16-820 Advanced Computer Vision: Homework 3 (Fall 2024)
3D Reconstruction

Ryan Wu (Andrew ID: weihuanw)

Wednesday October 23, 2024

Q1.1
Suppose two cameras fixate on a point x 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 F_{33} element of the fundamental matrix is zero.

For points $x_1 = [x_1, y_1, 1]^T$ and ${x_2}^T = [x_2, y_2, 1]$, the epipolar constraint with the fundamental matrix F is given by:

$$x_2^T F x_1 = 0$$

The epipolar constraint equation can be expanded to:

$$\begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$$

With further multiplication, we get:

$$x_2(F_{11}x_1 + F_{12}y_1 + F_{13}) + y_2(F_{21}x_1 + F_{22}y_1 + F_{23}) + (F_{31}x_1 + F_{32}y_1 + F_{33}) = 0$$

Evaluating the above at the optical centers, the points become $x_1 = (0, 0, 1)^T$, $x_2 = (0, 0, 1)^T$ and the above equation simplifies to:

$$F_{33} = 0$$

Q1.2

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. Back up your argument with relevant equations. You may assume both cameras have the same intrinsics.

Considering only pure translation parallel to the x-axis and assume both cameras have the same intrinsics, the epipolar lines for $\mathbf{l}_1$ and $\mathbf{l}_2$ can be represented as:

$$\mathbf{l}_1 = \mathbf{x_2}^T\mathbf{F}^T, \quad \mathbf{l}_2 = \mathbf{x_1}^T\mathbf{F}$$

where the fundamental matrix $\mathbf{F}$ is given by:

$$\mathbf{F} = \mathbf{K}^{-T}\mathbf{E}\mathbf{K}^{-1}$$

and the essential matrix $\mathbf{E}$ is given by:

$$\mathbf{E} = \hat{\mathbf{T}}\mathbf{R}$$

$$\mathbf{T} = \begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix}, \quad \hat{\mathbf{T}} \text{ (skew-symmetric matrix)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Substituting $\mathbf{E}$ into $\mathbf{F}$:

$$\mathbf{F} = \mathbf{K}^{-T} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \mathbf{K}^{-1}$$

Substituting $\mathbf{F}$ into $\mathbf{l}_1$ with ${x_2}^T = [x_2, y_2, 1]$, we get:

$$\mathbf{l}_1 = \mathbf{x_2}^T\mathbf{F}^T = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t_x \\ 0 & -t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & -t_x & y_2 t_x \end{bmatrix}$$

Substituting $\mathbf{F}$ into $\mathbf{l}_2$ with ${x_2}^T = [x_2, y_2, 1]$, we get:

$$\mathbf{l}_2 = \mathbf{x_1}^T\mathbf{F} = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & t_x & -y_1 t_x \end{bmatrix}$$

Given the line equation in the form $ax + by + c = 0$:

$$\mathbf{l}_1 = 0 \cdot x - t_x \cdot y + y_2 t_x = 0 \implies t_x y = y_2 t_x$$

$$\mathbf{l}_2 = 0 \cdot x + t_x \cdot y - y_1 t_x = 0 \implies t_x y = y_1 t_x$$

Thus,

$$y = y_1, \quad y = y_2 \quad (\text{if } t_x \neq 0)$$

proving both epipolar lines are parallel.

Q1.3

Suppose we have an inertial sensor that 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 timestamps? Suppose the camera intrinsics ($\mathbf{K}$) are known, express the essential matrix ($\mathbf{E}$) and the fundamental matrix ($\mathbf{F}$) in terms of K, $\mathbf{R_{rel}}$, and $\mathbf{t_{rel}}$.

Let P denote a point in the world coordinate system. The corresponding projection of this point onto the image plane at frame 1 is $\mathbf{p}_1$, and at frame 2 is $\mathbf{p}_2$, which can be represented by:

$$\mathbf{p}_1 = \mathbf{K}\left(\mathbf{R}_1\mathbf{P} + \mathbf{t}_1\right) \tag{1}$$

$$\mathbf{p}_2 = \mathbf{K}\left(\mathbf{R}_2\mathbf{P} + \mathbf{t}_2\right) \tag{2}$$

Rearranging (1) to factor out P, we get:

$$\mathbf{P} = \mathbf{R}_1^{-1}\left(\mathbf{K}^{-1}\mathbf{p}_1 - \mathbf{t}_1\right) \tag{3}$$

Substituting (3) back into (2) and expanding, we get:

$$\mathbf{p}_2 = \mathbf{K}\left(\mathbf{R}_2\mathbf{R}_1^{-1}\mathbf{K}^{-1}\mathbf{p}_1 + \mathbf{t}_2 - \mathbf{R}_2\mathbf{R}_1^{-1}\mathbf{t}_1\right)$$

The relative rotation between frame 1 and frame 2 can be expressed as:

$$\mathbf{R}_{\text{rel}} = \mathbf{R}_2\mathbf{R}_1^{-1}$$

The relative translation between frame 1 and frame 2 can be expressed as:

$$\mathbf{t}_{\text{rel}} = \mathbf{t}_2 - \mathbf{R}_{\text{rel}}\mathbf{t}_1$$

$\mathbf{p}_2$ can be simplified and expressed as:

$$\mathbf{p}_2 = \mathbf{K}\mathbf{R}_{\text{rel}}\mathbf{K}^{-1}\mathbf{p}_1 + \mathbf{K}\mathbf{t}_{\text{rel}}$$

