3.1 Camera Pose from Essential Matrix (20%)

首先使用奇異值分解（SVD）將 essential matrix (E) 分解為三個矩陣 U、D 和 V^T，得到 essential matrix 的特征分解，其中 U 和 V^T是正交矩陣，而 D 包含了奇異值。

```
U, D, VT = np.linalg.svd(E)
```

然後使用一個旋轉矩陣，用於計算兩種可能的旋轉矩陣。這個旋轉矩陣表示繞 Z 軸旋轉 90 度的操作。

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$Q = UWV^T \text{ or } UW^TV^T$$

並使用公式算出 Q1 和 Q2

```
W = np.array([[0, -1, 0], [1, 0, 0], [0, 0, 1]])
Q_1 = U.dot(W.dot(VT))
Q_2 = U.dot(W.T.dot(VT))
```

算出後，將檢查旋轉矩陣 Q1 和 Q2 的行列式是否小於 0。如果行列式小於 0，表示矩陣能包含反射變換，因此需要對其進行修正，即乘以 -1，以確保其是一個合法的旋轉矩陣。

```
if np.linalg.det(Q_1) < 0:
   Q_1 = - Q_1
if np.linalg.det(Q_2) < 0:
   Q_2 = - Q_2
```

接著分別獲取平移向量的兩個可能值，平移向量通常是 essential matrix 的第三列。

```
# Create the two possible translation vectors (T1 and
T2)
T1 = U[:, 2] # Third column of U
T2 = -U[:, 2] # Negative of the third column of U
```

最後創建了四個可能的相機變換矩陣，其中包含了兩種旋轉矩陣（Q1 和 Q2）和兩種平移向量（T1 和 T2）。這些變換矩陣描述了照像機之間的相對運動。

```
# Create the four possible transformation matrices
    RT = np.array([
        np.vstack([Q_1.T, T1]).T,
        np.vstack([Q_1.T, T2]).T,
        np.vstack([Q_2.T, T1]).T,
        np.vstack([Q_2.T, T2]).T
    ])
```

測試結果:

```
--------------------------------------------------
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:
 [[[ 0.98305251 -0.11787055 -0.14040758  0.99941228]
  [-0.11925737 -0.99286228 -0.00147453 -0.00886961]
  [-0.13923158  0.01819418 -0.99009269  0.03311219]]

 [[ 0.98305251 -0.11787055 -0.14040758 -0.99941228]
  [-0.11925737 -0.99286228 -0.00147453  0.00886961]
  [-0.13923158  0.01819418 -0.99009269 -0.03311219]]

 [[ 0.97364135 -0.09878708 -0.20558119  0.99941228]
  [ 0.10189204  0.99478508  0.00454512 -0.00886961]
  [ 0.2040601  -0.02537241  0.97862951  0.03311219]]

 [[ 0.97364135 -0.09878708 -0.20558119 -0.99941228]
  [ 0.10189204  0.99478508  0.00454512  0.00886961]
  [ 0.2040601  -0.02537241  0.97862951 -0.03311219]]]
```

3.2 Linear 3D Points Estimation (20%)

首先檢查影像點的數量 M 是否等於相機矩陣的數量，如果不相等，則會引發 ValueError，確保輸入數據的一致性。

```
M = len(image_points)

if len(camera_matrices) != M:
   raise ValueError("Number of camera matrices must match the number of image points.")
```

再來創建一個空的矩陣A，將用來存儲方程式，這個矩陣將有$2 * M$ 行和 4 列。

```
# Create an empty matrix to store the equations
A = np.zeros((2 * M, 4))
```

接著通過循環處理每個影像點，它從 image_points 中提取影像點的坐標 (u, v) 和相機矩陣 P。在每次循環中，它填充矩陣 A 的行。對於每個影像點，它將兩行分別添加到矩陣 A，這兩行將根據以下公式計算：

$$第\ 2 * i\ 行：u * P[2] - P[0]$$

$$第\ 2 * i + 1\ 行：v * P[2] - P[1]$$

這些方程式是來自於透視投影的幾何關係。

```
for i in range(M):
# Extract the image point coordinates (u, v) and the camera matrix (P)
   u, v = image_points[i]
   P = camera_matrices[i]
   # Fill the rows of the matrix A
   A[2 * i] = u * P[2] - P[0]
   A[2 * i + 1] = v * P[2] - P[1]
```

接下來使用奇異值分解 (SVD) 來解決線性系統。通過調用 **np.linalg.svd(A)**，它取得矩陣 A 的奇異值分解，並取得右奇異向量矩陣 V。

```
# Solve the linear system using SVD
_, _, V = np.linalg.svd(A)
```

