Q1.1

If both images are going through the origin the 3D point gets projection:

$$p1 = p2 = [0, 0, 1]^T$$

We also know the fundamental matrix allows us to make the following assertion:

$$p1^T * F * p2 = 0$$

Doing matrix multiplication:

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F11 & F12 & F13 \\ F21 & F22 & F23 \\ F31 & F32 & F33 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} F31 & F32 & F33 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

Through inspection of the above expression it becomes clear that the only way for the expression to hold true is for the F33 term to be equal to zero.

Q1.2

There is a pure rotation, so R is an identity matrix. The translation is paralell to the x axis, so the translation vector is t = [tx, 0, 0]^T. The skew symmetric matrix is given as T.

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix} \qquad K = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Since this is a pure translation, the T matrix is equivalent to the essential matrix. So now you can calculate the epipolar lines as the following:

$$\mathrm{L1}^T = \mathrm{X2}^T\mathrm{E} = \begin{bmatrix} x2 & y2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix} = \begin{bmatrix} 0 & tx & -tx * y2 \end{bmatrix}$$

$$\mathrm{L2}^T = \mathrm{x1}^T\mathrm{E} = \begin{bmatrix} x1 & y1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix} = \begin{bmatrix} 0 & -tx & tx * y1 \end{bmatrix}$$

Both of these lines are parallel to the x-axis.

Q1.3

First define the points in the frame of the camera, where w is the point in the real world, t is a translation vector, and R is the rotation matrix.

$$W1 = R1(w) + t1$$

$$W2 = R2(w) + t2$$

Rearranging the first equation allows us to yield the following equation:

$$w = R1^{-1}(W1 - t1)$$

Plugging that into the second equation gives:

$$W2 = R2(R1^{-1}(W1 - t1)) + t2$$

From inspection, the rotation and translation then becomes:

$$R_{relative} = R2*R1^{-1}$$

$$t_{relative} = R2*R1^{-1} * t1 + t2$$

We can then say that:

$$E = t_{relative} \ x \ R_{relative}$$

$$F = (K^{-1})^{T} * E * K^{-1} = (K^{-1})^{T} * t_{relative} \ x \ R_{relative} * K^{-1}$$

Q1.4

Make p a point of the object in the world frame, and it has a reflection we will call p’. So, the 2D projection points for p, p’ will be x1, x2 and x1’, x2’ respectively. We will leverage the property of the fundamental matrix to say:

$$(1)\quad x1^T * F * x2 = 0$$
$$(2)\quad x2' * F^T * x1' = 0$$

We know that both of our points are symmetric to the mirror, so with the equation (2) in mind, it is not far fetched to say that:

$$(3)\quad x1^T * F^T * x2 = 0$$

Add (1) and (3) to find that:

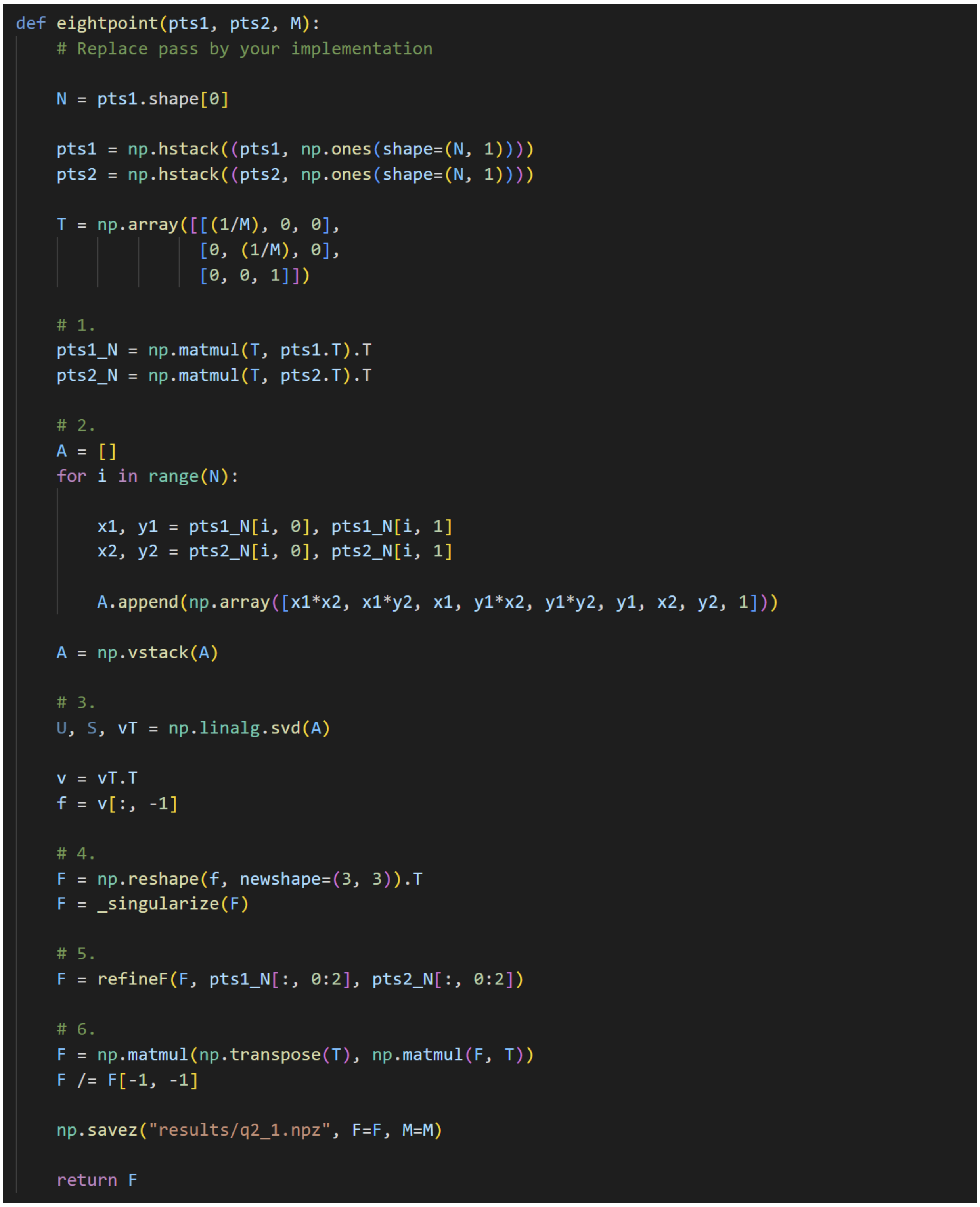
$$x1^T *(F + F^T) * x2 = 0$$

$$F + F^T = 0$$

$$(4)\quad F = -F^T$$

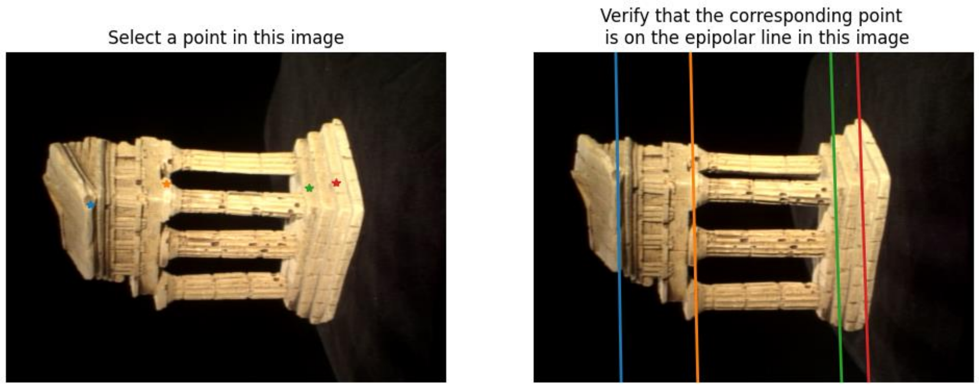
Equation (4) shows us that F is a skew-symmetric matrix.

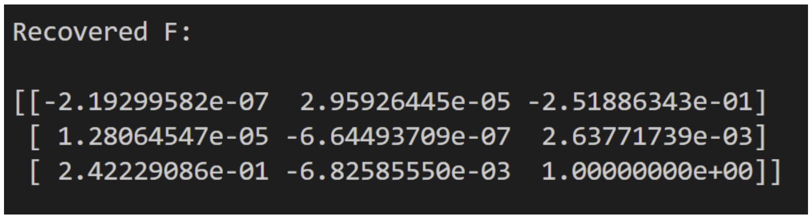
Q2.1: Code



```python
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation

    N = pts1.shape[0]

    pts1 = np.hstack((pts1, np.ones(shape=(N, 1))))
    pts2 = np.hstack((pts2, np.ones(shape=(N, 1))))

    T = np.array([[(1/M), 0, 0],
                  [0, (1/M), 0],
                  [0, 0, 1]])

    # 1.
    pts1_N = np.matmul(T, pts1.T).T
    pts2_N = np.matmul(T, pts2.T).T

    # 2.
    A = []
    for i in range(N):

        x1, y1 = pts1_N[i, 0], pts1_N[i, 1]
        x2, y2 = pts2_N[i, 0], pts2_N[i, 1]

        A.append(np.array([x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]))

    A = np.vstack(A)

    # 3.
    U, S, vT = np.linalg.svd(A)

    v = vT.T
    f = v[:, -1]

    # 4.
    F = np.reshape(f, newshape=(3, 3)).T
    F = _singularize(F)

    # 5.
    F = refineF(F, pts1_N[:, 0:2], pts2_N[:, 0:2])

    # 6.
    F = np.matmul(np.transpose(T), np.matmul(F, T))
    F /= F[-1, -1]

    np.savez("results/q2_1.npz", F=F, M=M)

    return F
```

