16-720 Computer Vision: Homework 4 (Fall 2022)

3D Reconstruction

Haejoon Lee

Q1.1 (5 points) Suppose two cameras fixate on a point $\mathbf{x}$ (see Figure 1) in space such that their principal axes intersect at that point. Show that if the image coordinates are normalized so that the coordinate origin $(0, 0)$ coincides with the principal point, the $\mathbf{F}_{33}$ element of the fundamental matrix is zero.

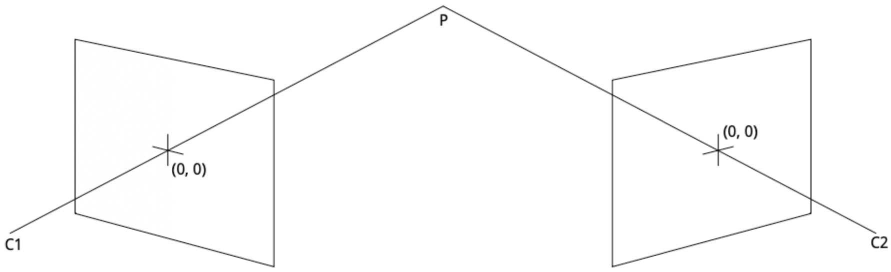


Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Epipolar constraint:

$$x_2 F x_1 = 0$$

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} F \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_{13} \\ F_{23} \\ F_{33} \end{bmatrix} = 0$$

$$F_{33} = 0$$

Q1.2 **(5 points)** Consider the case of two cameras viewing an object such that the second camera differs from the first by a *pure translation* that is parallel to the x-axis. Show that the epipolar lines in the two cameras are also parallel to the x-axis. Backup your argument with relevant equations. You may assume both cameras have the same intrinsics.

Pure translation -> R = I

Epipolar constraints:

$$x_2^T \hat{T} x_1 = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = -t_1 y_2 + t_1 y_1 = 0$$

$$y_2 = y_1$$

Thus, if we fix single y, then the epipolar line is parallel to the x-axis.

Q1.3 (5 points) Suppose we have an inertial sensor which gives us the accurate positions ($\mathbf{R}_i$ and $\mathbf{t}_i$, the rotation matrix and translation vector) of the robot at time i. What will be the effective rotation ($\mathbf{R}_{rel}$) and translation ($\mathbf{t}_{rel}$) between two frames at different time stamps? Suppose the camera intrinsics ($\mathbf{K}$) are known, express the essential matrix ($\mathbf{E}$) and the fundamental matrix ($\mathbf{F}$) in terms of $\mathbf{K}$, $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$.

$$R_{i+1} = R_{rel} R_i$$

$$R_{rel} = R_{i+1} R_i^{-1}$$

$$t_{rel} = t_{i+1} - t_i$$

$$E = t_{rel} \times R_{rel}$$

$$F = K^{-T} t_{rel} \times R_{rel} K^{-1}$$

Q1.4 (10 points) Suppose that a camera views an object and its reflection in a plane mirror. Show that this situation is equivalent to having two images of the object which are related by a skew-symmetric fundamental matrix. You may assume that the object is flat, meaning that all points on the object are of equal distance to the mirror.

From the matrix D, we can assume the x-axis of the world coordinate is perpendicular to the mirror.

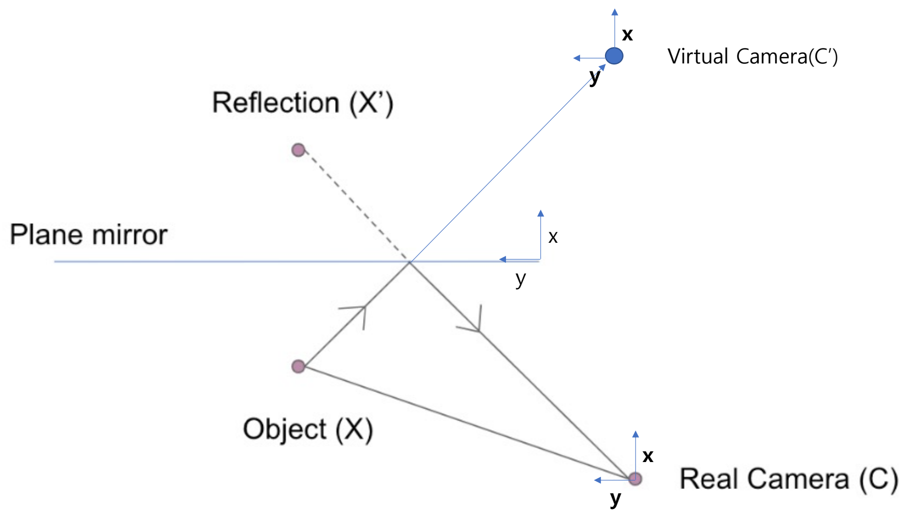
$$\lambda x = KX$$

$$\lambda x' = KX' = K\tilde{D}X = K'X, \tilde{D} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, it is equivalent to having another camera whose matrix is K'.

Let's assume that the camera coordinate's axis is parallel to word coordinates.

Then, the virtual camera is only at the translated position as below:



Epipolar constraints:

$$x^T K^{-T} \hat{T} (K')^{-1} x' = x^T K^{-T} \hat{T} D^{-1} K^{-1} x'$$

If $\hat{T}D^{-1}$ is skew-symmetric, then the fundamental matrix will be so.

$$\hat{T}D^{-1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix}$$

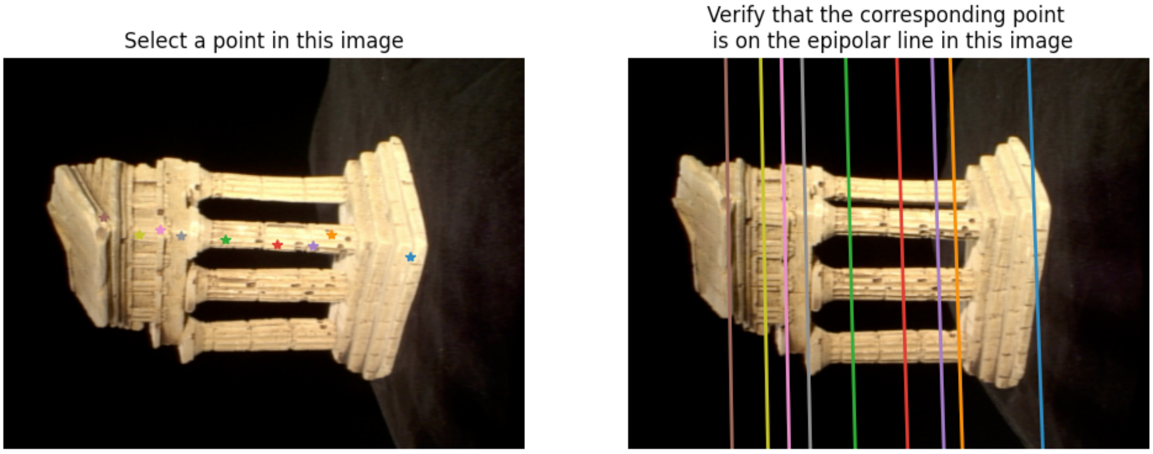
Thus, the fundamental matrix is skew-symmetric.

