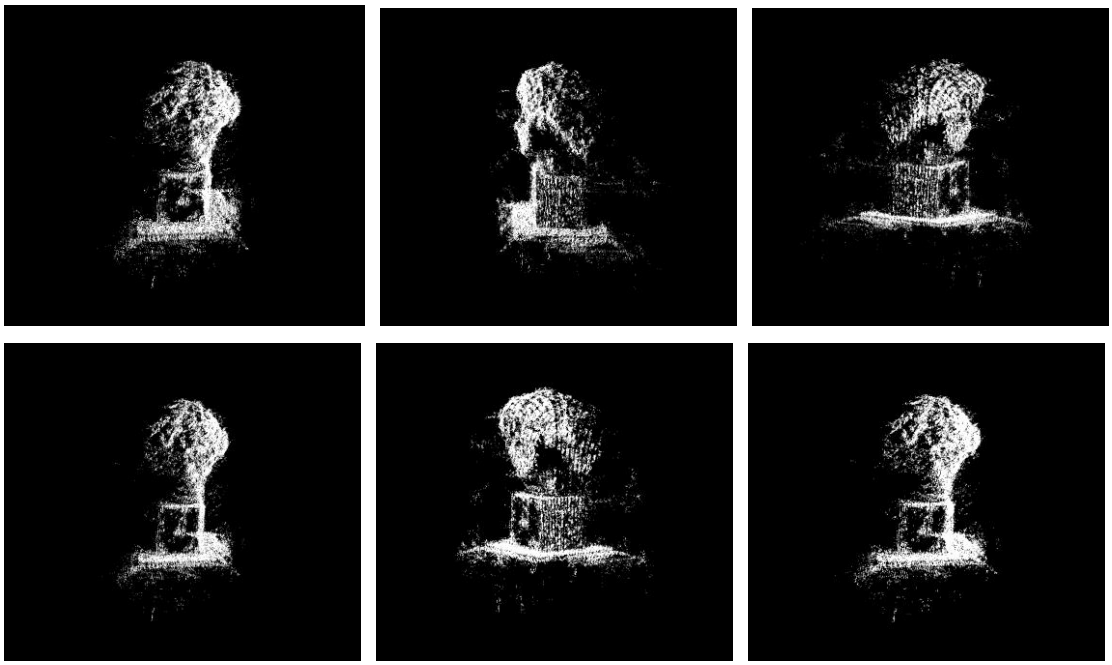


Fig 7 Result from example

My results:



These six pictures are results (The 3D point cloud visualization) from various angles, and the effects presented are close to the figures required by the assignment.

3. Discussion

This assignment requires drawing the 3D point cloud of a statue, inputting five images of the statue at different angles, and outputting the 3D point cloud visualization of the statue.

The def function is mainly divided into finding the RT combination, using the two operation methods of linear estimate and nonlinear estimate to find the 3D point of the image to be projected, and comparing the excellence of linear estimate and nonlinear estimate.

In the linear estimate part, the direct linear transformation method is used to find the 3d point, while in the nonlinear part, the 3d points of the linear estimate are first used as the starting value, and then the jacobian method is used to find the 3d point, and then the 3d points found respectively are put into the reprojection error to estimate the error value, you will find that nonlinear method performs better since the linear estimation from SVD is not robust to noise.

To find the most suitable RT process, we need to obtain its corresponding z coordinate, how to find the z coordinate is the difficult part of this assignment. Trying many methods, including applying reprojection error to determine which error value is the smallest, but there is still no suitable algorithm to get the correct RT which its z coordinate is the only one that possible. In the end, this assignment is to directly apply the four possibilities and observe the result graph to confirm which RT is the best combination.