Finally, the essential and fundamental matrices can be expressed as:

$$\mathbf{E} = \hat{\mathbf{t}}_{\text{rel}}\mathbf{R}_{\text{rel}}, \quad \mathbf{F} = \mathbf{K}^{-T}\hat{\mathbf{t}}_{\text{rel}}\mathbf{R}_{\text{rel}}\mathbf{K}^{-1}$$

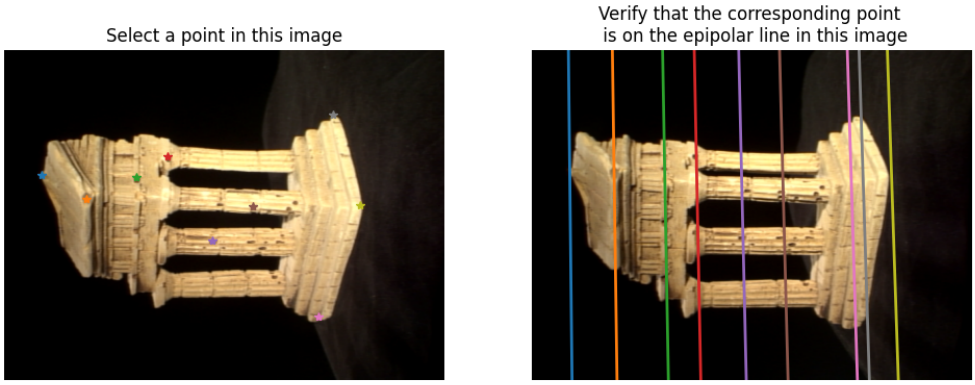
Q2.1

Finish the function `eightpoint` in `q2_1_eightpoint.py`. Save your matrix F and scale M to the file `q2_1.npz`. In your write-up, write your recovered F, include an image of some example output of `displayEpipolarF`, and include the code snippet of the `eightpoint` function.

The recovered fundamental matrix **F** is:

$$\mathbf{F} = \begin{bmatrix} 9.80213864 \times 10^{-10} & -1.32271663 \times 10^{-7} & 1.12586847 \times 10^{-3} \\ -5.72416248 \times 10^{-8} & 2.97011941 \times 10^{-9} & -1.17899320 \times 10^{-5} \\ -1.08270296 \times 10^{-3} & 3.05098538 \times 10^{-5} & -4.46974798 \times 10^{-3} \end{bmatrix}$$

The image of some example output of displayEpipolarF:



The example output of displayEpipolarF (eight-point).

```python
def eightpoint(pts1, pts2, M):
    # Normalize the input points
    T = np.array([[1 / M, 0, 0], [0, 1 / M, 0], [0, 0, 1]])
    pts1_norm = pts1 / M
    pts2_norm = pts2 / M

    # Constructing the A matrix
    A = np.zeros((pts1.shape[0], 9))
    for i in range(pts1.shape[0]):
        x1, y1 = pts1_norm[i]
        x2, y2 = pts2_norm[i]
        A[i] = np.array([x2 * x1, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1, 1])

    # Solve for least square solution using SVD
    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)

    # Enforcing the singularity condition
    F = _singularize(F)

    # Refining the computed fundamental matrix
    F = refineF(F, pts1_norm, pts2_norm)

    # Unscale the fundamental matrix
    F = np.dot(np.dot(T.T, F), T)

    return F
```

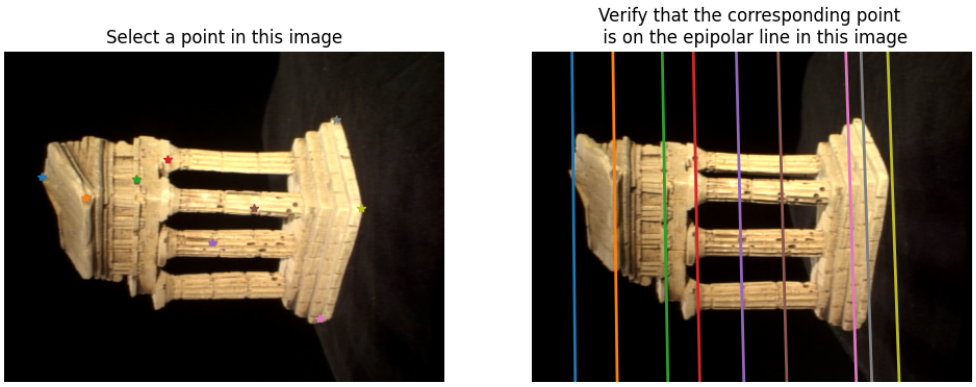
Q2.2 [Extra Credit]
Finish the function `sevenpoint` in `q2_2_sevenpoint.py`. Save your matrix F and scale M to the file `q2_2.npz`. In your write-up, write your recovered F, include an image of some example output of `displayEpipolarF`, and include the code snippet of the `sevenpoint` function.