Q2.1 **(10 points)** Finish the function `eightpoint` in `q2_1_eightpoint.py`. Make sure you follow the signature for this portion of the assignment:

```
F = eightpoint(pts1, pts2, M)
```

where `pts1` and `pts2` are $N \times 2$ matrices corresponding to the (x, y) coordinates of the N points in the first and second image respectively. `M` is a scale parameter.

```
[[-2.19299589e-07  2.95926454e-05 -2.51886347e-01]
 [ 1.28064550e-05 -6.64493729e-07  2.63771743e-03]
 [ 2.42229089e-01 -6.82585560e-03  1.00000000e+00]]
```



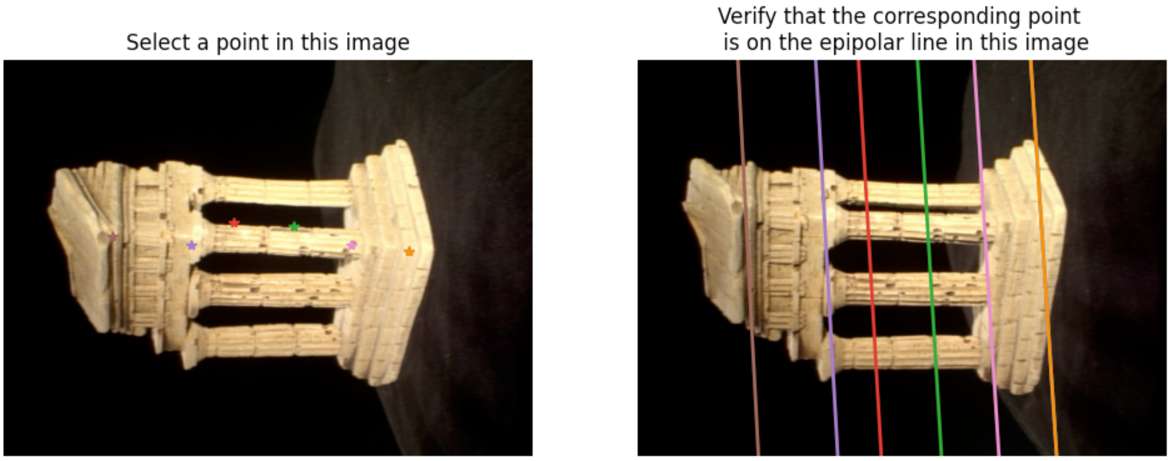

```python
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation
    N = pts1.shape[0]

    # Normalization
    pts1, pts2 = pts1/float(M), pts2/float(M)

    xcoords1, ycoords1 = pts1[:, 0], pts1[:, 1]
    xcoords2, ycoords2 = pts2[:, 0], pts2[:, 1]

    # A Matix
    cul0 = xcoords2 * xcoords1
    cul1 = xcoords2 * ycoords1
    cul2 = xcoords2
    cul3 = ycoords2 * xcoords1
    cul4 = ycoords2 * ycoords1
    cul5 = ycoords2
    cul6 = xcoords1
    cul7 = ycoords1
    cul8 = np.ones((N,), dtype=np.float32)

    A = np.stack((cul0, cul1, cul2, cul3, cul4, cul5, cul6, cul7, cul8), axis=1)
```

```python
# solve a raw f
_, _, Vt = np.linalg.svd(A)

F_vec = Vt[-1, :] #(9,)
F_raw = F_vec.reshape(3, 3)

#Refine F
F_norm = _singularize(F_raw)
F_norm = refineF(F_norm, pts1, pts2) # ?

# Unscale
T = np.zeros((3, 3), dtype=np.float32)
T[0, 0] = T[1, 1] = 1.0 / M
T[2, 2] = 1.0

F_final = T.transpose() @ F_norm @ T

return F_final
```

Q2.2 (15 points) Finish the function `sevenpoint` in `q2_1_sevenpoint.py`. Make sure you follow the signature for this portion of the assignment:

```
Farray = sevenpoint(pts1, pts2, M)
```

```
[[ 8.10457592e-07  8.90919532e-06 -2.01028427e-01]
 [ 2.63329756e-05 -6.00542611e-07  6.97429513e-04]
 [ 1.92182052e-01 -4.20123587e-03  1.00000000e+00]]
```




```python
def sevenpoint(pts1, pts2, M):

    Farray = []
    # ----- TODO -----
    # YOUR CODE HERE
    N = pts1.shape[0]

    # Normalization
    pts1, pts2 = pts1/float(M), pts2/float(M)

    xcoords1, ycoords1 = pts1[:, 0], pts1[:, 1]
    xcoords2, ycoords2 = pts2[:, 0], pts2[:, 1]

    # A Matix
    cul0 = xcoords2 * xcoords1
    cul1 = xcoords2 * ycoords1
    cul2 = xcoords2
    cul3 = ycoords2 * xcoords1
    cul4 = ycoords2 * ycoords1
    cul5 = ycoords2
    cul6 = xcoords1
    cul7 = ycoords1
    cul8 = np.ones((N,), dtype=np.float32)

    A = np.stack((cul0, cul1, cul2, cul3, cul4, cul5, cul6, cul7, cul8), axis=1)

    # Get F1 and F2
```

```python
_, _, Vt = np.linalg.svd(A)
F1_vec, F2_vec = Vt[-1, :], Vt[-2, :] #(9,)
F1, F2 = F1_vec.reshape(3, 3), F2_vec.reshape(3, 3)

#Find the coefficients for F1 and F2 spanning the null space
a, b = F1-F2, F2

funct = lambda x: np.linalg.det(x*a + b)

c0 = funct(0)
c1 = (2.0/3)*(funct(1)-funct(-1)) - (1.0/12)*(funct(2)-funct(-2))
c3 = (1.0/12)*(funct(2) - funct(-2)) - (1.0/6)*(funct(1)-funct(-1))
c2 = funct(1) - c0 - c1 - c3

#Solve the polynomial
roots = poly.polyroots([c0, c1, c2, c3])

# Unscale F
T = np.zeros((3, 3), dtype=np.float32)
T[0, 0] = T[1, 1] = 1.0 / M
T[2, 2] = 1.0

for root in roots:
    F_norm = root*a + b

    F_norm = _singularize(F_norm)
    # F_norm = refineF(F_norm, pts1, pts2)

    F_final = T.transpose() @ F_norm @ T
    Farray.append(F_final)

return Farray
```

Q3.1 (5 points) Complete the function `essentialMatrix` in `q3_1_essential_matrix.py` to compute the essential matrix $\mathbf{E}$ given $\mathbf{F}$, $\mathbf{K}_1$ and $\mathbf{K}_2$ with the signature:

```
E = essentialMatrix(F, K1, K2)
```

