Computer Vision

Homework 2: Structure from Motion (SfM)

110062573 陳筱薇

3.1 Camera Pose from Essential Matrix

First, I apply the SVD method to the essential matrix to get the U and VT. And then, we compute the rotation matrix R using Q. We can get two rotation matrices. Then we use U to calculate the translation matrix T and get two translation matrices. Finally, we have four combinations of RT.

```
def estimate_initial_RT(E):
    W = np.array([[0, -1, 0], [1, 0, 0], [0, 0, 1]])
    U, sigma, VT = np.linalg.svd(E)
    Q_1 = U.dot(W).dot(VT)
    Q_2 = U.dot(W.T).dot(VT)
    R_1 = np.dot(np.linalg.det(Q_1),Q_1)
    R_2 = np.dot(np.linalg.det(Q_2),Q_2)
    T_1 = U[:, 2]
    T_2 = U[:, 2]*(-1)
    RT = np.zeros((4, 3, 4))
    RT[0, :, :] = np.hstack((R_1, np.expand_dims(T_1.T, axis=1)))
    RT[1, :, :] = np.hstack((R_1, np.expand_dims(T_2.T, axis=1)))
    RT[2, :, :] = np.hstack((R_2, np.expand_dims(T_1.T, axis=1)))
    RT[3, :, :] = np.hstack((R_2, np.expand_dims(T_2.T, axis=1)))

    return RT
```

```
--------------------------------------------------------------------------------
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

We use the formula $\begin{bmatrix} v_1 M_1^3 - M_1^2 \\ M_1^1 - u_1 M_1^3 \\ \vdots \\ v_n M_n^3 - M_n^2 \\ M_n^1 - u_n M_n^3 \end{bmatrix} \cdot P = 0.$ to get P, so we can use the SVD method on this matrix $\begin{bmatrix} v_1 M_1^3 - M_1^2 \\ M_1^1 - u_1 M_1^3 \\ \vdots \\ v_n M_n^3 - M_n^2 \\ M_n^1 - u_n M_n^3 \end{bmatrix}$ to get the position of P. And the final row of VT will be the answer to P.

```
def linear_estimate_3d_point(image_points, camera_matrices):
    linear_matrix = np.zeros((image_points.shape[0]*2, camera_matrices.shape[-1]))
    for i in range(image_points.shape[0]) :
        pi = image_points[i]
        Mi = camera_matrices[i]
        u = pi[0]
        v = pi[1]
        Mi_1 = Mi[0, :]
        Mi_2 = Mi[1, :]
        Mi_3 = Mi[2, :]
        linear_matrix[i*2, :] = v*Mi_3 - Mi_2
        linear_matrix[i*2+1, :] = Mi_1 - u*Mi_3

    U, sigma, VT = np.linalg.svd(linear_matrix)
    P = VT[-1, :]
    P = (P / P[-1])[:-1]

    return P
```

```
--------------------------------------------------------------------------------
Part B: Check that the difference from expected point
--------------------------------------------------------------------------------
Difference:  0.0029243053036712707
--------------------------------------------------------------------------------
```

3.3 Non-Linear 3D Points Estimation

We can calculate y1, y2, y3 by Mi and the 3D location of a point and use the formula $p'_i = \begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{y_3}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$. to get the projected image coordinate of P. Use pi' to minus the ground-truth projected image coordinate pi, then we can get the reprojection error.

```
def reprojection_error(point_3d, image_points, camera_matrices):
    P = np.zeros(point_3d.shape[0]+1)
    P[:-1] = point_3d
    error = np.zeros(image_points.shape[0]*2)
    for i in range(image_points.shape[0]) :
        Mi_1 = camera_matrices[i][0, :]
        Mi_2 = camera_matrices[i][1, :]
        Mi_3 = camera_matrices[i][2, :]
        X = P[0]
        Y = P[1]
        Z = P[2]
        y1 = X * Mi_1[0] + Y * Mi_1[1] + Z * Mi_1[2] + Mi_1[3]
        y2 = X * Mi_2[0] + Y * Mi_2[1] + Z * Mi_2[2] + Mi_2[3]
        y3 = X * Mi_3[0] + Y * Mi_3[1] + Z * Mi_3[2] + Mi_3[3]
        pi = np.array([y1/y3, y2/y3])
        error[i*2] = pi[0] - image_points[i][0]
        error[i*2+1] = pi[1] - image_points[i][1]

    return error
```

Before we use the Jacobian matrix for calculating Gauss-Newton optimization, we have to get the Jacobian matrix first. So we use the Jacobian formula to get this.

```
def jacobian(point_3d, camera_matrices):
    # jacobian matrix
    J = np.zeros((camera_matrices.shape[0]*2, point_3d.shape[0]))
    P = np.zeros(point_3d.shape[0]+1)
    P[:-1] = point_3d
    for i in range(camera_matrices.shape[0]) :
        Mi_1 = camera_matrices[i][0, :]
        Mi_2 = camera_matrices[i][1, :]
        Mi_3 = camera_matrices[i][2, :]
        X = P[0]
        Y = P[1]
        Z = P[2]
        y1 = X * Mi_1[0] + Y * Mi_1[1] + Z * Mi_1[2] + Mi_1[3]
        y2 = X * Mi_2[0] + Y * Mi_2[1] + Z * Mi_2[2] + Mi_2[3]
        y3 = X * Mi_3[0] + Y * Mi_3[1] + Z * Mi_3[2] + Mi_3[3]