The recovered fundamental matrix **F** is:

$$\mathbf{F} = \begin{bmatrix} 1.21556012 \times 10^{-7} & 7.13923261 \times 10^{-7} & 2.12714621 \times 10^{-3} \\ -3.48090488 \times 10^{-8} & -6.51683250 \times 10^{-8} & -2.06509491 \times 10^{-4} \\ -2.28939753 \times 10^{-3} & 4.05924223 \times 10^{-4} & -1.78169873 \times 10^{-2} \end{bmatrix}$$

The image of some example output of `displayEpipolarF` using the best fundamental matrix:

$$\mathbf{F}_{\text{best}} = \begin{bmatrix} -3.05634602 \times 10^{-7} & -2.78377644 \times 10^{-7} & 3.53199537 \times 10^{-3} \\ -1.75530973 \times 10^{-7} & 7.18761192 \times 10^{-9} & -2.64429653 \times 10^{-5} \\ -3.26788820 \times 10^{-3} & 9.69027880 \times 10^{-5} & -3.82909922 \times 10^{-2} \end{bmatrix}, \quad \text{Error : } 8.813954530721533$$



The example output of displayEpipolarF (seven-point).

```python
def sevenpoint(pts1, pts2, M):
    Farray = []
    # Normalize the input points
    T = np.array([[1 / M, 0, 0], [0, 1 / M, 0], [0, 0, 1]])
    pts1_norm = pts1 / M
    pts2_norm = pts2 / M

    # Constructing the A matrix
    A = np.zeros((7, 9))
    for i in range(7):
        x1, y1 = pts1_norm[i]
        x2, y2 = pts2_norm[i]
        A[i] = np.array([x1 * x2, x1 * y2, x1, y1 * x2, y1 * y2, y1, x2, y2, 1])
        # A[i] = np.array([x2 * x1, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1, 1])

    # Solve for least square solution using SVD
    _, _, V = np.linalg.svd(A)

    # Pick the last two colum vector of vT.T (the two null space solution f1 and f2)
    f1 = V[-1].reshape(3, 3)
    f2 = V[-2].reshape(3, 3)

    # Coefficients of the polynomial equation
    # Set up the polynomial equation
    def cubic_poly(a):
        F = a * f1 + (1 - a) * f2
        return np.linalg.det(F)

    # Coefficients of the polynomial equation
    coeff = [cubic_poly(0), cubic_poly(1/3), cubic_poly(2/3), cubic_poly(1)]

    # Solve the polynomial equation (np.polynomial.polynomial.polyroots)
    roots = np.polynomial.polynomial.polyroots(coeff).real
    # print("Roots of the polynomial:\n", roots)

    # Unscale the fundamental matrixes
    for root in roots:
        F = root * f1 + (1 - root) * f2
        F = _singularize(F)
        F = np.dot(np.dot(T.T, F), T)  # Unnormlize the fundamental matrix
        Farray.append(F)

    return Farray
```

Q3.1
Complete the function `essentialMatrix` in `q3_1_essential_matrix.py` to compute the essential matrix E given F, K_1, and K_2 with the signature: $E =$ `essentialMatrix`(F, K_1, K_2). Save your estimated E using F from the eight-point algorithm to `q3_1.npz`. In your write-up, write your estimated E, and include the code snippet of the `essentialMatrix` function.

The estimated essential matrix **E** is:

$$\mathbf{E} = \begin{bmatrix} 2.26587821 \times 10^{-3} & -3.06867395 \times 10^{-1} & 1.66257398 \times 10^{0} \\ -1.32799331 \times 10^{-1} & 6.91553934 \times 10^{-3} & -4.32775554 \times 10^{-2} \\ -1.66717617 \times 10^{0} & -1.33444257 \times 10^{-2} & -6.72047195 \times 10^{-4} \end{bmatrix}$$

```python
def essentialMatrix(F, K1, K2):
    # Compute the essential matrix
    E = K2.T @ F @ K1

    return E
```

Q3.2
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: $[w, \mathbf{err}] = \texttt{triangulate}(C_1, \mathbf{pts1}, C_2, \mathbf{pts2})$. In your write-up, write down the expression for the matrix A_i for triangulating a pair of 2D coordinates in the image to a 3D point, and include the code snippet of the `triangulate` function.

Given two cameras, C_1 and C_2, and two corresponding 2D image points $p_1 = (u_1, v_1, 1)^T$ from camera 1 and $p_2 = (u_2, v_2, 1)^T$ from camera 2, and 3D coordinate $\tilde{w}_i = \begin{bmatrix} x_i, y_i, z_i, 1 \end{bmatrix}^T$, we can obtain the 2D homogeneous coordinates projected to camera 1 and camera 2 as:

$$C_1 \cdot \tilde{w}_i = p_1 \implies C_1 \cdot \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}$$

$$C_2 \cdot \tilde{w}_i = p_2 \implies C_2 \cdot \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix}$$

Each camera provides two constraints:

$$\begin{aligned} u_1(C_1[2] \cdot \tilde{w}_i) - (C_1[0] \cdot \tilde{w}_i) &= 0 \\ v_1(C_1[2] \cdot \tilde{w}_i) - (C_1[1] \cdot \tilde{w}_i) &= 0 \\ u_2(C_2[2] \cdot \tilde{w}_i) - (C_2[0] \cdot \tilde{w}_i) &= 0 \\ v_2(C_2[2] \cdot \tilde{w}_i) - (C_2[1] \cdot \tilde{w}_i) &= 0 \end{aligned}$$