Output: Save your estimated $\mathbf{E}$ using $\mathbf{F}$ from the eight-point algorithm to `q3_1.npz`. **Please include the code snippet of essentialMatrix function in your write-up.**

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    E = K2.transpose() @ F @ K1
    return E
```

Q3.2 (10 points) Using the above, complete the function `triangulate` in `q3_2_triangulate.py` to triangulate a set of 2D coordinates in the image to a set of 3D points with the signature:

```
[w, err] = triangulate(C1, pts1, C2, pts2)
```

where `pts1` and `pts2` are the $N \times 2$ matrices with the 2D image coordinates and `w` is an $N \times 3$ matrix with the corresponding 3D points per row. `C1` and `C2` are the 3×4 camera matrices. Remember that you will need to multiply the given intrinsics matrices with your solution for the canonical camera matrices to obtain the final camera matrices. Various methods exist for triangulation - probably the most familiar for you is based on least squares (see Szeliski Chapter 7 if you want to learn about other methods):

Assume we have two points x and x'

$$\lambda \vec{x} = CX, x \times CX = 0$$

$$\lambda \vec{x'} = C'X, x' \times C'X = 0$$

$$\begin{bmatrix} C_1 - xC_3 \\ C_2 - yC_3 \\ C'_1 - x'C'_3 \\ C'_2 - y'C'_3 \end{bmatrix} X = 0, C = \begin{bmatrix} -C_1- \\ -C_2- \\ -C_3- \end{bmatrix}$$

Q3.3 (10 points) Complete the function `findM2` in `q3_2_triangulate.py` to obtain the correct `M2` from `M2s` by testing the four solutions through triangulations. Use the correspondences from `data/some_corresp.npz`.

Output: Save the correct `M2`, the corresponding `C2`, and 3D points `P` to `q3_3.npz`. **Please include the code snippet of triangulate and findM2 function in your write-up.**

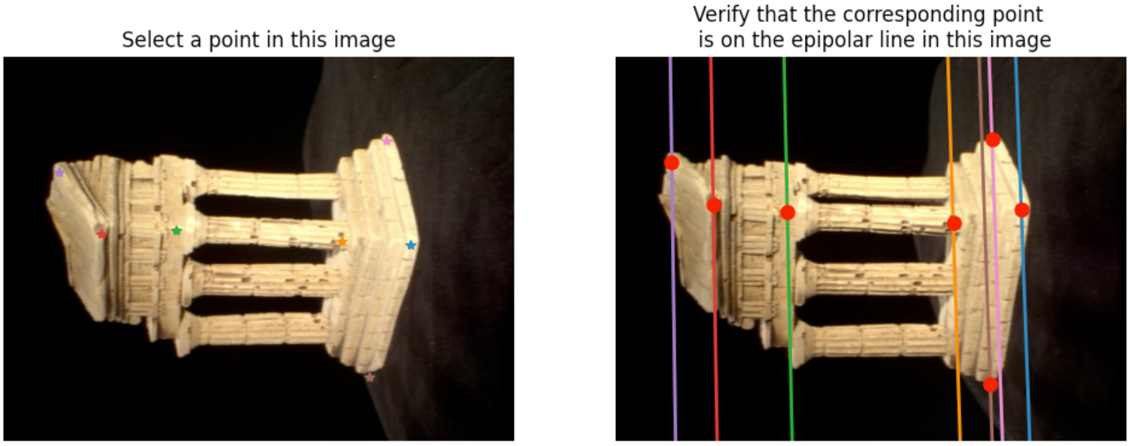
```
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation
    coord_word_list = []
    err_reproject = 0
    for i in range(pts1.shape[0]):
        x1, y1 = pts1[i, :]
        x2, y2 = pts2[i, :]

        # A matrix
        A0 = C1[0, :] - x1*C1[2, :]
        A1 = C1[1, :] - y1*C1[2, :]
        A2 = C2[0, :] - x2*C2[2, :]
        A3 = C2[1, :] - y2*C2[2, :]
        A = np.stack((A0, A1, A2, A3), axis=0)

        #Get world coordinates through SVD
        _, _, Vt = np.linalg.svd(A)
        coord_word = Vt[-1, :] #(4,)
        coord_word = coord_word[0:3] / coord_word[3] #(3,)
        coord_word_list.append(coord_word)

        #Reprojection
        coord_word_homo = np.zeros((4, 1), dtype=np.float32)
        coord_word_homo[0:3, 0] = coord_word
        coord_word_homo[3, 0] = 1
        coord_cam1_rep = C1 @ coord_word_homo
        coord_cam2_rep = C2 @ coord_word_homo

        #Calculate reprojection error
        x1_cam1_rep, y1_cam1_rep = coord_cam1_rep[0:2, 0] / coord_cam1_rep[2, 0]
        x2_cam2_rep, y2_cam2_rep = coord_cam2_rep[0:2, 0] / coord_cam2_rep[2, 0]

        err_reproject += (x1_cam1_rep-x1)**2 + (y1_cam1_rep-y1)**2 + (x2_cam2_rep-x2)**2 + (y2_cam2_rep-y2)**2

    P = np.stack(coord_word_list, axis=0)

    return P, err_reproject

def findM2(F, pts1, pts2, intrinsics, filename = 'q3_3.npz'):
    '''
    Q2.2: Function to find the camera2's projective matrix given correspondences
        Input:  F, the pre-computed fundamental matrix
```

```
            pts1, the Nx2 matrix with the 2D image coordinates per row
            pts2, the Nx2 matrix with the 2D image coordinates per row
            intrinsics, the intrinsics of the cameras, load from the .npz file
            filename, the filename to store results
        Output: [M2, C2, P] the computed M2 (3x4) camera projective matrix, C2 (3x4) K2 * M2, and the 3D points P
(Nx3)

    ***
    Hints:
    (1) Loop through the 'M2s' and use triangulate to calculate the 3D points and projection error. Keep track
        of the projection error through best_error and retain the best one.
    (2) Remember to take a look at camera2 to see how to correctly reterive the M2 matrix from 'M2s'.