從 V 中，選擇最小奇異值相對應的右奇異向量，將其儲存在 estimated_3d_point_homogeneous 中。

```
# The 3D point is the right singular vector
corresponding to the smallest singular value
estimated_3d_point_homogeneous = V[-1]
```

接著，對 estimated_3d_point_homogeneous 進行歸一化，以確保最後一個元素為 1，得到歸一化的齊次坐標。

```
# Normalize the homogeneous coordinates (set the last element to 1)
estimated_3d_point_homogeneous /= estimated_3d_point_homogeneous[3]
```

最後，提取 estimated_3d_point_homogeneous 中的前三個元素，得到非齊次的三維坐標 estimated_3d_point。

```
# Extract the non-homogeneous 3D coordinates
estimated_3d_point = estimated_3d_point_homogeneous[:3]
```

測試結果:

```
Part B: Check that the difference from expected point
is near zero
-------------------------------------------------------
Difference:  0.0029243053036863698
```

3.3 Non-Linear 3D Points Estimation (20%)

首先關於 Reprojection error:

M變數代表影像點的數量，也就是 $image\_points$ 列表的長度，它用來確定有多少個影像點。

```
M = len(image_points) # Determine the number of image points
```

P 變數是一個包含三維點的坐標$[X, Y, Z, 1]$的向量，其中 X、Y 和 Z 分別代表三維點的坐標，最後的 1 是一個齊次座標。

```
P = np.hstack([point_3d, 1]) # P = [X, Y, Z, 1] is
the 3D location of a point
```

M_i變數是相機投影矩陣(camera matrices)的列表，代表不同的相機。通常，每個M_i包含相機的內部參數和外部參數，用於將三維點投影到影像平面。

```
Mi = camera_matrices[:]
```

接下來，函數計算 $y = M_i P$ ，這個步驟將三維點 P 透過相機投影矩陣(camera matrices)投影到影像平面，生成一個包含投影影像點的向量 y。

```
# compute y = Mi*P
# Mi is projection matrix
y = np.matmul(Mi, P)
```

然後，對 y 函數進行歸一化，確保投影影像點的最後一個元素為 1，以處理齊次座標。

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

```
# Normalize the projected points
y = y.T
projected_image_coordinate = y / y[-1, :]
```

接下來計算投影誤差

$$e_i = p_i' - p_i \ ,$$

```
# Calculate the reprojection error
reprojection_error = projected_image_coordinate[:-1, :].T - image_points
```

最後，將投影誤差函數轉換成一維向量，將其重塑為 2M 長度的一維向量。

```
# Convert to a 2Mx1 vector
reprojection_error = reprojection_error.reshape(2 * M, )
```

關於 Jacobian:

num_cameras 變數是相機矩陣的數量，通常每個相機矩陣用於不同的相機，這個變數會用來確定雅克比矩陣的大小。

創建一個 jacobian 矩陣，用於存儲雅克比矩陣的結果。這個矩陣的大小為$(2 * num_cameras, 3)$，其中$2 * num_cameras$ 代表每個相機都有一個 x 和一個 y 方向的雅克比。

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

進入 for 迴圈，對每個相機進行操作，提取當前相機 M，創建齊次三維點坐標 P，並計算一個常見的分母(denominator)，接著計算雅克比矩陣中 x 和 y 方向的元素。這些元素計算了函數中 x 和 y 分量的偏導數，最後將計算出的 x 和 y 方向的雅克比值除以分母

(denominator)。將計算得到的 x 和 y 方向的雅克比值分別分配給 jacobian 矩陣，以形成完整的雅克比矩陣。

```
for i in range(num_cameras):
# Extract the current camera matrix
M = camera_matrices[i]
# Homogeneous 3D point coordinates
P = np.hstack([point_3d, 1])
# Compute the denominator for the common factor
denominator = (np.dot(M[2], P)) ** 2
# Compute the elements of the Jacobian for the x and y components
dx = np.array([
M[0, 0] * np.dot(M[2, 1:], P[1:4]) - M[2, 0] * np.dot(M[0, 1:],
P[1:4]),
M[0, 1] * np.dot(M[2, [0, 2, 3]], P[[0, 2, 3]]) - M[2, 1] * np.dot(M[0,
[0, 2, 3]], P[[0, 2, 3]]),
M[0, 2] * np.dot(M[2, [0, 1, 3]], P[[0, 1, 3]]) - M[2, 2] * np.dot(M[0,
[0, 1, 3]], P[[0, 1, 3]])
])
dy = np.array([
M[1, 0] * np.dot(M[2, 1:], P[1:4]) - M[2, 0] * np.dot(M[1, 1:],
P[1:4]),
M[1, 1] * np.dot(M[2, [0, 2, 3]], P[[0, 2, 3]]) - M[2, 1] * np.dot(M[1,
[0, 2, 3]], P[[0, 2, 3]]),
M[1, 2] * np.dot(M[2, [0, 1, 3]], P[[0, 1, 3]]) - M[2, 2] * np.dot(M[1,
[0, 1, 3]], P[[0, 1, 3]])
])
# Divide by the common denominator
dx /= denominator
dy /= denominator
# Assign the computed values to the Jacobian matrix
jacobian[2 * i] = dx
jacobian[2 * i + 1] = dy
```