Combining these four equations into a matrix form:

$$A_i \cdot \tilde{w}_i = 0$$

We get a 4×4 matrix A_i:

$$A_i = \begin{bmatrix} u_1 C_1[2] - C_1[0] \\ v_1 C_1[2] - C_1[1] \\ u_2 C_2[2] - C_2[0] \\ v_2 C_2[2] - C_2[1] \end{bmatrix}$$

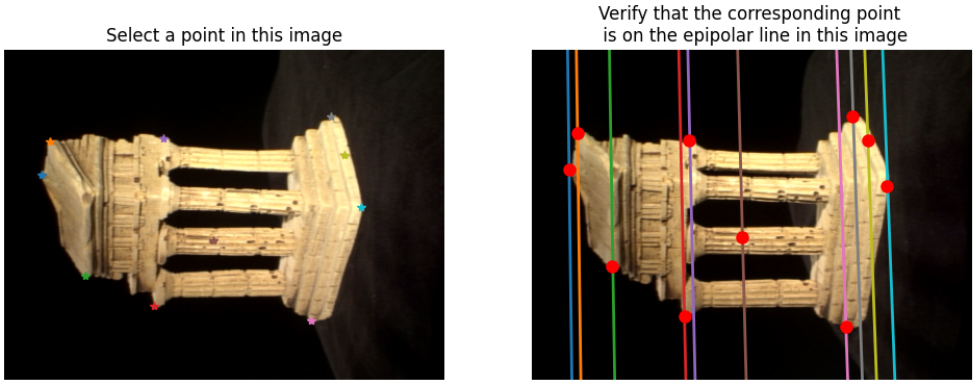
```python
def triangulate(C1, pts1, C2, pts2, C3=None, pts3=None):
    # Initialize the 3D points and error
    number_of_points = pts1.shape[0]
    P = np.zeros((number_of_points, 3))
    err = 0

    # Loop through the points
    A = np.zeros((4, 4))
    for i in range(number_of_points):
        u1, v1 = pts1[i]
        u2, v2 = pts2[i]

        # 3 views
        if C3 is not None and pts3 is not None:
            u3, v3 = pts3[i]
            # Construct the matrix A for 3 views
            A = np.zeros((6, 4))
            A[0] = u1 * C1[2] - C1[0]
            A[1] = v1 * C1[2] - C1[1]
            A[2] = u2 * C2[2] - C2[0]
            A[3] = v2 * C2[2] - C2[1]
            A[4] = u3 * C3[2] - C3[0]
            A[5] = v3 * C3[2] - C3[1]
        # 2 views
        else:
            # Construct the matrix A for 2 views
            A = np.zeros((4, 4))
            A[0] = u1 * C1[2] - C1[0]
            A[1] = v1 * C1[2] - C1[1]
            A[2] = u2 * C2[2] - C2[0]
            A[3] = v2 * C2[2] - C2[1]

        # Solve for the least square solution
        _, _, V = np.linalg.svd(A)
        w = V[-1, :]   # w represents the 3D point in homogeneous coordinates
        w /= w[3]      # Convert to non-homogeneous coordinates
        P[i, :] = w[0:3]

        # Calculate the reprojection error
        p1 = C1.dot(w.T)
        p1 /= p1[2]
        p2 = C2.dot(w.T)
        p2 /= p2[2]

        # 3 views
        if C3 is not None and pts3 is not None:
            p3 = C3.dot(w.T)
            p3 /= p3[2]
            err += np.linalg.norm(pts1[i] - p1[:2]) ** 2 + \
                   np.linalg.norm(pts2[i] - p2[:2]) ** 2 + \
                   np.linalg.norm(pts3[i] - p3[:2]) ** 2
        else:
            err += np.linalg.norm(pts1[i] - p1[:2]) ** 2 + \
                   np.linalg.norm(pts2[i] - p2[:2]) ** 2

    return P, err
```

Q3.3
Complete the function `findM2` in `q3_2_triangulate.py` to obtain the correct $M2$ from $M2s$ by testing the four solutions through triangulations. Save the correct $M2$, the corresponding $C2$, and 3D points P to `q3_3.npz`. In your write-up, include the code snippet of the `findM2` function.

```python
def findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):
    """
    Q2.2: Function to find 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'.

    """
    # Recover the essential matrix
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    E = essentialMatrix(F, K1, K2)
    M2s = camera2(E)

    # Initialize the best error, M2, C2, and P
    best_error = float('inf')
    best_M2 = None
    best_C2 = None
    best_P = None

    # Initialize the camera matrix for C1
    M1 = np.hstack((np.eye(3), np.zeros((3, 1))))
    C1 = K1.dot(M1)

    # Loop through the possible M2s
    for i in range(M2s.shape[2]):
        M2 = M2s[:, :, i]
        C2 = K2.dot(M2)

        # Triangulate the points
        P, err = triangulate(C1, pts1, C2, pts2)

        # Update the best error, M2, C2, and P
        if err < best_error:
            best_error = err
            best_M2 = M2
            best_C2 = C2
            best_P = P

    # Save the results
    np.savez(filename, M2=best_M2, C2=best_C2, P=best_P)

    return M2, C2, P
```

Q4.1
In `q4_1_epipolar_correspondence.py`, finish the function `epipolarCorrespondence` with the signature: $[x_2, y_2] =$ `epipolarCorrespondence`$(im_1, im_2, F, x_1, y_1)$. Save the matrix F, points pts_1 and pts_2 which you used to generate the screenshot to the file `q4_1.npz`. In your write-up, include a screenshot of `epipolarMatchGUI` with some detected correspondences; include the code snippet of the `epipolarCorrespondence` function.