    '''

    K1, K2 = intrinsics['K1'], intrinsics['K2']
    E = essentialMatrix(F, K1, K2)

    M2s = camera2(E)

    # Assume camera coordinates = word coordinates
    M1 = np.zeros((3, 4), dtype=np.float32)
    M1[0,0] = 1
    M1[1,1] = 1
    M1[2,2] = 1
    C1 = K1 @ M1

    # Find C2, P for each M2
    P_list = []
    C2_list = []
    err_list = []
    for i in range(M2s.shape[2]):
        M2 = M2s[:, :, i]
        C2 = K2 @ M2
        C2_list.append(C2)

        P, err_rep = triangulate(C1, pts1, C2, pts2)

        P_list.append(P)
        err_list.append(err_rep)

    #Remain best M2
    min_idx = np.argmin(np.abs(np.array(err_list)))
    # print(min_idx)
    M2 = M2s[:, :, min_idx]
    P = P_list[min_idx]
    C2 = C2_list[min_idx]
    return M2, C2, P
```

Q4.1 (15 points) In `q4_1_epipolar_correspondence.py` finish the function `epipolarCorrespondence` with the signature:

```
[x2, y2] = epipolarCorrespondence(im1, im2, F, x1, y1)
```

This function takes in the x and y coordinates of a pixel on `im1` and your fundamental matrix **F**, and returns the coordinates of the pixel on `im2` which correspond to the input point. The match is obtained by computing the similarity of a small window around the (x_1, y_1) coordinates in `im1` to various windows around possible matches in the `im2` and returning the closest.

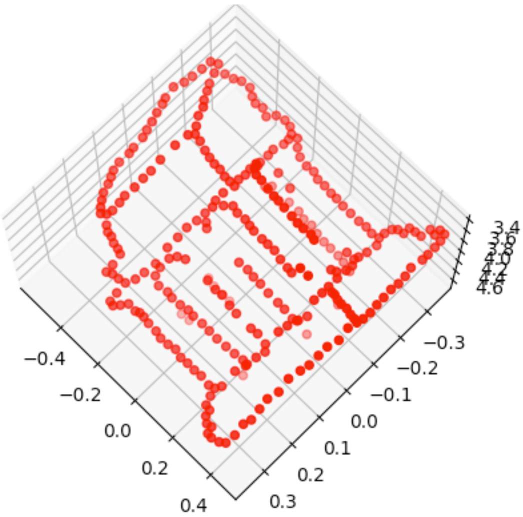


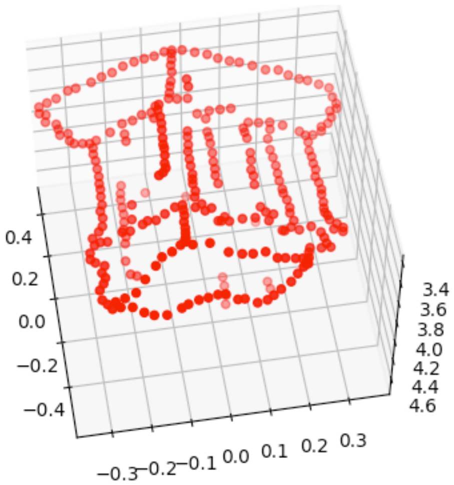

```python
def KernelResponse(im, x, y, kxs, kys):
    H, W = im.shape[0:2]
    xs, ys = kxs + x, kys + y
    xs, ys = np.clip(xs, 0, W-1).astype(np.int32), np.clip(ys, 0, H-1).astype(np.int32)
    response = im[ys, xs, :]
    return response

def epipolarCorrespondence(im1, im2, F, x1, y1):

    coord_img1_homo = np.array([[x1], [y1], [1]], dtype=np.float32)
    # Calculate epipolar line on im2
    epipolarline_img2 = F @ coord_img1_homo

    # Set search boundary
    r = 100
    ll = np.array([1, 0, -(x1-r)], dtype=np.float32).reshape(-1, 1)
    lt = np.array([0, 1, -(y1-r)], dtype=np.float32).reshape(-1, 1)
    lr = np.array([1, 0, -(x1+r)], dtype=np.float32).reshape(-1, 1)
    lb = np.array([0, 1, -(y1+r)], dtype=np.float32).reshape(-1, 1)
    lines_boundary = [ll, lt, lr, lb]

    # Get intersected points between search boundary and epipolar line
    points_intersect = [np.cross(l.reshape(-1), epipolarline_img2.reshape(-1)).reshape(-1, 1) for l
in lines_boundary]
```

```python
points_end = []
for points in points_intersect:
    if np.abs(points[2, 0]) > 0.0000001:
        points_homo = points / points[2, 0]
        dist_max = np.max(np.abs(points_homo[0:2, :]-coord_img1_homo[0:2, :]))
        if dist_max < r*(1.1):
            points_end.append(points_homo)

search_begin = None
search_end = None
if len(points_end) > 2:
    point0_end = points_end[0]
    for i in range(1, len(points_end)):
        coord_img1_homo = points_end[i]
        if np.min(np.abs(point0_end[0:2, :]-coord_img1_homo[0:2, :])) > r*0.1:
            search_begin = point0_end[0:2, :]
            search_end = coord_img1_homo[0:2, :]
            break
else:
    search_begin = points_end[0][0:2, :]
    search_end = points_end[1][0:2, :]

# Generate Gaussian weighting kernel
kernel_r = 20
kernel_xaxis = np.arange(-kernel_r, kernel_r+1, 1.0)
kernel_yaxis = np.arange(-kernel_r, kernel_r+1, 1.0)
kernel_xaxis, kernel_yaxis = np.meshgrid(kernel_xaxis, kernel_yaxis)
kernel_xaxis, kernel_yaxis = kernel_xaxis.reshape(-1), kernel_yaxis.reshape(-1)

STD = kernel_r / 2.0
kernel_window = np.exp( -( (kernel_xaxis**2 + kernel_yaxis**2)/(2*STD**2) ) ) / (2*np.pi*STD**2)

response_img1 = KernelResponse(im1, x1, y1, kernel_xaxis, kernel_yaxis)

#Calculate distances by compare kernel response and get the best matching point on im2
dist_min = np.Inf
num_steps = int(np.sum((search_begin-search_end)**2)**0.5)
x_step = (search_end[0, 0] - search_begin[0, 0]) / num_steps
y_step = (search_end[1, 0] - search_begin[1, 0]) / num_steps
x2, y2 = search_begin[0, 0], search_begin[1, 0]
x2_best, y2_best = -1, -1
for i in range(num_steps):
    response_img2 = KernelResponse(im2, x2, y2, kernel_xaxis, kernel_yaxis)
    dist = np.sum((response_img2 - response_img1)**2, axis=1)**0.5 # (Nk,)
    dist = np.sum(dist*kernel_window)

    if dist < dist_min:
        dist_min = dist
        x2_best, y2_best = x2, y2
```

```
        x2, y2 = x2 + x_step, y2 + y_step

    return x2_best, y2_best
```

Q4.2 (10 points) Included in this homework is a file `data/templeCoords.npz` which contains 288 hand-selected points from `im1` saved in the variables `x1` and `y1`.






