Theory 1.1

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 0. We can see that through the mathematical proof below:

$$F = \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix}$$

$$P_1^T F P_2 = 0$$

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

$$\begin{bmatrix} F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

$$F_{33} = 0$$

Theory 1.2

We can define the translation matrix as such since the translation is parallel to the x-axis

$$t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix}$$

The skew symmetric t can then be written as:

$$t_{\times} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

The rotation matrix can be written as the identity matrix since it's a pure rotation

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then, the essential matrix can be written as

$$E = t_{\times} R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

Therefore, the epipolar lines are:

$$x_2^T E = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & t_x & -t_x y_2 \end{bmatrix}$$

$$x_1^T E = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & -t_x & t_x y_1 \end{bmatrix}$$

The equations of both these two lines are parallel to the x-axis

Theory 1.3

Assuming w is the 3D point in the real world, then the corresponding points at two timestamps are:

$$w_1 = R_1 w + t_1$$
$$w_2 = R_2 w + t_2$$

From that we can calculate the relative rotation and translation between the frames:

$$w = R_1^{-1}(\omega_1 - t_1)$$

$$\begin{aligned} w_2 &= R_2 R_1^{-1}(w_1 - t_1) + t_2 \\ &= R_2 R_1^{-1} w_1 - R_2 R_1^{-1} + t_2 \end{aligned}$$

$$R_{re1} = R_2 R_1^{-1}$$
$$t_{re1} = -R_2 R_1^{-1} t_1 + t_2$$

Then, E and F are shown below

$$E = t_{re1\times} R_{re1}$$
$$E = K^{-1} F K$$
$$F = KEK^{-1} = K\, t_{re1\times}\, R_{re1} K^{-1}$$

Theory 1.4

Assume that there is a point P in 3D of the object and its reflection in the mirror is P'. Assume the 2D coordinates on image 1 and image2 of P are x1 and x2 respectively. Similarly, assume the 2D coordinates on image1 and image2 of P' are x1' and x2' respectively.

Based on the properties of F, we have:

$$x_1^T F x_2 = 0$$
$$x_2' F^T x_1' = 0$$

Through the fact that the points are symmetric through the mirror, we can derive that:

$$x_1^T F^T x_2 = 0$$

Adding the above equations together gives us:

$$x_1^T \left(F + F^T\right) x_2 = 0$$
$$F + F^T = 0$$
$$F = -F^T$$

From the final equation, we can see that F is a skew-symmetric matrix.

2.1 The Eight Point Algorithm

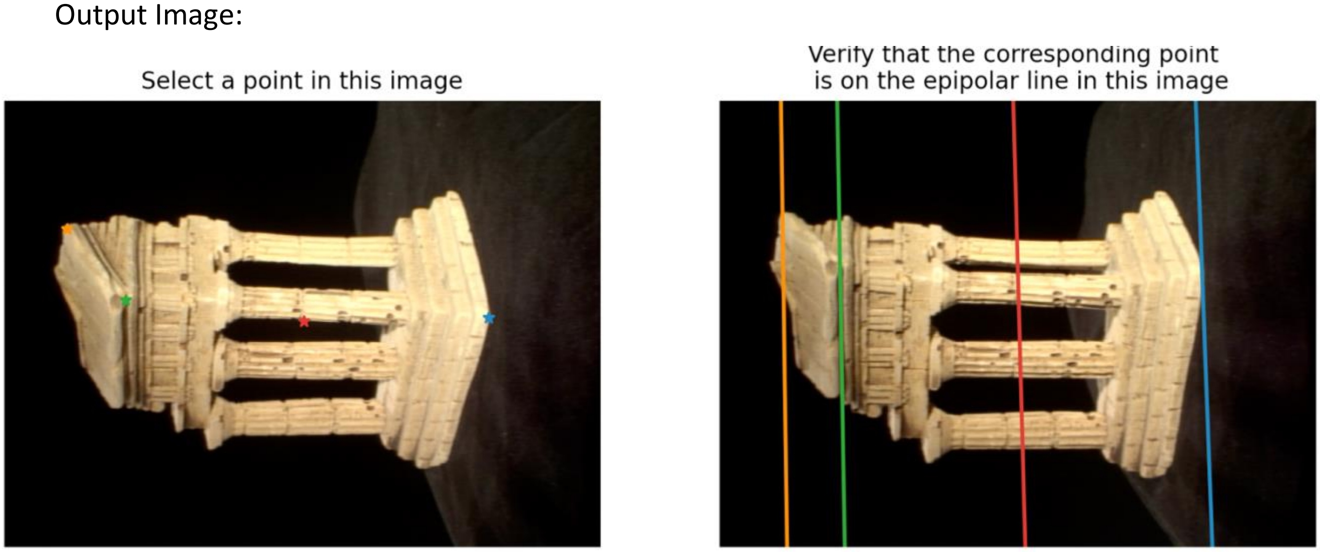
Code Snippet:

```python
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation
    T = np.diag([1/M,1/M,1])
    N = pts1.shape[0]
    ones_arr = np.ones((N,1))
    pts1 = np.hstack((pts1,ones_arr))
    pts2 = np.hstack((pts2,ones_arr))
    norm1 = np.matmul(T,pts1.T).T
    norm2 = np.matmul(T,pts2.T).T

    # compute A
    A = np.ones((N,9))
    for i in range(N):
        x1 = norm1[i,0]
        x2 = norm2[i,0]
        y1 = norm1[i,1]
        y2 = norm2[i,1]
        A[i,:] = [x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]

    # SVD
    D, V = np.linalg.eig(np.dot(A.T, A))
    idx = np.argmin(D)
    F = np.reshape(V[:, idx], (3, 3)).T

    # singularize and refine
    F = _singularize(F)
    F = refineF(F,norm1[:,0:2],norm2[:,0:2])

    # denormalize
    F = np.matmul(T.T,np.matmul(F, T))

    # scale
    F = F / F[2,2]

    return F
```