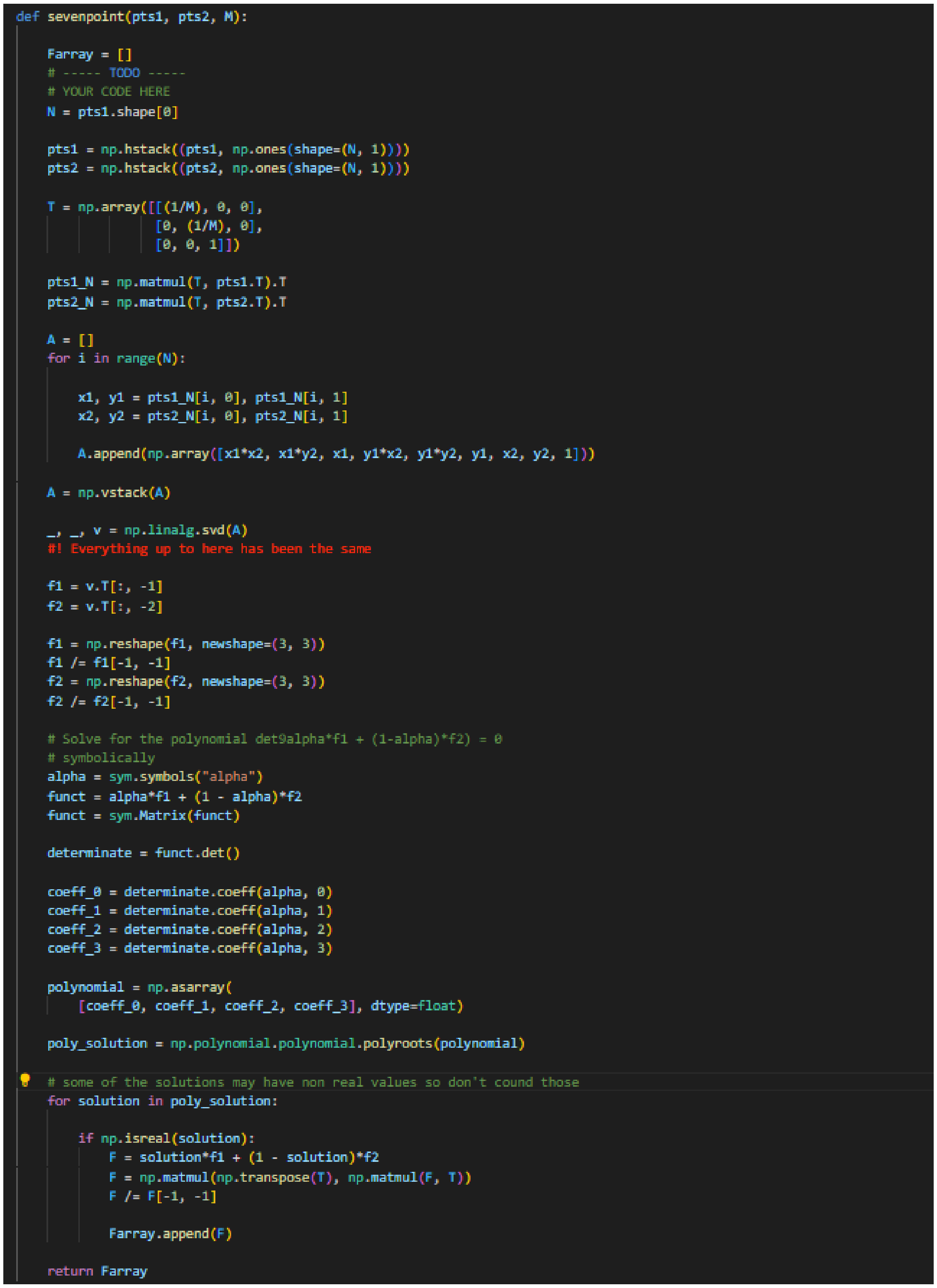
Q2.1: Results


Select a point in this image
Verify that the corresponding point
is on the epipolar line in this image



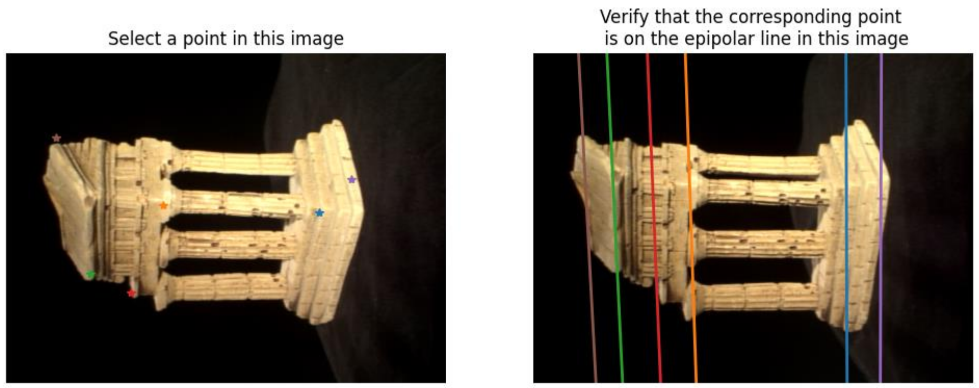
```
Recovered F:

[[-2.19299582e-07  2.95926445e-05 -2.51886343e-01]
 [ 1.28064547e-05 -6.64493709e-07  2.63771739e-03]
 [ 2.42229086e-01 -6.82585550e-03  1.00000000e+00]]
```

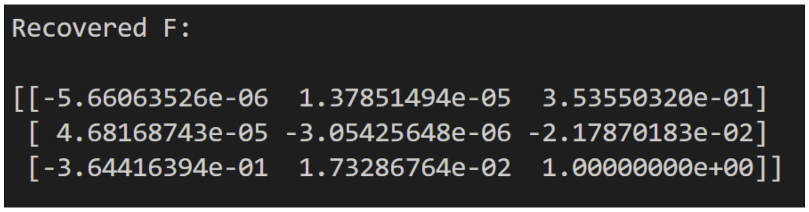
Q2.2: Code



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

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

    pts1 = np.hstack((pts1, np.ones(shape=(N, 1))))
    pts2 = np.hstack((pts2, np.ones(shape=(N, 1))))

    T = np.array([[(1/M), 0, 0],
                  [0, (1/M), 0],
                  [0, 0, 1]])

    pts1_N = np.matmul(T, pts1.T).T
    pts2_N = np.matmul(T, pts2.T).T

    A = []
    for i in range(N):

        x1, y1 = pts1_N[i, 0], pts1_N[i, 1]
        x2, y2 = pts2_N[i, 0], pts2_N[i, 1]

        A.append(np.array([x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]))

    A = np.vstack(A)

    _, _, v = np.linalg.svd(A)
    #! Everything up to here has been the same

    f1 = v.T[:, -1]
    f2 = v.T[:, -2]

    f1 = np.reshape(f1, newshape=(3, 3))
    f1 /= f1[-1, -1]
    f2 = np.reshape(f2, newshape=(3, 3))
    f2 /= f2[-1, -1]

    # Solve for the polynomial det9alpha*f1 + (1-alpha)*f2) = 0
    # symbolically
    alpha = sym.symbols("alpha")
    funct = alpha*f1 + (1 - alpha)*f2
    funct = sym.Matrix(funct)

    determinate = funct.det()

    coeff_0 = determinate.coeff(alpha, 0)
    coeff_1 = determinate.coeff(alpha, 1)
    coeff_2 = determinate.coeff(alpha, 2)
    coeff_3 = determinate.coeff(alpha, 3)

    polynomial = np.asarray(
        [coeff_0, coeff_1, coeff_2, coeff_3], dtype=float)

    poly_solution = np.polynomial.polynomial.polyroots(polynomial)

    # some of the solutions may have non real values so don't cound those
    for solution in poly_solution:

        if np.isreal(solution):
            F = solution*f1 + (1 - solution)*f2
            F = np.matmul(np.transpose(T), np.matmul(F, T))
            F /= F[-1, -1]

            Farray.append(F)

    return Farray
```

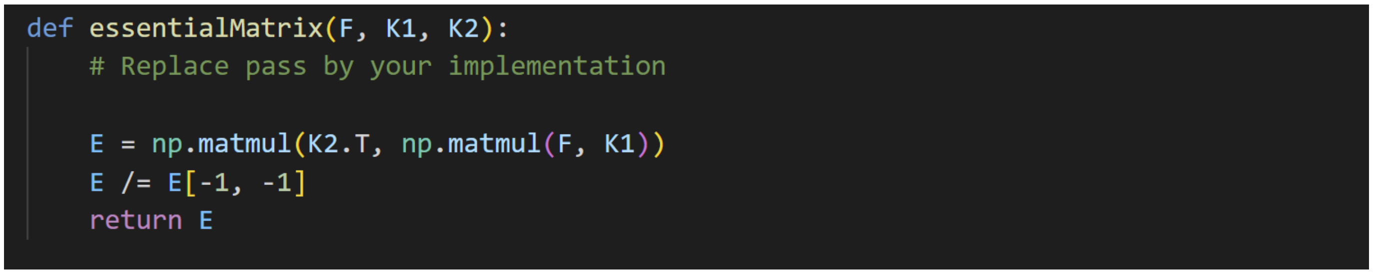
Q2.2: Results


Select a point in this image
Verify that the corresponding point is on the epipolar line in this image



```
Recovered F:

[[-5.66063526e-06  1.37851494e-05  3.53550320e-01]
 [ 4.68168743e-05 -3.05425648e-06 -2.17870183e-02]
 [-3.64416394e-01  1.73286764e-02  1.00000000e+00]]
```

Q3.1



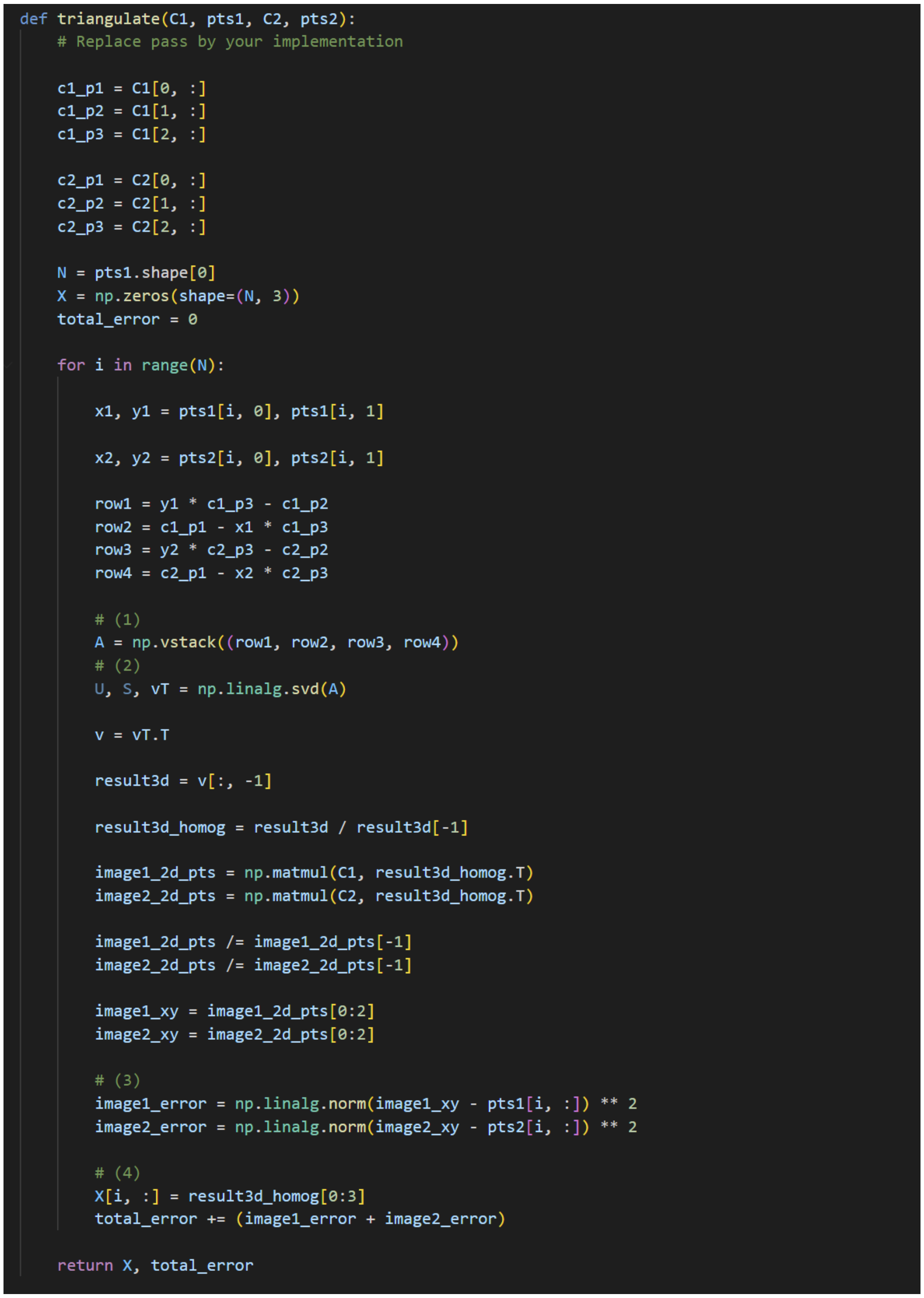
```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation

    E = np.matmul(K2.T, np.matmul(F, K1))
    E /= E[-1, -1]
    return E
```

Q3.2: A-Matrix

$$\begin{bmatrix} y p_3^T - p_2^T \\ p_1^T - x p_3^T \\ y' p'^T_3 - p'^T_2 \\ p'^T_1 - x' p'^T_3 \end{bmatrix}$$

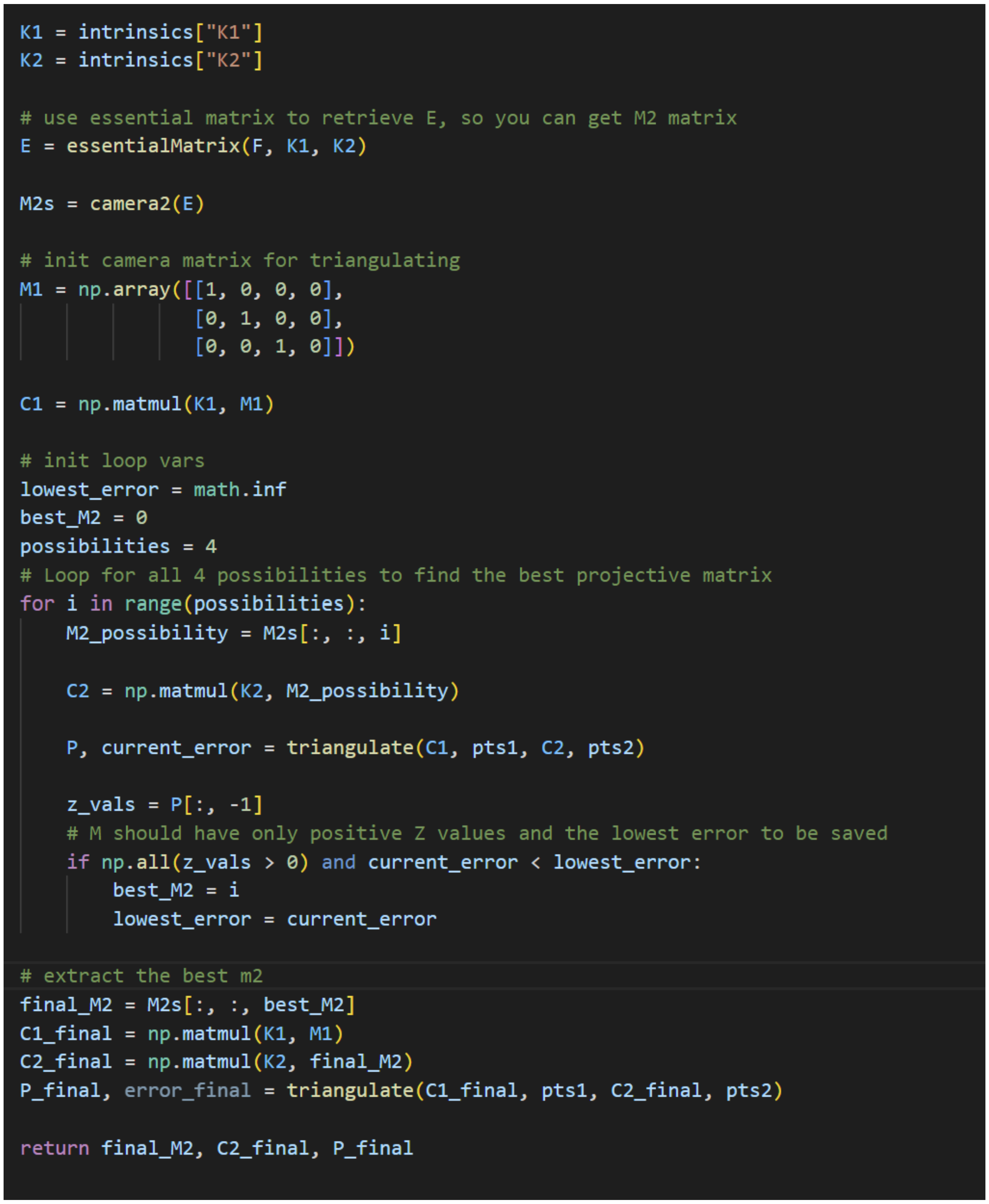
Where x, y values come from points in image 1 and x', y' come from points in image 2, and p1^T, p2^T, p3^T are the transposed rows of the projection matrix of camera one and the p1'^T, p2'^T, p3'^T are the transposed rows of the projection matrix of camera matrix two. p1^T, p2^T, p3^T.

Q3.2: Code



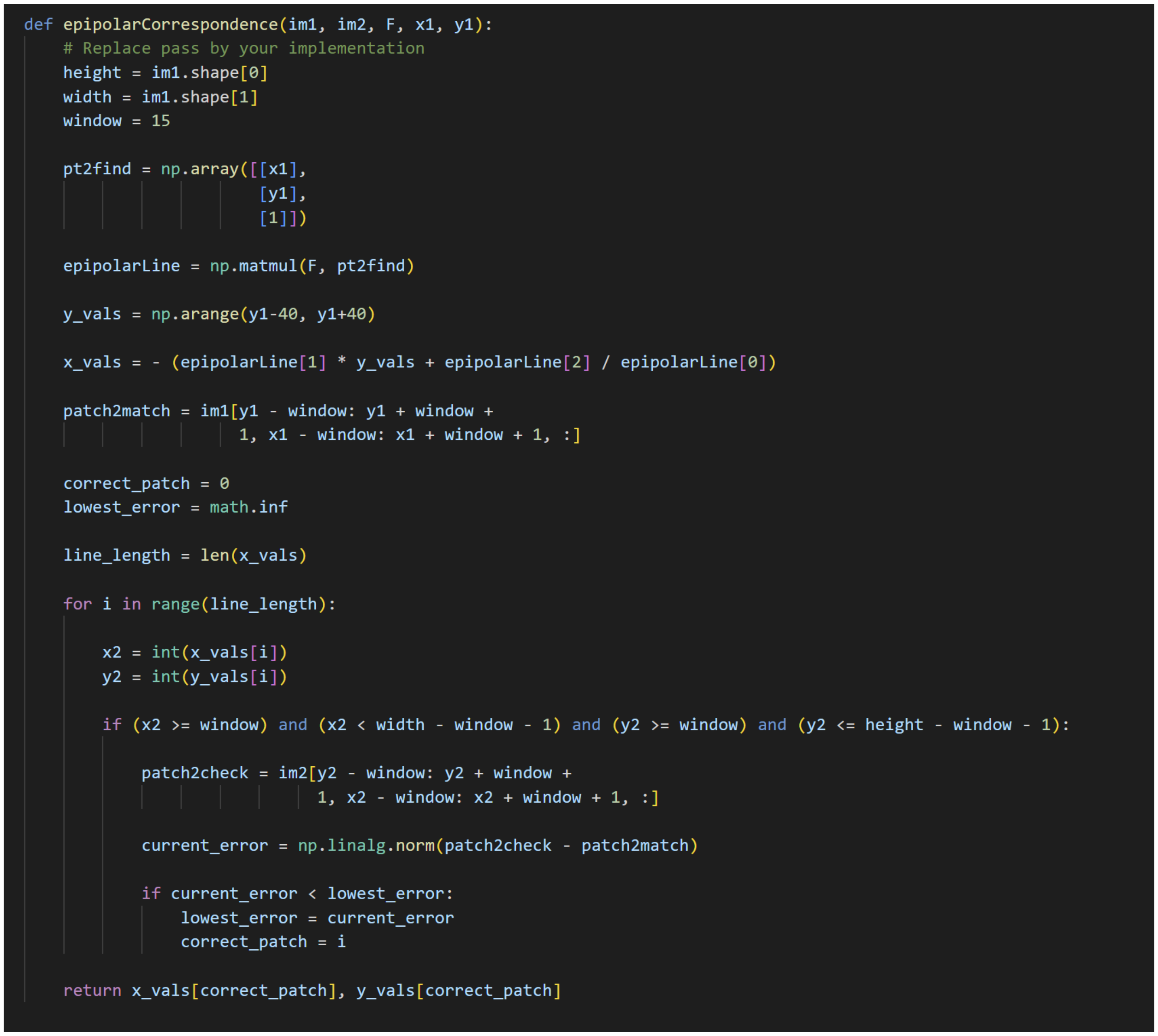
```python
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation

    c1_p1 = C1[0, :]
    c1_p2 = C1[1, :]
    c1_p3 = C1[2, :]

    c2_p1 = C2[0, :]
    c2_p2 = C2[1, :]
    c2_p3 = C2[2, :]

    N = pts1.shape[0]
    X = np.zeros(shape=(N, 3))
    total_error = 0

    for i in range(N):

        x1, y1 = pts1[i, 0], pts1[i, 1]

        x2, y2 = pts2[i, 0], pts2[i, 1]

        row1 = y1 * c1_p3 - c1_p2
        row2 = c1_p1 - x1 * c1_p3
        row3 = y2 * c2_p3 - c2_p2
        row4 = c2_p1 - x2 * c2_p3

        # (1)
        A = np.vstack((row1, row2, row3, row4))
        # (2)
        U, S, vT = np.linalg.svd(A)

        v = vT.T

        result3d = v[:, -1]

        result3d_homog = result3d / result3d[-1]

        image1_2d_pts = np.matmul(C1, result3d_homog.T)
        image2_2d_pts = np.matmul(C2, result3d_homog.T)

        image1_2d_pts /= image1_2d_pts[-1]
        image2_2d_pts /= image2_2d_pts[-1]

        image1_xy = image1_2d_pts[0:2]
        image2_xy = image2_2d_pts[0:2]

        # (3)
        image1_error = np.linalg.norm(image1_xy - pts1[i, :]) ** 2
        image2_error = np.linalg.norm(image2_xy - pts2[i, :]) ** 2

        # (4)
        X[i, :] = result3d_homog[0:3]
        total_error += (image1_error + image2_error)

    return X, total_error
```

Q3.3



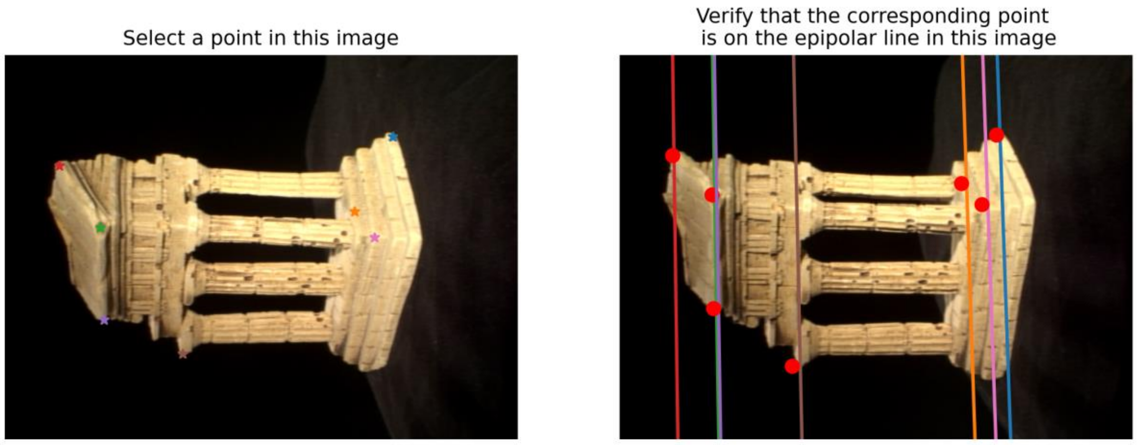
```python
K1 = intrinsics["K1"]
K2 = intrinsics["K2"]

# use essential matrix to retrieve E, so you can get M2 matrix
E = essentialMatrix(F, K1, K2)

M2s = camera2(E)

# init camera matrix for triangulating
M1 = np.array([[1, 0, 0, 0],
               [0, 1, 0, 0],
               [0, 0, 1, 0]])

C1 = np.matmul(K1, M1)

# init loop vars
lowest_error = math.inf
best_M2 = 0
possibilities = 4
# Loop for all 4 possibilities to find the best projective matrix
for i in range(possibilities):
    M2_possibility = M2s[:, :, i]

    C2 = np.matmul(K2, M2_possibility)

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

    z_vals = P[:, -1]
    # M should have only positive Z values and the lowest error to be saved
    if np.all(z_vals > 0) and current_error < lowest_error:
        best_M2 = i
        lowest_error = current_error

# extract the best m2
final_M2 = M2s[:, :, best_M2]
C1_final = np.matmul(K1, M1)
C2_final = np.matmul(K2, final_M2)
P_final, error_final = triangulate(C1_final, pts1, C2_final, pts2)

return final_M2, C2_final, P_final
```

Q4.1

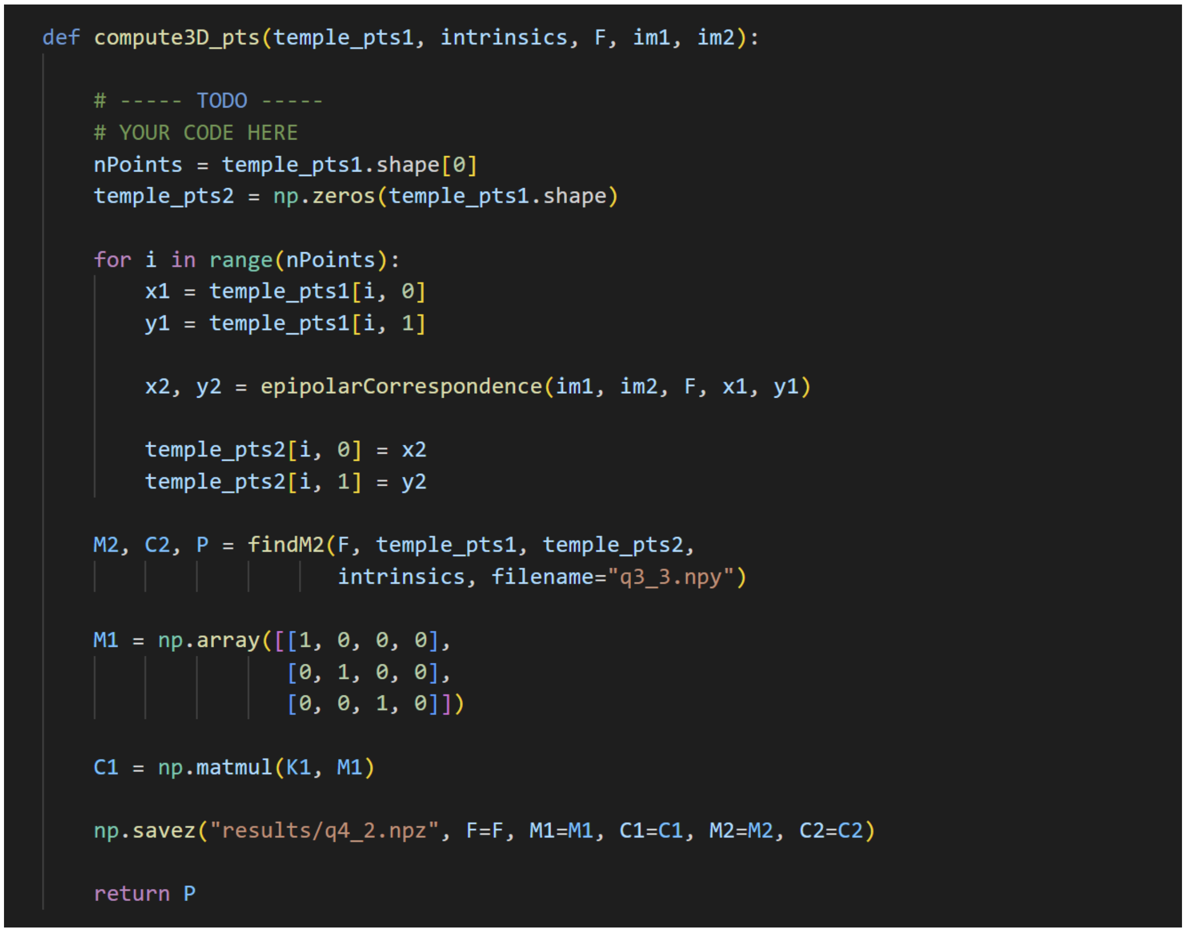