```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):

    # ----- TODO -----
    # YOUR CODE HERE
    coords_x1 = temple_pts1['x1']
    coords_y1 = temple_pts1['y1']

    coords_x2 = coords_y2 = []

    # Find x2, y2
    for i in range(coords_x1.shape[0]):
        x1, y1 = coords_x1[i, 0], coords_y1[i, 0]
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)
        coords_x2.append(x2)
```

```
    coords_y2.append(y2)

coords_x2, coords_y2 = np.array(coords_x2).reshape(-1, 1), np.array(coords_y2).reshape(-1, 1)

pts1 = np.concatenate((coords_x1, coords_y1), axis=1)
pts2 = np.concatenate((coords_x2, coords_y2), axis=1)

K1, K2 = intrinsics['K1'], intrinsics['K2']
E = essentialMatrix(F, K1, K2)
M2s = camera2(E)

# Assume camera coordinates = word coordinates
M1 = np.zeros((3, 4), dtype=np.float32)
M1[0,0] = M1[1,1] = M1[2,2] = 1

C1 = K1 @ M1

# Find C2, P for each M2
P_list = []
C2_list = []
err_list = []
for i in range(M2s.shape[2]):
    M2 = M2s[:, :, i]
    C2 = K2 @ M2
    C2_list.append(C2)

    P, err_rep = triangulate(C1, pts1, C2, pts2)
    P_list.append(P)

    print('Reprojection error of M2_%d: %f' % (i, err_rep))
    err_list.append(err_rep)

#Remain best M2
# min_idx = np.argmin(np.abs(np.array(err_list)))
min_idx = 2

print(min_idx)

M2 = M2s[:, :, min_idx]
P = P_list[min_idx]
C2 = C2_list[min_idx]

np.savez('q4_2.npz', F=F, M1=M1, M2=M2, C1=C1, C2=C2)

return P
```

Q5.1 RANSAC for Fundamental Matrix Recovery (15 points) In some real world applications, manually determining correspondences is infeasible and often there will be noisy correspondences. Fortunately, the RANSAC method seen in class can be applied to the problem of fundamental matrix estimation.

Implement the above algorithm with the signature:

```
[F, inliers] = ransacF(pts1, pts2, M, nIters, tol)
```

where `M` is defined in the same way as in Section 2 and inliers is a boolean vector of size equivalent to the number of points. Here inliers is set to true only for the points that satisfy the threshold defined for the given fundamental matrix F.

```
Distances before ransac: [2.40794690e+01 7.30360086e+00 1.06768378e+01 7.45619385e+01
 6.20462251e+01 4.73556322e-06 4.32861983e+01 5.69080641e+01
 6.48121585e-01 7.30051572e+01 3.85064663e+01 2.52112320e+00
 7.23744664e+01 8.24600439e+01 1.21476969e+02 8.23397948e+00
 6.15373300e+01 7.91125512e+01 1.28650089e+02 1.83947713e+01
 3.45370802e+01 8.26155032e+01 1.12156900e+00 3.46556179e+01
 7.97900053e+01 1.09979778e+02 3.29832793e+01 5.28317277e+01
 7.68860971e+00 1.31577690e-06 1.87047133e+02 4.78325484e+01
 6.71957948e+01 7.02706037e+01 6.06100302e+01 1.91826684e+02
 6.22174762e+00 4.90792901e+01 2.13034666e+02 5.39614156e+01
 3.48943981e+01 2.78215412e+01 1.73252358e-05 3.60447823e+01
 4.76535865e+01 2.36279670e+02 3.77051250e+01 8.42296863e+00
 4.28701073e-01 2.24721681e+01 5.08269518e+01 6.10556652e+01
 1.11490624e+02 1.89963579e+01 7.40033414e+00 8.30687800e+01
 6.37467305e+01 9.01010558e+01 5.32445636e-07 9.33243544e+00
 8.73020174e+01 4.11824721e+01 1.66316909e+01 5.30793878e+01
 1.15668453e+01 1.08436275e-06 1.09873729e+01 3.32369866e+01
 1.37976995e+02 3.97487906e+01 2.69294853e+02 9.93426841e+01
 6.40671308e+00 3.71759148e+01 2.10764098e+01 4.90850409e+01
 4.92859474e+00 1.27910828e+01 1.07601586e+02 8.69108068e+00
 1.95101489e+01 4.61957196e+01 2.22676199e+01 3.61906919e-07
 1.05245555e+02 9.11986694e+01 3.87880779e+01 8.75135875e+01
 1.79648155e+00 2.68464803e+01 6.73190606e+01 7.90761570e+01
 6.93228108e+01 3.24369873e+01 2.36140415e+00 1.19727631e+01
 2.19842940e+01 1.49988182e+01 2.96814837e+01 3.64106791e+01
 2.45880058e+02 4.81651911e+01 1.24778919e+02 1.11042417e+02
 7.56108061e-01 1.70670371e+00 1.01765199e-05 6.56138721e+01
 5.02230463e+01 6.39300990e+01 3.50877233e+01 1.37329520e+02
 1.14534895e+02 3.06848206e+01 1.60359304e+01 1.30198784e+02
 7.21742692e+01 2.13756582e+01 3.37768791e+01 7.75068800e+01
 7.57586673e+01 7.19655456e+01 3.41903015e+01 7.04776620e+01
 9.37062106e+01 3.90230632e+01 1.02906695e+02 1.65117418e+01
 5.19663860e+01 5.13339756e+01 1.04957928e+02 5.44073101e+01
 7.92333641e+01 1.76931473e+02 6.77476528e-01 6.30371113e+00
 2.69103346e+01 9.52290828e+01 2.84083510e+01 2.46094194e+01]
```

```
Distances after ransac: [1.06387192e+00 3.07283045e-01 2.23575719e-01 3.25469154e-01
 3.23695059e-01 4.79948901e-01 2.83416966e+02 7.84054228e-01
 1.74149009e-01 7.85246405e-01 4.52643495e-01 4.71941390e-01
 6.18444461e-01 2.88834144e-01 4.22667673e-01 3.19251097e-01
 6.21760464e-01 2.86540708e+01 1.23680178e-02 1.37748472e-01
 2.29301911e-01 1.56686007e-01 1.45195837e-01 4.53917244e+01
 6.58272077e-01 7.67051929e-02 4.50564373e-01 2.40527216e-01
 1.16006076e-02 8.78598691e+01 6.04084260e-01 6.20569734e-01
 5.78771101e-01 3.40082195e-01 6.24556677e+01 1.19362848e+00
 2.96274090e-01 6.26724511e-01 3.56233600e+02 2.46601541e-01
 6.52009689e+01 4.32708974e-01 4.55301361e+01 1.61121056e-01
 6.02372660e-02 4.50578865e+02 2.14717113e-01 3.32406340e-01
 1.62424009e+00 4.05604016e+02 1.90721670e+02 8.74049754e-01
 1.74921763e-01 5.88040275e-02 3.10199973e-01 9.62643288e-01
 2.99272556e-01 9.47714237e+01 6.01348363e-02 1.87984134e-01
 2.82758775e-01 1.28303721e-01 3.07834734e-02 2.43071379e-01
 1.05173568e-01 3.18278003e-01 2.99377402e-01 4.35430841e-01
 1.60670241e+02 1.58978629e-01 2.33262816e+01 1.28195206e+02
 3.39885840e-01 5.54446293e+01 1.41252786e-02 1.27344749e+01
 3.60012654e-01 7.02585227e-01 5.82791180e-01 7.27423499e-01
 6.88827457e-01 1.87549117e+02 7.10566514e-01 3.13145382e+01
 1.80042121e-01 2.14112960e+02 2.15672930e-01 5.02566059e-01
 4.21001943e-01 2.37118675e+02 7.66527764e-01 4.03558578e-01
 1.20178697e+00 2.13669847e-01 3.29940836e-01 6.43048392e-01
 1.01391040e-01 5.54784275e-03 4.93505805e-01 2.46251475e-01
 4.11483757e+01 6.28300775e-01 1.65930263e+02 9.19100294e+01
 1.68878698e-01 1.93841237e+02 8.55137031e+01 2.03489324e-01
 5.02033870e-01 5.64468394e-01 1.63439455e-01 5.94819149e+01
 6.46533863e-01 3.76865977e-01 3.88277012e-01 5.58172914e-01
 6.49126430e-01 8.00266246e-02 2.82445480e-01 6.43132361e-01
 5.02969634e-02 1.82109324e-01 4.09680460e-01 6.49838643e-02
 6.43474523e-01 5.40004533e-01 1.03744483e-01 2.12343026e-01
 2.62099587e-01 1.72161069e-01 1.91571821e-01 3.09956114e-01
 3.48605787e-01 4.04735266e-01 5.47719079e+01 4.94593070e-01
 1.67039157e+02 8.34390223e-01 1.24342089e+02 1.58188814e-01]
```

Using sevenpoint algorithm, the F was computed and then multiplied with pts2 to get epipolar line on im2. Then, the distance between the line and pts1 was computed, and inliers were selected with smaller distance than ‘tol’. Using the inliers, F was computed again. Above process went interatively. By increasing ‘nIters’, we can go more iteration and get more accurate F for the noisy points with increased computation time. By increasing tol, we can include more candidates for inliers.