F:

```
[[-2.19906944e-07  2.95853591e-05 -2.51889048e-01]
 [ 1.28145764e-05 -6.64464480e-07  2.63663152e-03]
 [ 2.42232038e-01 -6.82502127e-03  1.00000000e+00]]
```

Output Image:



2.2 The Seven Point Algorithm

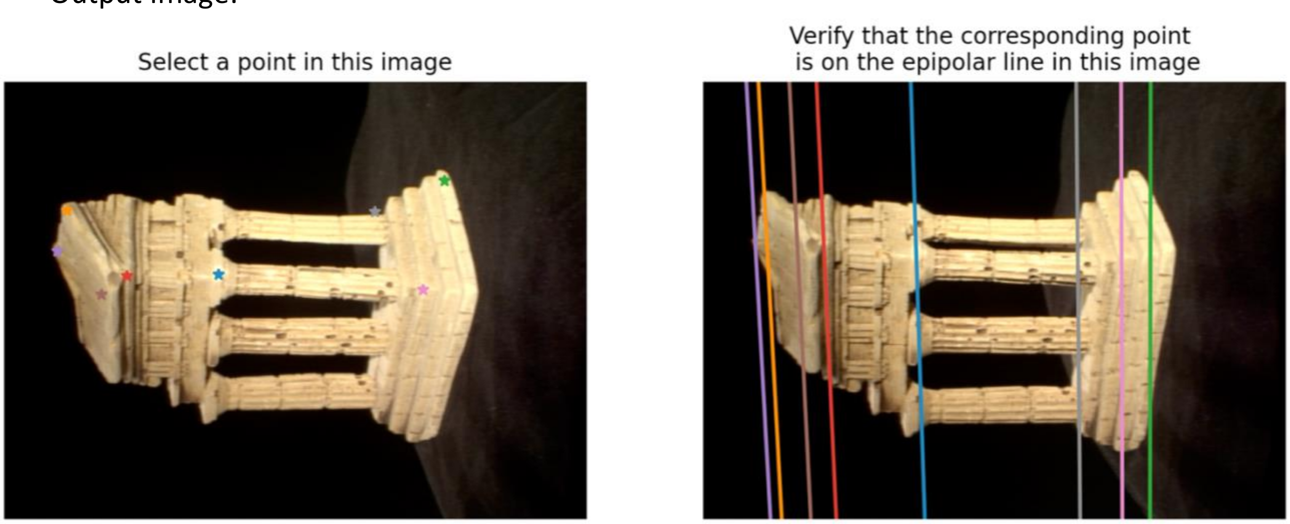
Code Snippet:

```python
def sevenpoint(pts1, pts2, M):
    T = np.diag([1 / M, 1 / M, 1])
    N = pts1.shape[0]
    ones_arr = np.ones((N,1))
    pts1 = np.hstack((pts1,ones_arr))
    pts2 = np.hstack((pts2,ones_arr))
    norm1 = np.matmul(T,pts1.T).T
    norm2 = np.matmul(T,pts2.T).T

    A = np.ones((N,9))
    for i in range(N):
        x1 = norm1[i,0]
        x2 = norm2[i,0]
        y1 = norm1[i,1]
        y2 = norm2[i,1]
        A[i,:] = [x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]

    A = np.asarray(A)
    D, V = np.linalg.eig(np.dot(A.T, A))
    f1 = np.reshape(V[:, -1], (3, 3))
    f2 = np.reshape(V[:, -2], (3, 3))
    f1 = f1 / f1[2, 2]
    f2 = f2 / f2[2, 2]

    a = sym.symbols('a')
    func = a*f1+(1-a)*f2
    func = sym.Matrix(func)
    d = func.det()
    c0 = d.coeff(a,0)
    c1 = d.coeff(a,1)
    c2 = d.coeff(a,2)
    c3 = d.coeff(a,3)
    C = np.asarray([c0,c1,c2,c3]).astype(float)
    sol = np.polynomial.polynomial.polyroots(C)

    F_list = []
    for root in sol:
        if np.isreal(root):
            F = root*f1+(1-root)*f2
            F = np.matmul(T.T, np.matmul(F, T))
            F = F / F[2, 2]
            F_list.append(F)

    return F_list
```