```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Replace pass by your implementation
    height = im1.shape[0]
    width = im1.shape[1]
    window = 15

    pt2find = np.array([[x1],
                        [y1],
                        [1]])

    epipolarLine = np.matmul(F, pt2find)

    y_vals = np.arange(y1-40, y1+40)

    x_vals = - (epipolarLine[1] * y_vals + epipolarLine[2] / epipolarLine[0])

    patch2match = im1[y1 - window: y1 + window +
                      1, x1 - window: x1 + window + 1, :]

    correct_patch = 0
    lowest_error = math.inf

    line_length = len(x_vals)

    for i in range(line_length):

        x2 = int(x_vals[i])
        y2 = int(y_vals[i])

        if (x2 >= window) and (x2 < width - window - 1) and (y2 >= window) and (y2 <= height - window - 1):

            patch2check = im2[y2 - window: y2 + window +
                              1, x2 - window: x2 + window + 1, :]

            current_error = np.linalg.norm(patch2check - patch2match)

            if current_error < lowest_error:
                lowest_error = current_error
                correct_patch = i

    return x_vals[correct_patch], y_vals[correct_patch]
```

Q4.1: Results

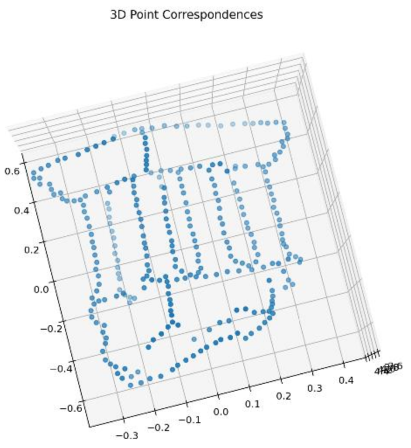

Select a point in this image
Verify that the corresponding point
is on the epipolar line in this image

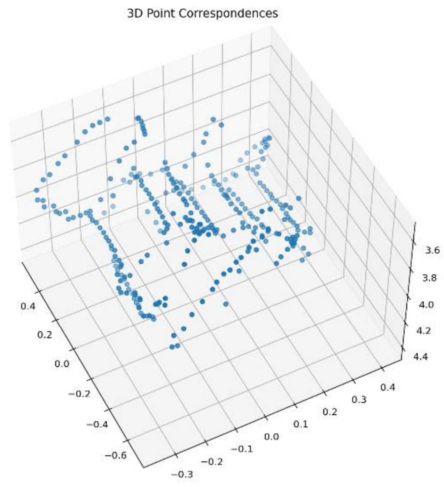
Q4.2: Code

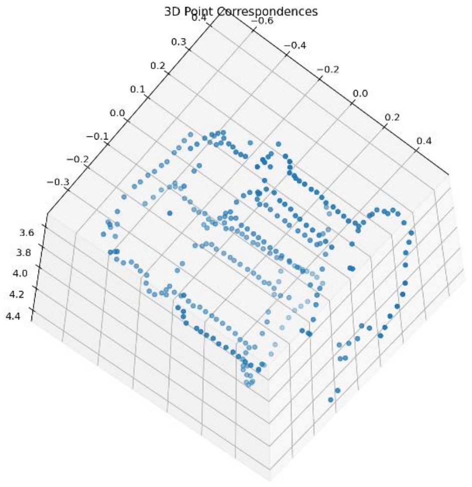


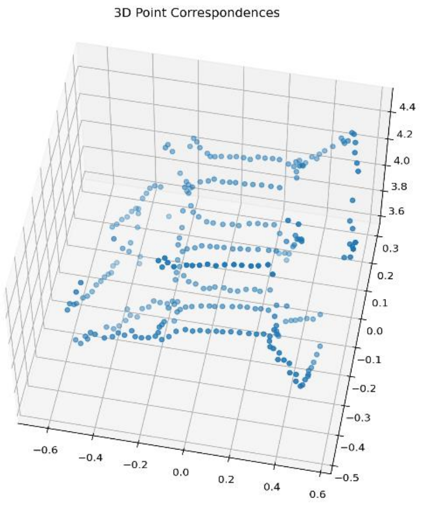
```python
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):

    # ----- TODO -----
    # YOUR CODE HERE
    nPoints = temple_pts1.shape[0]
    temple_pts2 = np.zeros(temple_pts1.shape)

    for i in range(nPoints):
        x1 = temple_pts1[i, 0]
        y1 = temple_pts1[i, 1]

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

        temple_pts2[i, 0] = x2
        temple_pts2[i, 1] = y2

    M2, C2, P = findM2(F, temple_pts1, temple_pts2,
                       intrinsics, filename="q3_3.npy")

    M1 = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0]])

    C1 = np.matmul(K1, M1)

    np.savez("results/q4_2.npz", F=F, M1=M1, C1=C1, M2=M2, C2=C2)

    return P
```

Q4.2: Results

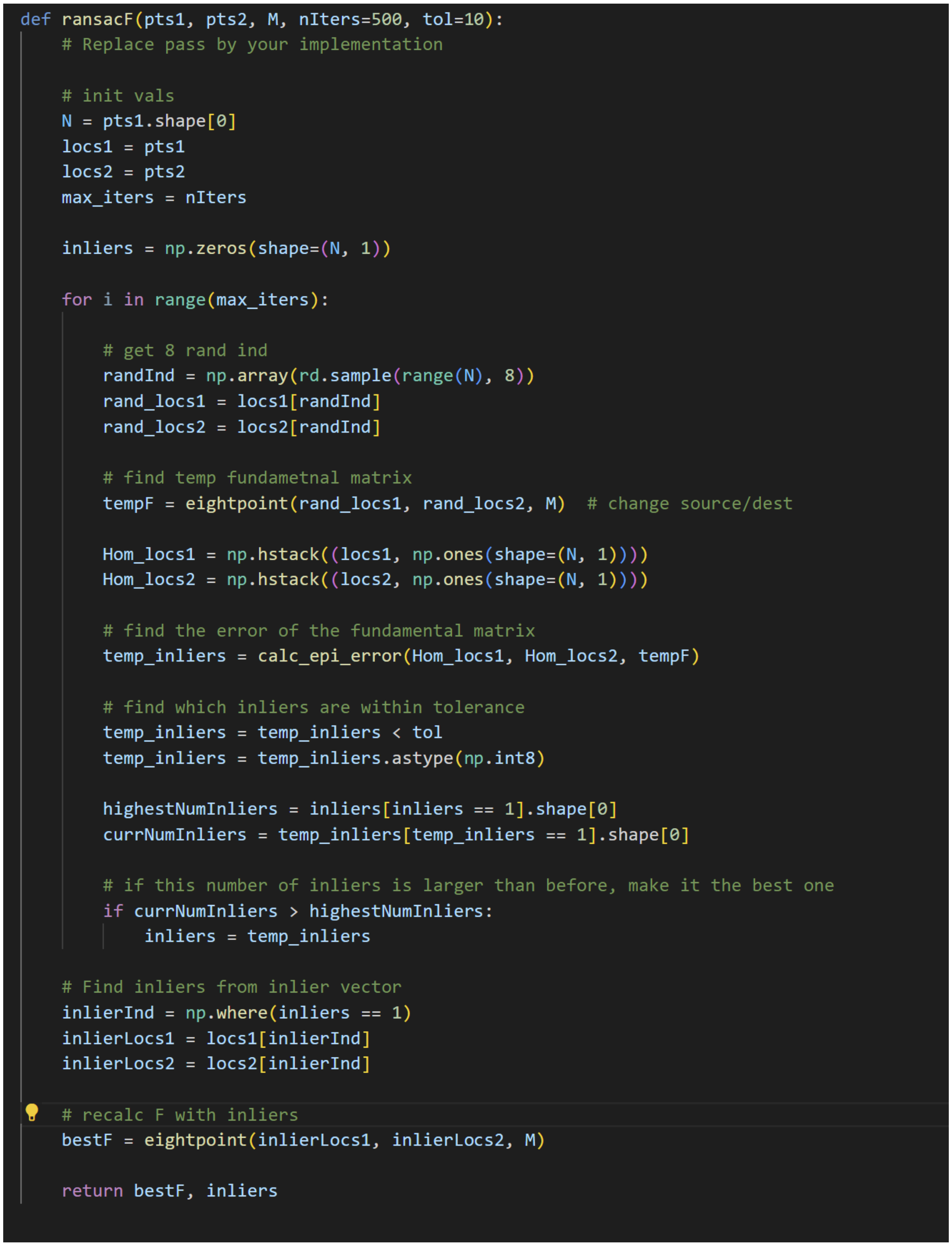

3D Point Correspondences
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4


3D Point Correspondences
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
3.6
3.8
4.0
4.2
4.4


3D Point Correspondences
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.6
−0.4
−0.2
0.0
0.2
0.4
3.6
3.8
4.0
4.2
4.4


3D Point Correspondences
4.4
4.2
4.0
3.8
3.6
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6

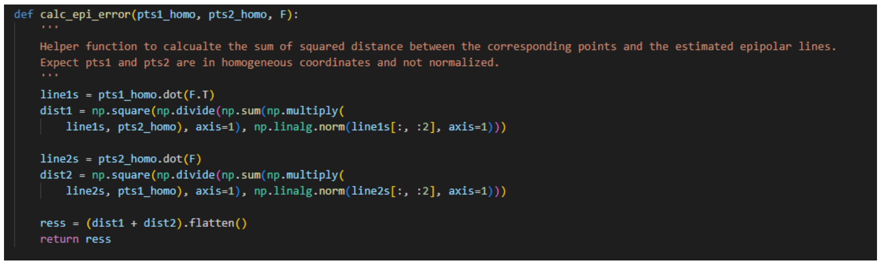
Q5.1: Code