```python
def dist_to_epipolarline(pts1_homo_t, pts2_homo_t, F):
    epipolar_lines = (pts2_homo_t @ F)
    # Calculate the distance between pts1 and epipolar line of pt2 on im1
    dist = np.abs(np.sum(epipolar_lines * pts1_homo_t, axis=1)) / (epipolar_lines[:, 0]**2 +
epipolar_lines[:, 1]**2)**0.5

    return np.abs(dist)

def ransacF(pts1, pts2, M, nIters=1000, tol=2.0):
    # Replace pass by your implementation

    N = pts1.shape[0]
    pts1_homo_t = np.concatenate((pts1, np.ones((N, 1))), axis=1)
```

```python
pts2_homo_t = np.concatenate((pts2, np.ones((N, 1))), axis=1)

max_inlier_num = 0
best_inlier_idx = None
for i in range(nIters):
    idx_selected = np.random.choice(N, 7, replace=False)
    pts1_selected = pts1[idx_selected, :]
    pts2_selected = pts2[idx_selected, :]

    #Try sevenpoint algorithm
    Farray = sevenpoint(pts1_selected, pts2_selected, M)
    for F in Farray:
        dist_epipolarline2_pts1 = dist_to_epipolarline(pts1_homo_t, pts2_homo_t, F)
        inlier_idx = np.where(np.abs(dist_epipolarline2_pts1) < tol)[0]
        if inlier_idx.shape[0] > max_inlier_num:
            max_inlier_num = inlier_idx.shape[0]
            best_inlier_idx = inlier_idx
            print('Inlier number: {}'.format(max_inlier_num))

    if i == 0:
        dist_epipolarline2_pts1 = dist_to_epipolarline(pts1_homo_t, pts2_homo_t, F)
        print("Distances before ransac: {0}".format(np.abs(dist_epipolarline2_pts1)))

#Run sevenpoint algorithm with the best inliers
pts1_best, pts2_best = pts1[best_inlier_idx, :], pts2[best_inlier_idx, :]
Farray = sevenpoint(pts1_best, pts2_best, M)

#Find best F maximizing inliers within Farray
F_best = None
max_inlier_num = 0
for F in Farray:
    epipolar_lines = (pts2_homo_t @ F)
    dist_epipolarline2_pts1 = dist_to_epipolarline(pts1_homo_t, pts2_homo_t, F)
    inlier_idx = np.where(np.abs(dist_epipolarline2_pts1) < 2.0)[0]
    if inlier_idx.shape[0] > max_inlier_num:
        max_inlier_num = inlier_idx.shape[0]
        F_best = F

dist_epipolarline2_pts1 = dist_to_epipolarline(pts1_homo_t, pts2_homo_t, F)
print("Distances after ransac: {0}".format(np.abs(dist_epipolarline2_pts1)))

return F_best, best_inlier_idx
```

Q5.2 Rodrigues and Invsere Rodrigues (15 points)

So far we have independently solved for camera matrix, $\mathbf{M}_j$ and 3D points $\mathbf{w}_i$. In bundle adjustment, we will jointly optimize the reprojection error with respect to the points $\mathbf{w}_i$ and the camera matrix $\mathbf{C}_j$.

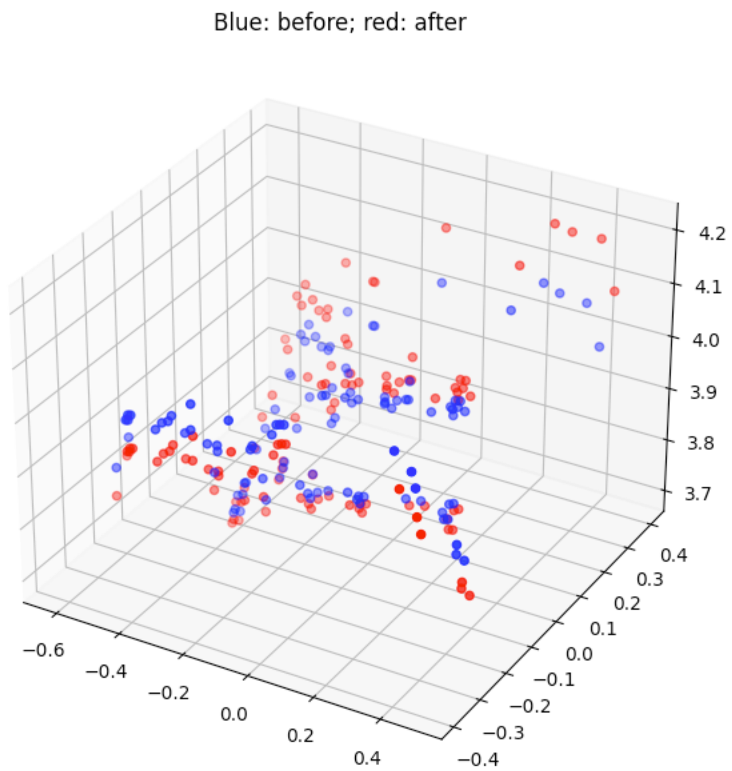
$$err = \sum_{ij} \|\mathbf{x}_{ij} - Proj(\mathbf{C}_j, \mathbf{w}_i)\|^2 ,$$

where $\mathbf{C}_j = \mathbf{K}_j\mathbf{M}_j$, same as in Q3.2.