最後 Optmization:

設定跌代 10 次，並容忍的收斂誤差設 1e-6，用於控制迭代的停止條件。

```
max_iterations = 10
tolerance = 1e-6
```

接著使用前面的函式

```
estimated_3d_point = linear_estimate_3d_point(pi, Mi)
```

並初始化投影誤差(prev_reprojection_error)為正無窮大，以便在迭代過程中跟蹤前一次的重投影誤差。

```
prev_reprojection_error = float('inf')
```

進入跌代運算，在每次迭代中，首先計算雅克比矩陣 J，然後計算當前估計的三維點位置下的投影誤差(prev_reprojection_error)。然後使用最小二乘法(np.linalg.lstsq)，將雅克比矩陣 J 和投影誤差(prev_reprojection_error)用於更新三維點位置(estimated_3d_point)。計算當前迭代下的投影誤差，並檢查是否收斂。如果新的投影誤差和前一次的誤差之間的差異小於收斂誤差，則停止迭代。如果未達到停止條件，則更新投影誤差(prev_reprojection_error)，增加迭代計數，並繼續下一輪迭代。最終，返回估計的三維點位置(estimated_3d_point)，它應該是在最小化投影誤差的過程中優化得到的值。

```
while iteration < max_iterations:
    J = jacobian(estimated_3d_point, Mi)
    reprojection_error_ = reprojection_error(estimated_3d_point, pi, Mi)
    estimated_3d_point -= np.linalg.lstsq(J, reprojection_error_,
rcond=None)[0]
    current_reprojection_error = np.sum(reprojection_error_ ** 2)
```

```
   if abs(current_reprojection_error - prev_reprojection_error) <
tolerance:
   break
   prev_reprojection_error = current_reprojection_error
    iteration += 1
```

得到的測試結果:

```
Part C: Check that the difference from expected error/Jacobian
is near zero
-------------------------------------------------------------
Error Difference:  8.301300130674275e-07
Jacobian Difference:  1.8171174787084965e-08
```

```
Part D: Check that the reprojection error from nonlinear method
is lower than linear method
-----------------------------------------------------------------------------
Linear method error: 98.735423568942
Nonlinear method error: 95.59481784846037
```

3.4 Decide the Correct RT (20%)

首先使用前面創建的函式根據 essential matrix 估計初始的相對姿態 estimate_initial_RTs 獲得可能的相對旋轉和平移的初始估計。

```
estimate_initial_RTs = estimate_initial_RT(E)
```

初始化(best_RT) 為空，(max_in_front_count)為-1，這兩個變量將用於跟蹤最佳的相對姿態和最大可視點數。

接下來，根據內部參數矩陣 K（相機校準矩陣）和估計的相對姿態(RT)，計算兩個投影矩陣 M1、M2。

```
M1 = K @ np.hstack((np.eye(3), np.zeros((3, 1))))
```

進入迴圈，對每個估計的相對姿態(RT)，計算對應的投影矩陣(M2)，對於每個影像點，使用(linear_estimate_3d_point)，估計三維點(X)，如果估計的三維點(Z)的深度大於 0，則將(max_in_front_count)加 1，表示這個點在相機前方。檢查這個相對姿態下在相機前方的點的數量，