```python
def ransacF(pts1, pts2, M, nIters=500, tol=10):
    # Replace pass by your implementation

    # init vals
    N = pts1.shape[0]
    locs1 = pts1
    locs2 = pts2
    max_iters = nIters

    inliers = np.zeros(shape=(N, 1))

    for i in range(max_iters):

        # get 8 rand ind
        randInd = np.array(rd.sample(range(N), 8))
        rand_locs1 = locs1[randInd]
        rand_locs2 = locs2[randInd]

        # find temp fundametnal matrix
        tempF = eightpoint(rand_locs1, rand_locs2, M)  # change source/dest

        Hom_locs1 = np.hstack((locs1, np.ones(shape=(N, 1))))
        Hom_locs2 = np.hstack((locs2, np.ones(shape=(N, 1))))

        # find the error of the fundamental matrix
        temp_inliers = calc_epi_error(Hom_locs1, Hom_locs2, tempF)

        # find which inliers are within tolerance
        temp_inliers = temp_inliers < tol
        temp_inliers = temp_inliers.astype(np.int8)

        highestNumInliers = inliers[inliers == 1].shape[0]
        currNumInliers = temp_inliers[temp_inliers == 1].shape[0]

        # if this number of inliers is larger than before, make it the best one
        if currNumInliers > highestNumInliers:
            inliers = temp_inliers

    # Find inliers from inlier vector
    inlierInd = np.where(inliers == 1)
    inlierLocs1 = locs1[inlierInd]
    inlierLocs2 = locs2[inlierInd]

    # recalc F with inliers
    bestF = eightpoint(inlierLocs1, inlierLocs2, M)

    return bestF, inliers
```

Q5.1: RANSAC Discussion

Error Metrics



```python
def calc_epi_error(pts1_homo, pts2_homo, F):
    '''
    Helper function to calcualte the sum of squared distance between the corresponding points and the estimated epipolar lines.
    Expect pts1 and pts2 are in homogeneous coordinates and not normalized.
    '''
    line1s = pts1_homo.dot(F.T)
    dist1 = np.square(np.divide(np.sum(np.multiply(
        line1s, pts2_homo), axis=1), np.linalg.norm(line1s[:, :2], axis=1)))

    line2s = pts2_homo.dot(F)
    dist2 = np.square(np.divide(np.sum(np.multiply(
        line2s, pts1_homo), axis=1), np.linalg.norm(line2s[:, :2], axis=1)))