```
'''
Q5.2: Rodrigues formula.
   Input:  r, a 3x1 vector
   Output: R, a rotation matrix
'''
def rodrigues(r):
   # Replace pass by your implementation
   eps = 0.001
   theta = np.sum(r**2)**0.5
   if np.abs(theta) < eps:
      return np.eye(3, dtype=np.float32)
   else:
      u = r / theta
      u1, u2, u3 = u[0, 0], u[1, 0], u[2, 0]
      # u1, u2, u3 = u[0], u[1], u[2]
      u_cross = np.array([[0, -u3, u2], [u3, 0, -u1], [-u2, u1, 0]], dtype=np.float32)

      R = np.eye(3, dtype=np.float32) * np.cos(theta) \
         + (1 - np.cos(theta)) * (u @ u.transpose()) \
         + u_cross * np.sin(theta)

      return R

'''
Q5.2: Inverse Rodrigues formula.
   Input:  R, a rotation matrix
   Output: r, a 3x1 vector
'''

## additionally defined functions
def eq(a, b):
   eps = 0.001
   return np.abs(a - b) < eps

def gt(a, b):
   eps = 0.001
   return a - b > eps
```

```python
def S_half(r):
    length = np.sum(r**2)**0.5
    r1, r2, r3 = r[0, 0], r[1, 0], r[2, 0]
    if (eq(length, np.pi) and eq(r1, r2) and eq(r1, 0) and gt(0, r3)) \
        or (eq(r1, 0) and gt(0, r2)) \
        or (gt(0, r1)):
        return -r
    else:
        return r

def arctan2(y, x):
    if gt(x, 0):
        return np.arctan(y / x)
    elif gt(0, x):
        return np.pi + np.arctan(y / x)
    elif eq(x, 0) and gt(y, 0):
        return np.pi*0.5
    elif eq(x, 0) and gt(0, y):
        return -np.pi*0.5

def invRodrigues(R):
    # Replace pass by your implementation
    eps = 0.001
    A = (R - R.transpose())*0.5
    a32, a13, a21 = A[2, 1], A[0, 2], A[1, 0]
    rho = np.array([[a32], [a13], [a21]], dtype=np.float32)
    s = np.sum(rho**2)**0.5
    c = (R[0, 0]+R[1, 1]+R[2, 2] - 1) / 2.0
    if eq(s, 0) and eq(c, 1):
        return np.zeros((3, 1), dtype=np.float32)
    elif eq(s, 0) and eq(c, -1):
        V = R+np.eye(3, dtype=np.float32)
        # find a nonzero column of V
        mark = np.where(np.sum(V**2, axis=0) > eps)[0]
        v = V[:, mark[0]]
        u = v / (np.sum(v**2)**0.5)
        r = S_half(u*np.pi)
        return r
    elif not eq(s, 0):
        u = rho / s
        theta = arctan2(s, c)
        return u*theta
```

Q5.3 Bundle Adjustment (10 points)




```
 Reprojection error of M2 before BA: 2892.757924
```

```
Reprojection error of M2_BA: 11.151892
```

```
'''
Q5.3: Rodrigues residual.
   Input:  K1, the intrinsics of camera 1
         M1, the extrinsics of camera 1
         p1, the 2D coordinates of points in image 1
         K2, the intrinsics of camera 2
         p2, the 2D coordinates of points in image 2
         x, the flattened concatenationg of P, r2, and t2.
   Output: residuals, 4N x 1 vector, the difference between original and estimated projections
'''
## additionally defined functions
def flatten(P, r2, t2):
   # P: (N, 3)
   # r2: (3, 1)
   # t2: (3, 1)
   # (3+3+N*3,)
   return np.concatenate((r2.reshape(-1), t2.reshape(-1), P.reshape(-1)), axis=0)
```

```python
def inflate(x):
    r2 = x[0:3].reshape(-1, 1)
    t2 = x[3:6].reshape(-1, 1)
    P  = x[6:].reshape(-1, 3)
    return P, r2, t2

#Addi functions

def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation
    P, r2, t2 = inflate(x)
    R2 = rodrigues(r2)
    M2 = np.concatenate((R2, t2), axis=1)
    points_word_homo = np.concatenate( ( P, np.ones( (P.shape[0], 1) ) ), axis=1 ).transpose()
    points_img1_rep_homo = K1 @ M1 @ points_word_homo
    points_img1_rep = points_img1_rep_homo[0:2, :] / points_img1_rep_homo[2, :]
    points_img2_rep_homo = K2 @ M2 @ points_word_homo
    points_img2_rep = points_img2_rep_homo[0:2, :] / points_img2_rep_homo[2, :]

    error_img1_rep = (p1 - points_img1_rep).reshape(-1)
    error_img2_rep = (p2 - points_img2_rep).reshape(-1)

    residuals = np.concatenate((error_img1_rep, error_img2_rep), axis=0)
    return residuals

'''
Q5.3 Bundle adjustment.
    Input:  K1, the intrinsics of camera 1
            M1, the extrinsics of camera 1
            p1, the 2D coordinates of points in image 1
            K2,  the intrinsics of camera 2
            M2_init, the initial extrinsics of camera 1
            p2, the 2D coordinates of points in image 2
            P_init, the initial 3D coordinates of points
    Output: M2, the optimized extrinsics of camera 1
            P2, the optimized 3D coordinates of points
            o1, the starting objective function value with the initial input
            o2, the ending objective function value after bundle adjustment

    Hints:
    (1) Use the scipy.optimize.minimize function to minimize the objective function,
rodriguesResidual.
        You can try different (method='..') in scipy.optimize.minimize for best results.
'''

def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    # Replace pass by your implementation

    obj_start = obj_end = 0
```

```
# ----- TODO -----
# YOUR CODE HERE
p1 = p1.transpose()
p2 = p2.transpose()

residual = lambda x: rodriguesResidual(K1, M1, p1, K2, p2, x)
R2_init = M2_init[:, 0:3]
t2_init = M2_init[:, 3]
r2_init = invRodrigues(R2_init)
x_init = flatten(P_init, r2_init, t2_init)
x_optim, _ = scipy.optimize.leastsq(residual, x_init)

# print('Reprojection error after BA: %f' % np.sum(residual(x_optim)**2))

P2, r2, t2 = inflate(x_optim)
R2 = rodrigues(r2)
M2 = np.concatenate((R2, t2), axis=1)

obj_start = rodriguesResidual(K1, M1, p1, K2, p2, x_init)
obj_end = rodriguesResidual(K1, M1, p1, K2, p2, x_optim)

# print('Object start: {}'.format(obj_start))
# print('Object end: {}'.format(obj_end))

return M2, P2, obj_start, obj_end

# raise NotImplementedError()
# return M2, P, obj_start, obj_start
```

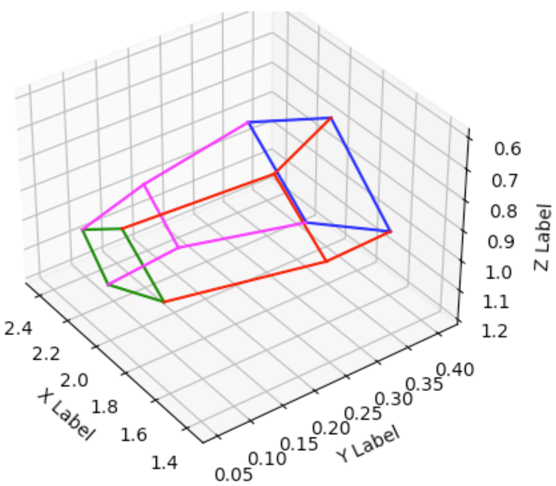
Q6.1 (Extra Credit - 15 points) Write a function to compute the 3D keypoint locations `P` given the 2d part detections `pts1`, `pts2` and `pts3` and the camera projection matrices `C1`, `C3`, `C3`. The camera matrices are given in the numpy files.