The screenshot output of `epipolarMatchGUI`:



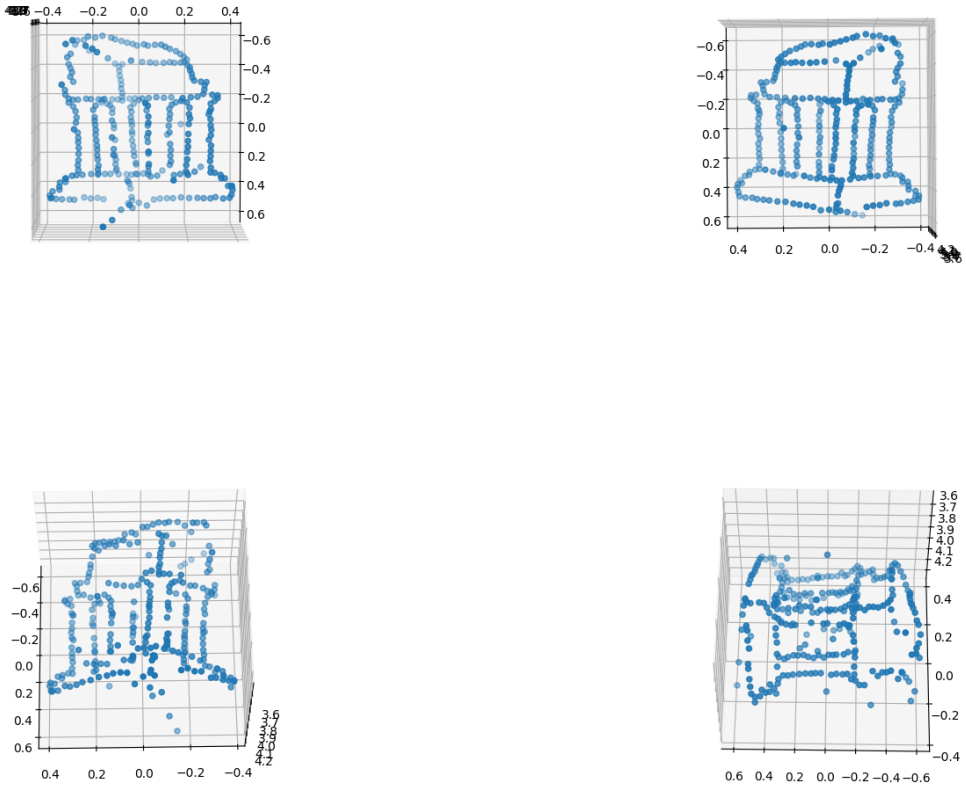
The screenshot output of `epipolarMatchGUI`.

```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Get the epipolar line in the second image
    v = np.array([x1, y1, 1])
    l2 = F.dot(v)

    # Get the image dimensions
    h, w = im2.shape[:2]

    # Define the search window
    search_window = 30

    # Initialize the best match and the best match error
    best_match = None
    best_match_error = float('inf')

    # Gaussian weighting kernel
    gaussian_size = 2 * search_window + 1
    gaussian = cv2.getGaussianKernel(gaussian_size, 1)
    gaussian = gaussian.dot(gaussian.T)

    # Loop through the search window
    for y2 in range(max(0, y1 - search_window), min(h, y1 + search_window)):
        x2 = int(-(l2[1] * y2 + l2[2]) / l2[0])

        # Check if the pixel is within the image
        if x2 < search_window or x2 >= w - search_window:
            continue

        # Get image patches for the two images
        patch1 = im1[y1 - search_window:y1 + search_window + 1, x1 - search_window:x1 +
            search_window + 1]
        patch2 = im2[y2 - search_window:y2 + search_window + 1, x2 - search_window:x2 +
            search_window + 1]

        # Apply the guassian weighting
        if patch1.shape == patch2.shape == (gaussian_size, gaussian_size, 3):
            patch1_weighted = patch1 * gaussian[:, :, np.newaxis]
            patch2_weighted = patch2 * gaussian[:, :, np.newaxis]

            # Compute the Euclidean distance
            error = np.linalg.norm(patch1_weighted - patch2_weighted)

            # Update the best match
            if error < best_match_error:
                best_match_error = error
                best_match = (x2, y2)

    # Return the best match
    x2, y2 = best_match

    return x2, y2
```

Q4.2

Complete the `compute3D_pts` function in `q4_2_visualize.py`, which loads the necessary files from `../data/` to generate the 3D reconstruction using the `scatter` function from Matplotlib. Again, save the matrix F, matrices M_1, M_2, C_1, and C_2 which you used to generate the screenshots to the file `q4_2.npz`. In your write-up, take a few screenshots of the 3D visualization so that the outline of the temple is clearly visible, and include them in your write-up; include the code snippet of the `compute3D_pts` function in your write-up.