    ress = (dist1 + dist2).flatten()
    return ress
```

The error metric used was to basically use the distance how far each point was for each corresponding epipolar line. The sum of this squared distance was used to generate the error for the respective fundamental matrix. This metric was given from helper.py.

Explanation of RANSAC Algorithm

The RANSAC algorithm first starting by taking 8 random points from the potential points to given to be able to compute a fundamental matrix using the eightpoint algorithm. I chose the eightpoint algorithm for consistency and there was no clear advantage to chosing the seven-point algorithm. Then, using the error metric described above, found the sum of the squared distance to the epipolar lines for each point. If this distance was below the tolerance that was predetermined, then the point was considered an inlier. If this fundamental matrix yielded the most inliers up to this point it was considered the best F and its inliers where saved. This process was repeated for a set number of iterations, and after it was completed with those iteraions the Fundamental matrix was recomputed using only the inliers.

Q5.1: RANSAC Discussion

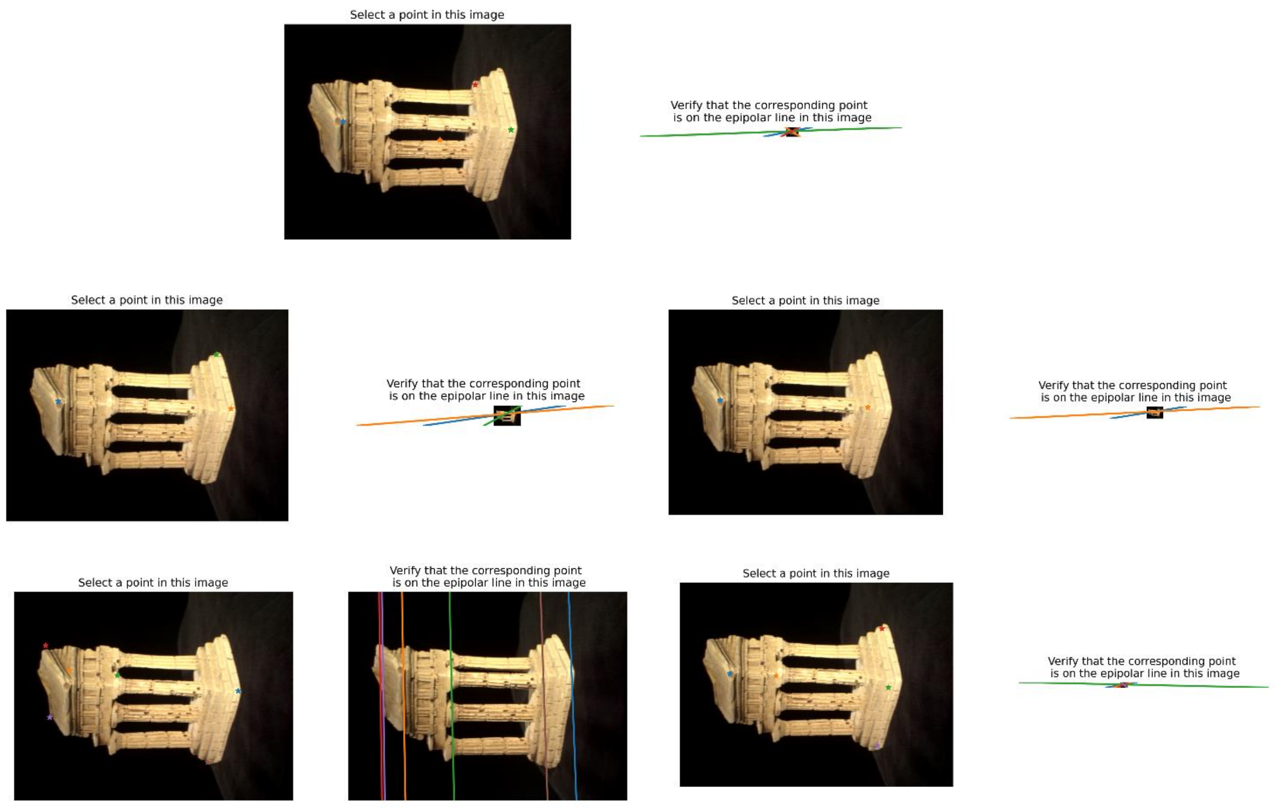
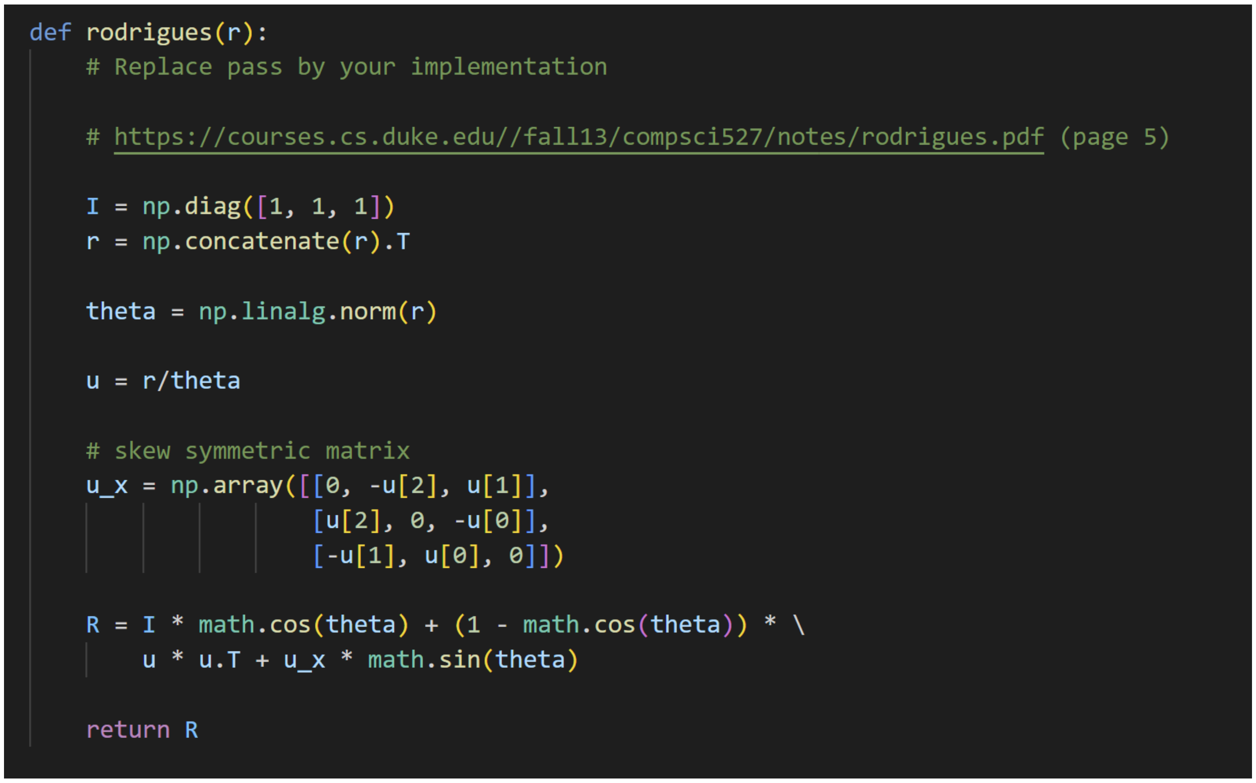

Select a point in this image
Verify that the corresponding point is on the epipolar line in this image
Select a point in this image
Verify that the corresponding point is on the epipolar line in this image
Select a point in this image
Verify that the corresponding point is on the epipolar line in this image
Select a point in this image
Verify that the corresponding point is on the epipolar line in this image
Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Fig. *Displayed epipolar lines for: no RANSAC on F (top), 100 iterations & tolerance of 10 (middle left), 500 iterations & tolerance of 10 (middle left), 500 iterations & tolerance of 5 (top right), 500 iterations & tolerance of 1 (bottom right),*

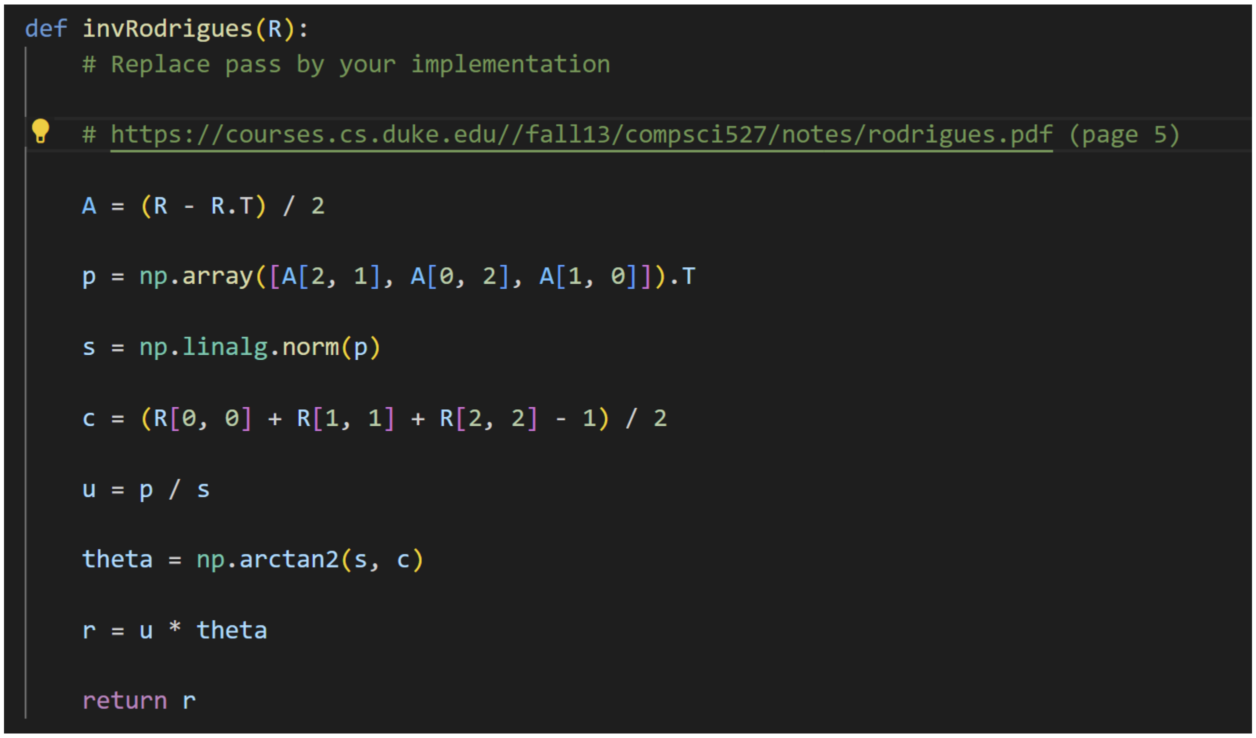
Comments:

Visually, what I was seeing was that there were a certain threshold for bother iterations and tolerance that the RANSAC was able to perform well for. Here, it seemed that the data was fairly noisy, so finding 8 points that were representative the correct number of inliers proved too difficult to do in any number of inliers below 500. Also, tolerance proved to be important as well: if the tolerance was too low RANSAC was filtering out too many points to represent the transformation properly. The best performing iteration I had was for 500 iterations and a tolerance of 10, and for those settings it seemed that the rough number of inliers I was obtaining was 110, meaning that roughly 75% of the noisy data was retained – exactly what we desired.

Q5.2

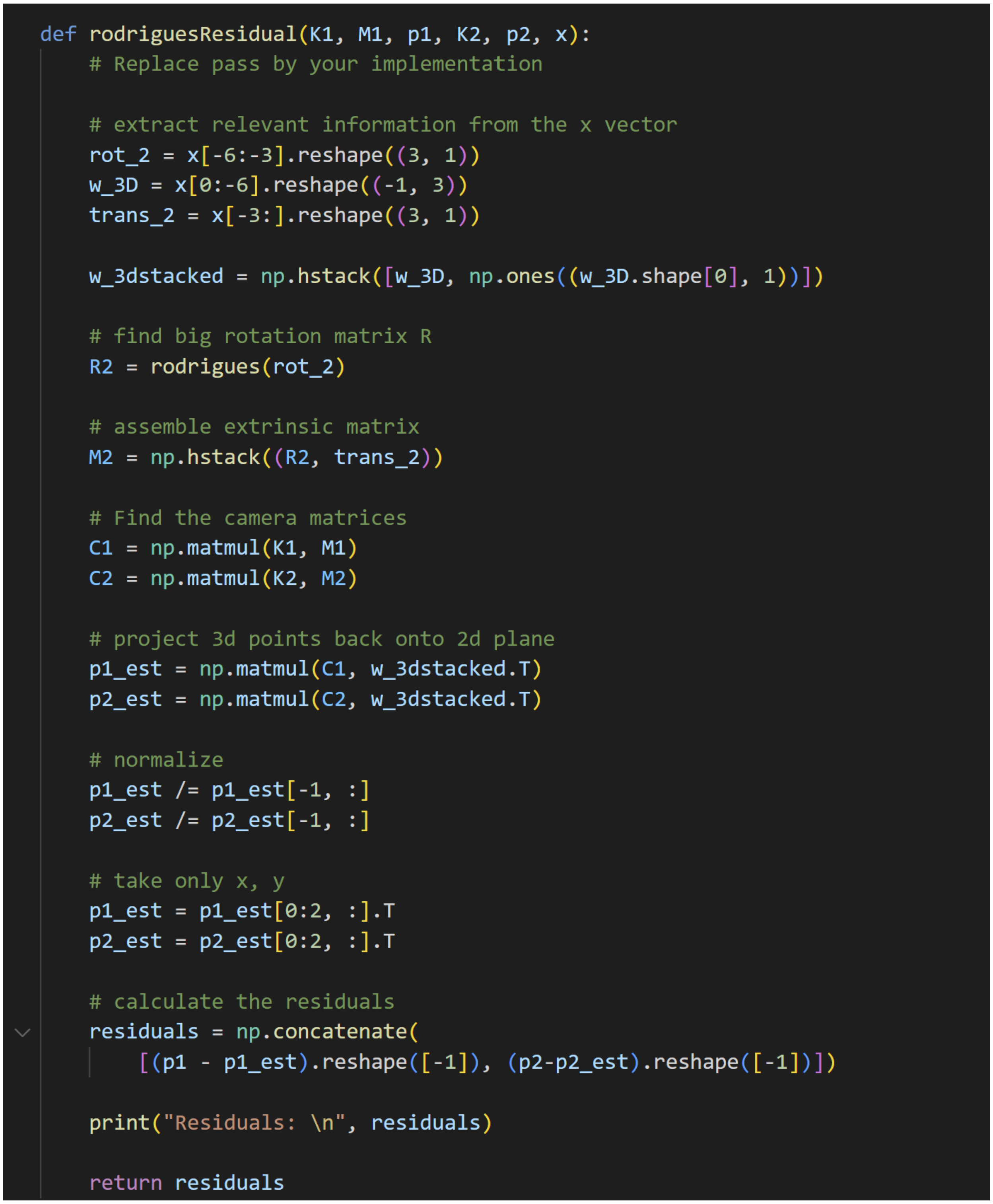