        # multiply by y3 first, then divide by y3**2, to avoid wrong answer
        J[i*2, 0] = (Mi_1[0]*y3 - Mi_3[0]*y1)/y3**2
        J[i*2, 1] = (Mi_1[1]*y3 - Mi_3[1]*y1)/y3**2
        J[i*2, 2] = (Mi_1[2]*y3 - Mi_3[2]*y1)/y3**2
        J[i*2+1, 0] = (Mi_2[0]*y3 - Mi_3[0]*y2)/y3**2
        J[i*2+1, 1] = (Mi_2[1]*y3 - Mi_3[1]*y2)/y3**2
        J[i*2+1, 2] = (Mi_2[2]*y3 - Mi_3[2]*y2)/y3**2

    return J
```

```
--------------------------------------------------------------------------------
Part C: Check that the difference from expected error/Jacobian
is near zero
--------------------------------------------------------------------------------
Error Difference:  8.301300130674275e-07
Jacobian Difference:  1.817115702351657e-08
--------------------------------------------------------------------------------
```

Because the linear 3D points estimation method may have a larger reprojection error, we use Gauss-Newton optimization to do ten iterations to decrease the error of 3D points.

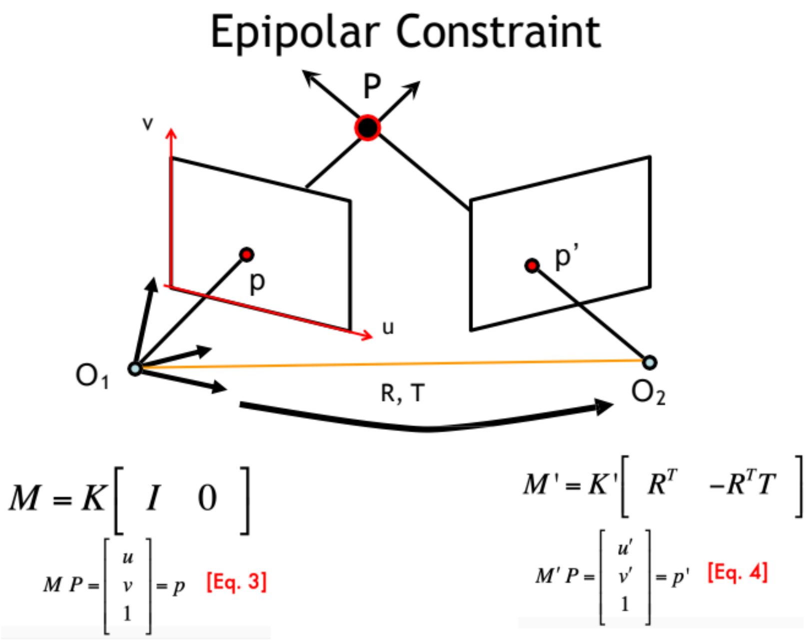
```python
def nonlinear_estimate_3d_point(image_points, camera_matrices):
    point_3d = linear_estimate_3d_point(image_points, camera_matrices)
    for i in range(10) :
        error = reprojection_error(point_3d, image_points, camera_matrices)
        J = jacobian(point_3d, camera_matrices)
        point_3d = point_3d - np.linalg.inv(J.T @ J) @ J.T @ error

    return point_3d
```

```
--------------------------------------------------------------------------------
Part D: Check that the reprojection error from nonlinear method
is lower than linear method
--------------------------------------------------------------------------------
Linear method error: 98.73542356894177
Nonlinear method error: 95.59481784846034
--------------------------------------------------------------------------------
```

3.4 Decide the Correct RT

In the end, we must use the estimated 3D points to check which RT is correct, so we estimate the 3D point by the non-linear estimate method, then we convert the coordinate of this 3D point from camera1 space to camera2 space. Then, we can check if the current RT can get most 3D points having positive z-coordinate at camera1 and camera2 space. This RT will be the correct RT.



```python
def estimate_RT_from_E(E, image_points, K):
    estimated_RT = estimate_initial_RT(E)
    max_count = -1
    correct_RT = None
    for i in range(estimated_RT.shape[0]) :
        count = 0
        RT = estimated_RT[i]
        camera_matrices = np.zeros((2, 3, 4))
        camera_matrices[0, :, :] = K.dot(np.hstack((np.eye(3), np.zeros((3,1)))))

        R = RT[:,:3]
        T = RT[:,3:]
        new_RT = np.concatenate((R.T, -R.T.dot(T)), axis=1)
        camera_matrices[1, :, :] = K.dot(new_RT)

        for j in range(image_points.shape[0]) :
            point_3d_camera1_coor = nonlinear_estimate_3d_point(image_points[j], camera_matrices)

            # convert camera1 coordinate to camera2 coordinate
            point_3d_tmp = np.ones((4,1))
            point_3d_tmp[0:3, :] = point_3d_camera1_coor.reshape((3,1))
            point_3d_camera2_coor = new_RT.dot(point_3d_tmp)

            if point_3d_camera1_coor[2] > 0 and point_3d_camera2_coor[2] > 0 :
                count += 1

        if count > max_count :
            max_count = count
            correct_RT = RT

    return correct_RT
```

```
--------------------------------------------------------------------------------
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]]
--------------------------------------------------------------------------------
```

We use visualize.py to show the results of camera motions to structure.