A few screenshots of the 3D visualization:



3D visualization with clearly visible temple outline at various angles.

```python
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    # Initialize the 3D points
    P = np.zeros((temple_pts1.shape[0], 3))

    # Initialize the corresponding points in the second image
    temple_pts2 = np.zeros_like(temple_pts1)

    # Loop through the points
    for i in range(temple_pts1.shape[0]):
        x1, y1 = temple_pts1[i]

        # Get the corresponding point in the second image
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)

        # Store the corresponding points in the second image
        temple_pts2[i] = [x2, y2]

    # Find the 3D points P
    M2, C2, P = findM2(F, temple_pts1, temple_pts2, intrinsics)

    return M2, C2, P
```

Q5.1
Implement the above algorithm with the signature: $[F, \textbf{inliers}] =$ `ransacF`$(pts_1, pts_2, M, nIters, \textbf{tol})$**. In your write-up, compare the result of RANSAC with the result of the eight-point algorithm when run on the noisy correspondences. Briefly explain the error metrics you used, how you decided which points were inliers, and any other optimizations you may have made.** $nIters$ **is the maximum number of iterations of RANSAC and tol is the tolerance of the error to be considered as inliers. Discuss the effect on the fundamental matrix by varying these values. Please include the code snippet of the** `ransacF` **function in your write-up.**

I used the epipolar error as the error metrics for my RANSAC implementation. It compares the results with the fundamental matrix's epipolar constraint. Points within this epipolar error will be considered an inlier. The fundamental matrix is also computed from the minimal set of points using the 8-point algorithm.

By varying the RANSAC parameters, we can observe that lower tolerance decreases the inlier count and the average re-projection error, likewise, higher tolerance increases the inlier count and the average re-projection error. It determines RANSAC's strictness in rejecting inliers and the accuracy of the estimated fundamental matrix. On the other hand, number of iterations determines RANSAC's overall robustness. However, longer iterations will increase computation time.

The RANSAC study results are shown below:

nIters	**Tolerance**	**Inlier Count**	**Average Re-projection Error**
1000	1	92	0.2722
1000	5	108	0.9748
1000	10	110	1.1068
1000	15	110	1.1068
100	10	110	1.1068
500	10	110	1.1068
1000	10	110	1.1068
1500	10	110	1.1068

Table 1: RANSAC results with varying parameters

```python
def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    # Initialize variables
    num = pts1.shape[0]  # number of points
    # handle special case when number of points is less than 8
    if num < 8:
        raise ValueError("Number of points is less than 8")

    # Initialize the best F, best inliers, max_inliers
    best_F = None
    best_inliers = np.zeros(pts1.shape[0], dtype=bool)
    max_inliers = 0

    # Convert to homogeneous coordinates
    pts1_homo = toHomogenous(pts1)
    pts2_homo = toHomogenous(pts2)

    for i in range(nIters):
        # Randomly choose points (7 or 8)
        # idx = np.random.choice(num, 7, replace=False)
        idx = np.random.choice(num, 8, replace=False)
        pts1_sample = pts1[idx]
        pts2_sample = pts2[idx]

        # Compute the fundamental matrix (7 or 8 point algorithm)
        # F = sevenpoint(pts1_sample, pts2_sample, M)
        F = eightpoint(pts1_sample, pts2_sample, M)

        # Calculate the error
        epi_error = calc_epi_error(pts1_homo, pts2_homo, F)

        # Determine inliers
        inliers = epi_error < tol
        inlier_count = np.sum(inliers)

        # Update the best F and inliers
        if inlier_count > max_inliers:
            max_inliers = inlier_count
            best_F = F
            best_inliers = inliers

    return best_F, best_inliers
```

Q5.2
Write a function that converts a Rodrigues vector r to a rotation matrix R with the signature R = `rodrigues`(r), as well as the inverse function that converts a rotation matrix R to a Rodrigues vector r with the signature r = `invRodrigues`(R). In your write-up, include the code snippets of the `rodrigues` and `invRodrigues` functions.

```python
def rodrigues(r):
    theta = np.linalg.norm(r)

    # No rotation, return identity
    if theta == 0:
        R = np.eye(3)
        return R

    # Unit vector
    u = r / theta

    # Skew-symmetric matric u_x
    u_x = np.array([[0, -u[2], u[1]],
                    [u[2], 0, -u[0]],
                    [-u[1], u[0], 0]])

    # Reshape for matrix multiplication
    u = u.reshape(-1, 1)

    # Rodrigues rotation formula
    I = np.eye(3)
    R = I * np.cos(theta) + (1 - np.cos(theta)) * (u @ u.T) + np.sin(theta) * u_x

    return R
```

```python
def invRodrigues(R):
    # Compute and construct varibles: A, rho, s, c, theta
    A = (R - R.T) / 2
    rho = np.array([A[2, 1], A[0, 2], A[1, 0]])
    s = np.linalg.norm(rho)
    c = (np.trace(R) - 1) / 2
    theta = np.arctan2(s, c)

    # Special Case
    if s == 0 and c == 1:
        r = np.zeros(3)  # no rotation
        return r

    elif s == 0 and c == -1:
        # 180 degree rotation
        R_plus_I = R + np.eye(3)
        v = R_plus_I[:, np.argmax(np.diag(R_plus_I))]
        u = v / np.linalg.norm(v)
        r = u * np.pi

        # Sign flip condition
        if np.linalg.norm(r) == np.pi and \
           ((r[0] == 0 and r[1] == 0 and r[2] < 0) or
            (r[0] == 0 and r[1] < 0) or
            (r[0] < 0)):
            r = -r
        return r

    # General case
    elif np.sin(theta) != 0:
        u = rho / s
        r = u * theta
        return r
```