```python
def rodrigues(r):
    # Replace pass by your implementation

    # https://courses.cs.duke.edu//fall13/compsci527/notes/rodrigues.pdf (page 5)

    I = np.diag([1, 1, 1])
    r = np.concatenate(r).T

    theta = np.linalg.norm(r)

    u = r/theta

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

    R = I * math.cos(theta) + (1 - math.cos(theta)) * \
        u * u.T + u_x * math.sin(theta)

    return R
```



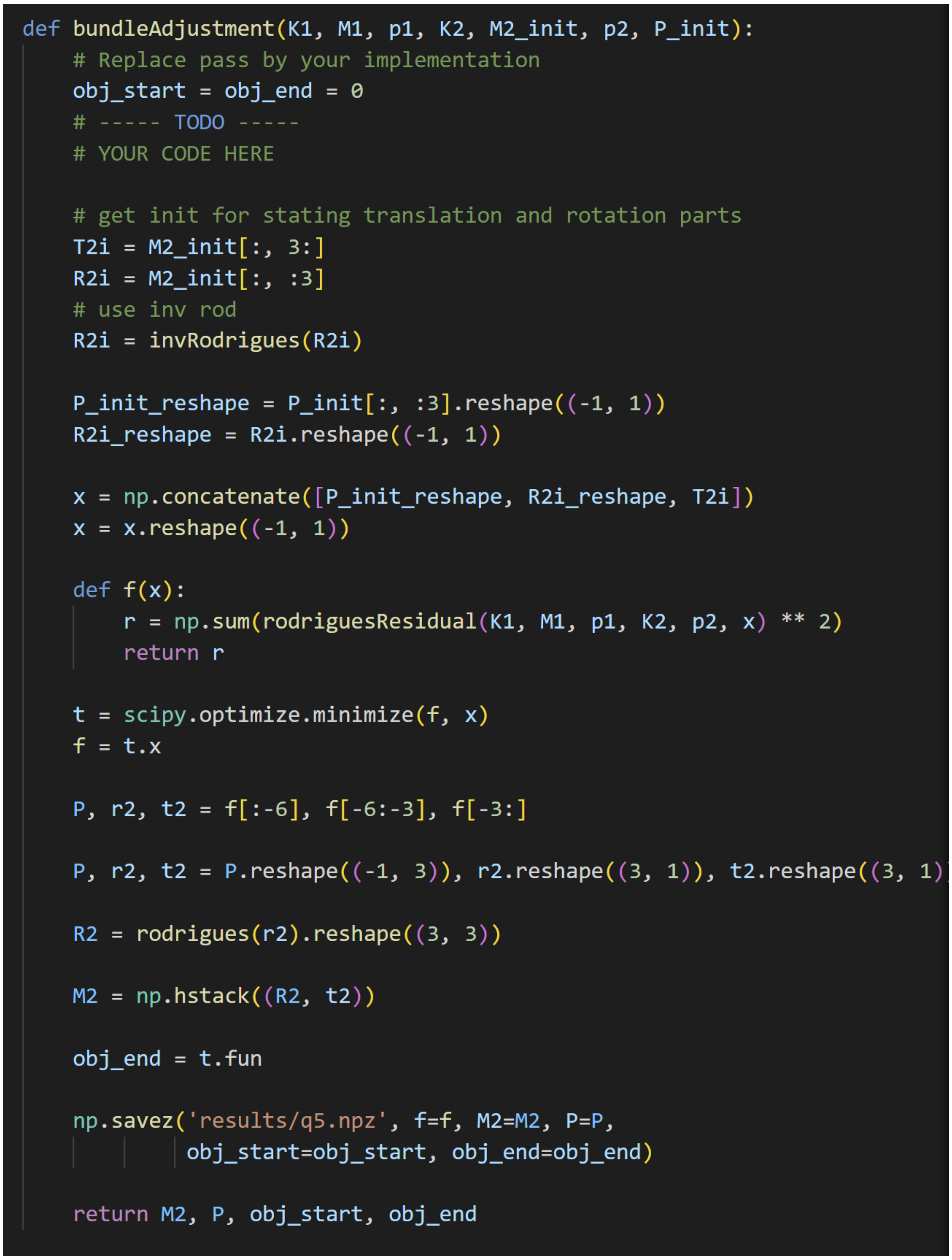
```python
def invRodrigues(R):
    # Replace pass by your implementation

    # https://courses.cs.duke.edu//fall13/compsci527/notes/rodrigues.pdf (page 5)

    A = (R - R.T) / 2

    p = np.array([A[2, 1], A[0, 2], A[1, 0]]).T

    s = np.linalg.norm(p)

    c = (R[0, 0] + R[1, 1] + R[2, 2] - 1) / 2

    u = p / s

    theta = np.arctan2(s, c)

    r = u * theta

    return r
```

Q5.3: Code



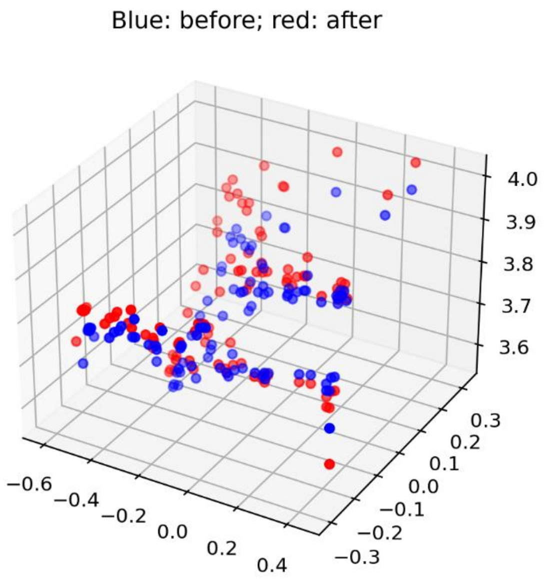
```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation

    # extract relevant information from the x vector
    rot_2 = x[-6:-3].reshape((3, 1))
    w_3D = x[0:-6].reshape((-1, 3))
    trans_2 = x[-3:].reshape((3, 1))

    w_3dstacked = np.hstack([w_3D, np.ones((w_3D.shape[0], 1))])

    # find big rotation matrix R
    R2 = rodrigues(rot_2)

    # assemble extrinsic matrix
    M2 = np.hstack((R2, trans_2))

    # Find the camera matrices
    C1 = np.matmul(K1, M1)
    C2 = np.matmul(K2, M2)

    # project 3d points back onto 2d plane
    p1_est = np.matmul(C1, w_3dstacked.T)
    p2_est = np.matmul(C2, w_3dstacked.T)

    # normalize
    p1_est /= p1_est[-1, :]
    p2_est /= p2_est[-1, :]

    # take only x, y
    p1_est = p1_est[0:2, :].T
    p2_est = p2_est[0:2, :].T

    # calculate the residuals
    residuals = np.concatenate(
        [(p1 - p1_est).reshape([-1]), (p2-p2_est).reshape([-1])])

    print("Residuals: \n", residuals)

    return residuals
```

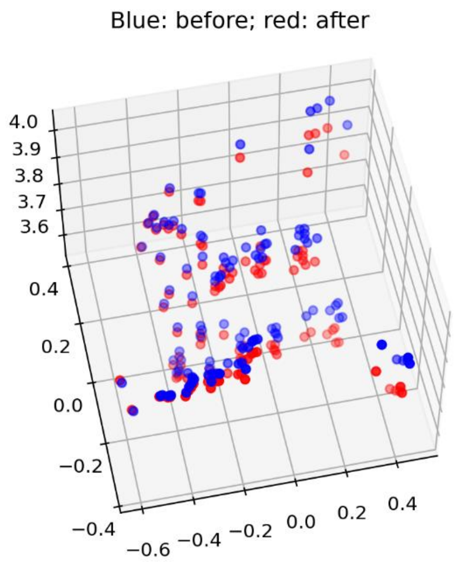
Q5.3: Code

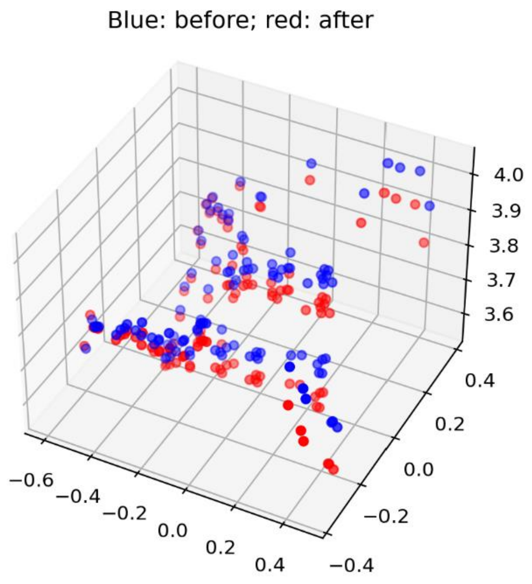


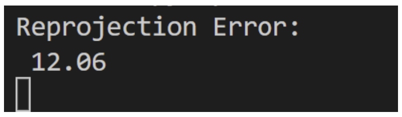
```python
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    # Replace pass by your implementation
    obj_start = obj_end = 0
    # ----- TODO -----
    # YOUR CODE HERE

    # get init for stating translation and rotation parts
    T2i = M2_init[:, 3:]
    R2i = M2_init[:, :3]
    # use inv rod
    R2i = invRodrigues(R2i)

    P_init_reshape = P_init[:, :3].reshape((-1, 1))
    R2i_reshape = R2i.reshape((-1, 1))

    x = np.concatenate([P_init_reshape, R2i_reshape, T2i])
    x = x.reshape((-1, 1))

    def f(x):
        r = np.sum(rodriguesResidual(K1, M1, p1, K2, p2, x) ** 2)
        return r

    t = scipy.optimize.minimize(f, x)
    f = t.x

    P, r2, t2 = f[:-6], f[-6:-3], f[-3:]

    P, r2, t2 = P.reshape((-1, 3)), r2.reshape((3, 1)), t2.reshape((3, 1)

    R2 = rodrigues(r2).reshape((3, 3))

    M2 = np.hstack((R2, t2))

    obj_end = t.fun

    np.savez('results/q5.npz', f=f, M2=M2, P=P,
             obj_start=obj_start, obj_end=obj_end)

    return M2, P, obj_start, obj_end
```

Q5.3: Results









```
Reprojection Error:
 12.06
```