如果大於之前最大的數量，則更新(max_in_front_count)和(best_RT)，

最後，返回具有最大可視點數的相對姿態(best_RT)。

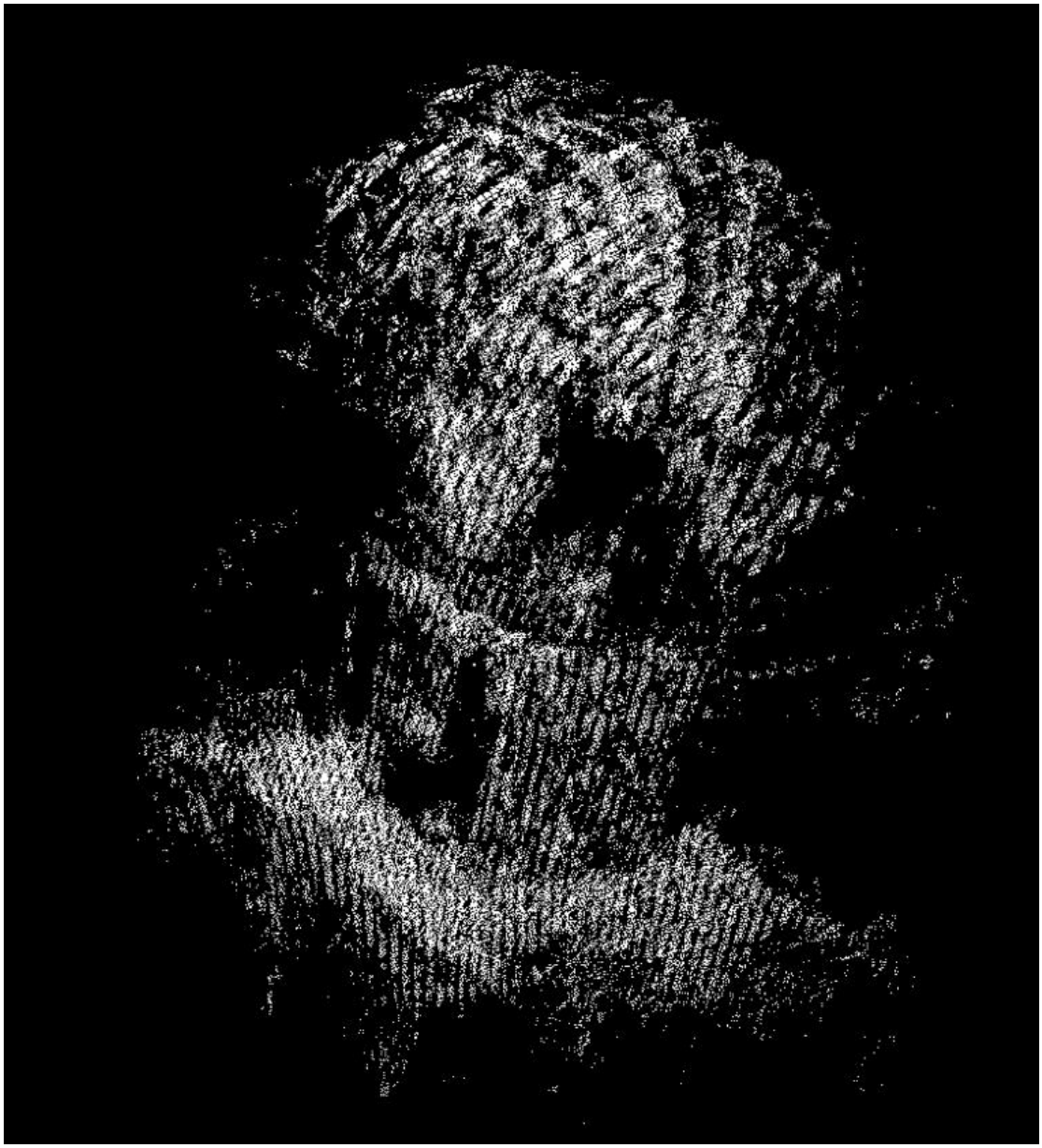
```
best_RT = None
max_in_front_count = -1
for RT in estimate_initial_RTs:
    M2 = K @ RT
    in_front_count = 0
    for i in range(image_points.shape[0]):
        X = linear_estimate_3d_point(image_points[i], np.array([M1,
M2]))
    if X[2] > 0:
      in_front_count += 1
    if in_front_count > max_in_front_count:
       max_in_front_count = in_front_count
       best_RT = RT
```

測試結果:

```
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:
 [[ 0.98305251 -0.11787055 -0.14040758  0.99941228]
 [-0.11925737 -0.99286228 -0.00147453 -0.00886961]
 [-0.13923158  0.01819418 -0.99009269  0.03311219]]
```

The final estimated 3D point cloud:



我覺得 Structure from Motion（SfM）技術可應用於許多領域:

三維重建和建模，用於重建真實世界場景的三維模型，包括建築物、自然景觀、考古遺址等。這在城市規劃、文化遺產保存、虛擬旅遊等方面非常有用。或是機器視覺和物體辨識，用於增強機器視覺系統的理解能力。它可以幫助機器識別和跟踪物體，並理解它們在三維空間

中的位置和運動。或是用於自駕車中，以估計車輛周圍環境的三維結構，從而實現智能導航和障礙物避免。總之 Structure from Motion（SfM）有助於從影像中獲得三維信息，提供許多有用的和實用的技術。