Q5.3

Using this parameterization, write an optimization function with the signature residuals = `rodriguesResidual`$(K_1, M_1, p_1, K_2, p_2, x)$. Use this error function and Scipy's optimizer `minimize` to write a function that optimizes for the best extrinsic matrix and 3D points using the inlier correspondences from some `corresp_noisy.npz` and the RANSAC estimate of the extrinsics and 3D points as an initialization. The function should have the signature $[M_2, w, o_1, o_2]$ = `bundleAdjustment`$(K_1, M_1, p_1, K_2, M_2^{\text{init}}, p_2, w^{\text{init}})$. In your write-up, include an image of the original 3D points and the optimized points (use the provided `plot3Ddual` function). Report the re-projection error with your initial M_2 and w, as well as with the optimized matrices. Include the code snippets for `rodriguesResidual` and `bundleAdjustment` in your write-up.

Image of the original 3D points and the optimized points:

(a) Original 3D points (b) Optimized 3D points

3D visualization of the original and optimized points from different angles.

Re-projection Errors:

- Initial re-projection error: **42,244.3641**
- Optimized re-projection error: **10.1901**

Code snippet of the function `rodriguesResidual`:

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Extract the 3D points, P, r2, and t2 from x
    num_points = p1.shape[0]
    P = x[:3 * num_points].reshape(-1, 3)
    r2 = x[3 * num_points:3 * num_points + 3]
    t2 = x[3 * num_points + 3:]

    # Compute the rotation matrix R2
    R2 = rodrigues(r2)

    # Compute the projection matrix M2
    M2 = np.hstack((R2, t2.reshape(-1, 1)))

    # Homogeneous coordinates
    P_homo = np.hstack((P, np.ones((P.shape[0], 1))))

    # Compute the projection points p1_hat and p2_hat
    p1_hat_homo = K1 @ (M1 @ P_homo.T)
    p1_hat = (p1_hat_homo[:2] / p1_hat_homo[2]).T
    p2_hat_homo = K2 @ (M2 @ P_homo.T)
    p2_hat = (p2_hat_homo[:2] / p2_hat_homo[2]).T

    # Compute the residuals
    residuals = np.concatenate([(p1 - p1_hat).reshape([-1]), (p2 - p2_hat).reshape([-1])])

    return residuals
```

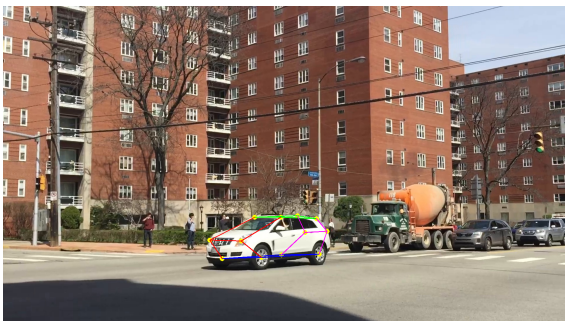
Code snippet of the function `bundleAdjustment`:

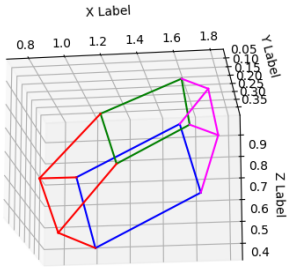
```python
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    # Extract the rotation and translation from M2_init
    r2 = invRodrigues(M2_init[:, :3])
    t2 = M2_init[:, 3]

    # Initial objective function value
    obj_start = np.concatenate((P_init.flatten(), r2.flatten(), t2))

    # Initial reprojection error
    residuals = rodriguesResidual(K1, M1, p1, K2, p2, obj_start)
    print(f"Initial reprojection error: {np.sum(residuals ** 2)}")

    # Optimize the objective function
    obj_end = opt.minimize(lambda x: np.sum(rodriguesResidual(K1, M1, p1, K2, p2, x) ** 2),
        obj_start).x

    # Optimized reprojection error
    residuals = rodriguesResidual(K1, M1, p1, K2, p2, obj_end)
    print(f"Optimized reprojection error: {np.sum(residuals ** 2)}")

    # Extract the 3D points, r2, and t2 from obj_end
    num_points = p1.shape[0]
    P = obj_end[:3 * num_points].reshape(-1, 3)
    r2 = obj_end[3 * num_points:3 * num_points + 3]
    t2 = obj_end[3 * num_points + 3:]