```
[P, err] = MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres)
```

In your write-up: Describe the method you used to compute the 3D locations and include an image of the Reconstructed 3D points with the points connected using the helper function `plot_3d_keypoint(P)` with the reprojection error. **Please include the code snippets in your write-up.**

I modified triangulate function to generate A matrix with 3 points and compute word coordinates according to it as follow:

$$Ax = \begin{bmatrix} C1_1 - x1C1_3 \\ C1_2 - y1C1_3 \\ C2_1 - x2C2_3 \\ C2_2 - y2C2_3 \\ C3_1 - x3C3_3 \\ C3_2 - y3C3_3 \end{bmatrix} = 0$$



When only one of points' confidence score is smaller than threshold, I just computed 3D coordinates using the other two points.

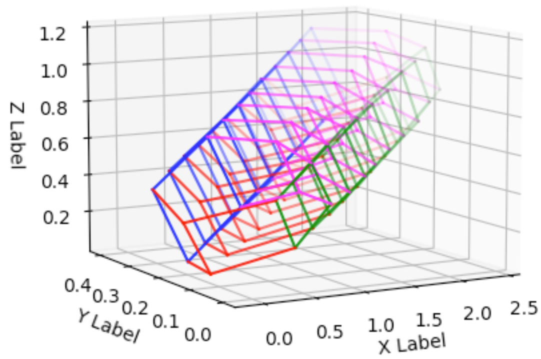
```
def triangulate_3pts(C1, pts1, C2, pts2, C3, pts3):
    # Replace pass by your implementation
    coord_word_list = []
    err_reproject = 0

    for i in range(pts1.shape[0]):
```

```
        x1, y1 = pts1[i, :]
        x2, y2 = pts2[i, :]
        x3, y3 = pts3[i, :]

        # A matrix
        A0 = C1[0, :] - x1*C1[2, :]
        A1 = C1[1, :] - y1*C1[2, :]
        A2 = C2[0, :] - x2*C2[2, :]
        A3 = C2[1, :] - y2*C2[2, :]
        A4 = C3[0, :] - x3*C3[2, :]
        A5 = C3[1, :] - y3*C3[2, :]
        A = np.stack((A0, A1, A2, A3, A4, A5), axis=0)

        #Get world coordinates through SVD
        U, s, Vt = np.linalg.svd(A)
        coord_word = Vt[-1, :] #(4,)
        coord_word = coord_word[0:3] / coord_word[3] #(3,)
        coord_word_list.append(coord_word)

        #Ceprojection
        coord_word_homo = np.zeros((4, 1), dtype=np.float32)
        coord_word_homo[0:3, 0] = coord_word
        coord_word_homo[3, 0] = 1

        coord_cam1_rep = C1 @ coord_word_homo
        coord_cam2_rep = C2 @ coord_word_homo
        coord_cam3_rep = C3 @ coord_word_homo

        #Calculate reprojection error
        x1_cam1_rep, y1_cam1_rep = coord_cam1_rep[0:2, 0] / coord_cam1_rep[2, 0]
        x2_cam2_rep, y2_cam2_rep = coord_cam2_rep[0:2, 0] / coord_cam2_rep[2, 0]
        x3_cam3_rep, y3_cam3_rep = coord_cam3_rep[0:2, 0] / coord_cam3_rep[2, 0]


        err_reproject += (x1_cam1_rep-x1)**2 + (y1_cam1_rep-y1)**2 + (x2_cam2_rep-x2)**2 +
(y2_cam2_rep-y2)**2 + (x3_cam3_rep-x3)**2 + (y3_cam3_rep-y3)**2

    P = np.stack(coord_word_list, axis=0)
    print(P.shape)

    return P, err_reproject


def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres = 300):

    p1s, p2s, p3s = pts1[:,:2], pts2[:,:2], pts3[:,:2]
    confidence1s, confidence2s, confidence3s = pts1[:,2], pts2[:,2], pts3[:,2]

    P, err = triangulate_3pts(C1, p1s, C2, p2s ,C3, p3s)
    P_12, _ = triangulate(C1, p1s, C2, p2s)
```

```python
    P_23, _ = triangulate(C2, p2s, C3, p3s)
    P_31, _ = triangulate(C3, p3s, C1, p1s)

    # If only a point's confidence score is smaller than threshold, don't consider it and get P from
triangulation using the other two points
    N_pts = len(pts1)
    for i in range(N_pts):
        if confidence1s[i] < Thres and confidence2s[i] > Thres and confidence3s[i] > Thres:
            P[i] = P_23[i]

        if confidence2s[i] < Thres and confidence3s[i] > Thres and confidence1s[i] > Thres:
            P[i] = P_31[i]

        if confidence3s[i] < Thres and confidence1s[i] > Thres and confidence2s[i] > Thres:
            P[i] = P_12[i]

    return P, err
```

Q6.2 (Extra Credit - 15 points)

From the previous question you have done a 3D reconstruction at a time instance. Now you are going to iteratively repeat the process over time and compute a spatio temporal reconstruction of the car. The images in the `data/q6` folder shows the motion of the car at an intersection captured from multiple views. The images are given as (`cam1_time0.jpg, ..., cam1_time9.jpg`) for camera 1 and (`cam2_time0.jpg, ..., cam2_time9.jpg`) for camera2 and (`cam3_time0.jpg, ..., cam3_time9.jpg`) for camera3. The corresponding detections and camera matrices are given in (`time0.npz, ..., time9.npz`). Use the above details and compute the spatio temporal reconstruction of the car for all 10 time instances and plot them by completing the `plot_3d_keypoint_video` function. A sample plot with the first and last time instance reconstution of the car with the reprojections shown in the Figure 10. **Please include the code snippets in your write-up.**




```python
def plot_3d_keypoint_video(pts_3d_video):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    for i in range(10):
        pts_word = pts_3d_video[i]
        for j in range(len(connections_3d)):
            index0, index1 = connections_3d[j]
            xline = [pts_word[index0,0], pts_word[index1,0]]
            yline = [pts_word[index0,1], pts_word[index1,1]]
            zline = [pts_word[index0,2], pts_word[index1,2]]
            ax.plot(xline, yline, zline, color = colors[j], alpha = 0.1 * i)
        np.set_printoptions(threshold = 1e6, suppress = True)

    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()
```