    # Compute the rotation matrix R2
    R2 = rodrigues(r2)

    # Compute the projection matrix M2
    M2 = np.hstack((R2, t2.reshape(-1, 1)))

    return M2, P, obj_start, obj_end
```

Q6.1 [Extra Credit]
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, C2, C3. The camera matrices are given in the numpy files. $[P, \text{err}] = \text{MultiviewReconstruction}(\text{C1}, \text{pts1}, \text{C2}, \text{pts2}, \text{C3}, \text{pts3}, \text{Thres})$. Save the best reconstruction (the 3D locations of the parts) from these parameters into a `q6_1.npz` file. In your write-up: Describe the method you used to compute the 3D locations. Include an image of the Reconstructed 3D points with the points connected using the helper function `plot_3d_keypoint(P)` with the reprojection error. Include the code snippets `MultiviewReconstruction` in your write-up.

I first modified the triangulation function to accept three views as input. To ensure the quality of the 3D reconstruction, I created a mask for the 3D points that performs confidence filtering. Using a predefined threshold, the implementation effectively filters out points that do not meet the required confidence level. Finally, the filtered 2D points are passed through the triangulation function to compute the corresponding 3D locations.

Image of the reconstructed 3D points with re-projection error (**229.0442**) :



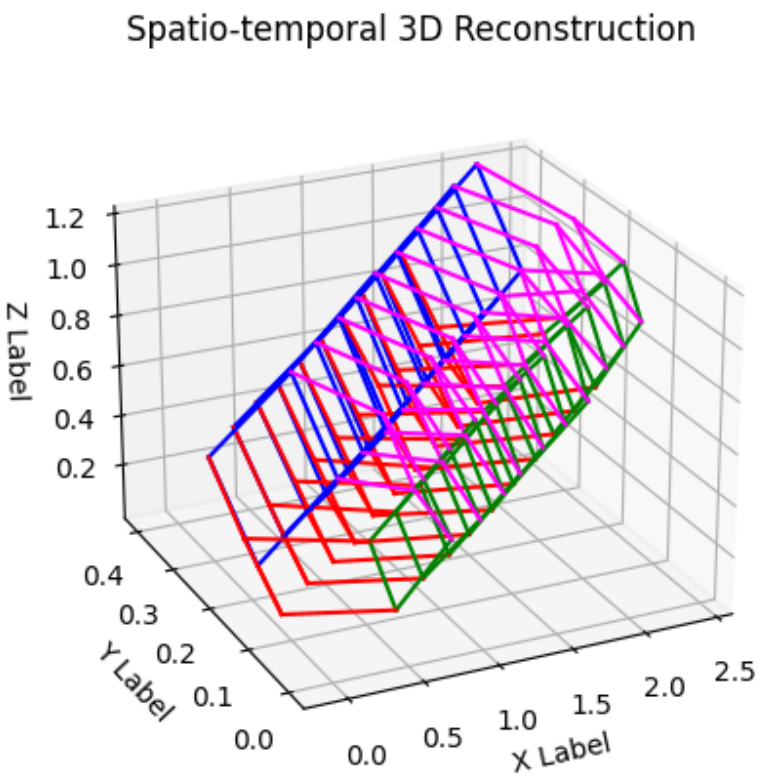


Visualization of the reconstructed 3D points.

```python
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres=100, filename="q6_1.npz"):
    # Extract the confidence values
    conf1 = pts1[:, 2]
    conf2 = pts2[:, 2]
    conf3 = pts3[:, 2]

    # Mask for points that meet the threshold
    mask = (conf1 > Thres) & (conf2 > Thres) & (conf3 > Thres)

    # Apply mask to the points
    u1, v1 = pts1[mask][:, 0], pts1[mask][:, 1]
    u2, v2 = pts2[mask][:, 0], pts2[mask][:, 1]
    u3, v3 = pts3[mask][:, 0], pts3[mask][:, 1]

    # Filter the points
    pts1_filtered = np.column_stack((u1, v1))
    pts2_filtered = np.column_stack((u2, v2))
    pts3_filtered = np.column_stack((u3, v3))

    # Initialize the 3D points and error
    total_err = 0
    valid_point_count = np.sum(mask)

    # Triangulate the 3D points
    P, total_err = triangulate(C1, pts1_filtered, C2, pts2_filtered, C3, pts3_filtered)

    # Calculate the average error
    err = total_err / valid_point_count

    # Save P & Print the results
    np.savez(filename, P=P)
    print("Total Points: " + str(valid_point_count))
    print("Projection Error: " + str(err))

    return P, err
```

Q6.2 [Extra Credit]
Compute the spatio temporal reconstruction of the car for all 10 time instances and plot them by completing the `plot_3d_keypoint_video` function. In your write-up: Plot the spatio-temporal reconstruction of the car for the 10 timesteps. Include the code snippets `plot_3d_keypoint_video` in your write-up.

Spatio-temporal reconstruction of the car for the 10 timesteps:



3D spatio-temporal reconstruction.

```python
def plot_3d_keypoint_video(pts_3d_video):
    # Initialize the figure
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    ax.set_xlabel("X Label")
    ax.set_ylabel("Y Label")
    ax.set_zlabel("Z Label")
    ax.set_title("Spatio-temporal 3D Reconstruction")

    # Loop through the points
    for i in range(len(pts_3d_video)):
        # Extract the points
        pts_3d = pts_3d_video[i]

        # Plot
        for j in range(len(connections_3d)):
            index0, index1 = connections_3d[j]
            xline = [pts_3d[index0, 0], pts_3d[index1, 0]]
            yline = [pts_3d[index0, 1], pts_3d[index1, 1]]
            zline = [pts_3d[index0, 2], pts_3d[index1, 2]]
            ax.plot(xline, yline, zline, color=colors[j])

    plt.show()
```