F:

```
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]]
```

Output Image:



3.1 Essential Matrix

Code Snippet:

```python
def essentialMatrix(F, K1, K2):
    E = np.matmul(np.matmul(K2.T,F),K1)
    E = E/E[2, 2]
    return E
```

E:

```
[[-3.38046291e+00  4.56438521e+02 -2.47359032e+03]
 [ 1.97701379e+02 -1.02883428e+01  6.44014152e+01]
 [ 2.48043824e+03  1.98397474e+01  1.00000000e+00]]
```

3.2 Triangulate

Matrix Ai:

Below is the matrix Ai where C_{11}, C_{12}, and C_{13} are the rows of C_1 and C_{21}, C_{22}, C_{23} are the rows of C_2. Also, (x1, y1) are the 2D coordinates of w_i when projected onto camera 1's image and (x2, y2) are the 2D coordinates of w_i when projected onto camera 2's image.

$$\mathbf{A}_i = \begin{bmatrix} y_1\mathbf{c}_{13} - \mathbf{c}_{12} \\ \mathbf{c}_{11} - x_1\mathbf{c}_{13} \\ y_2\mathbf{c}_{23} - \mathbf{c}_{22} \\ \mathbf{c}_{21} - x_2\mathbf{c}_{23} \end{bmatrix}$$

Code Snippet:

```python
def triangulate(C1, pts1, C2, pts2):
    n = pts1.shape[0]
    X = np.ones((n, 3))
    error = 0

    P1_1 = C1[0, :]
    P1_2 = C1[1, :]
    P1_3 = C1[2, :]
    P2_1 = C2[0, :]
    P2_2 = C2[1, :]
    P2_3 = C2[2, :]

    for i in range(n):
        # get 2D image points
        x1 = pts1[i,0]
        y1 = pts1[i,1]
        x2 = pts2[i,0]
        y2 = pts2[i,1]

        # make A matrix
        A1 = y1*P1_3-P1_2
        A2 = P1_1-x1*P1_3
        A3 = y2*P2_3-P2_2
        A4 = P2_1-x2*P2_3
        A = np.vstack((A1,A2,A3,A4))

        # SVD to get 3D point
        D, V = np.linalg.eig(np.dot(A.T, A))
        idx = np.argmin(D)
        pt3D = V[:, idx]
        pt3D = pt3D/pt3D[3]
        X[i,:] = pt3D[0:3]

        # project back to 2D to get error
        img1_pt = np.matmul(C1,pt3D.T)
        img2_pt = np.matmul(C2,pt3D.T)
        img1_pt = (img1_pt/img1_pt[2])[0:2]
        img2_pt = (img2_pt/img2_pt[2])[0:2]

        # sum error
        e1 = (np.linalg.norm(img1_pt - pts1[i, :]))**2
        e2 = (np.linalg.norm(img2_pt - pts2[i, :]))**2
        error += (e1+e2)

    return X, error
```

3.3 Find M2

FindM2 Code Snippet:

```
def findM2(F, pts1, pts2, intrinsics, filename = 'q3_3.npz'):

    error = math.inf
    best_idx = 0
    K1, K2 = intrinsics['K1'], intrinsics['K2']
    E = essentialMatrix(F, K1, K2)
    M2s = camera2(E)
    M1 = np.hstack((np.eye(3), np.zeros((3, 1))))
    C1 = np.matmul(K1, M1)

    for i in range(4):
        M2 = M2s[:, :, i]
        C2 = np.matmul(K2,M2)
        P, e = triangulate(C1, pts1, C2, pts2)
        if e<error and np.all(P[:, -1] > 0):
            best_idx = i
            error = e

    M2 = M2s[:, :, best_idx]
    C1 = np.matmul(K1,M1)
    C2 = np.matmul(K2,M2)
    P, e = triangulate(C1, pts1, C2, pts2)
    np.savez('q3_3.npz', M2=M2, C2=C2, P=P)
    return M2, C2, P
```

Triangulate Code Snippet:

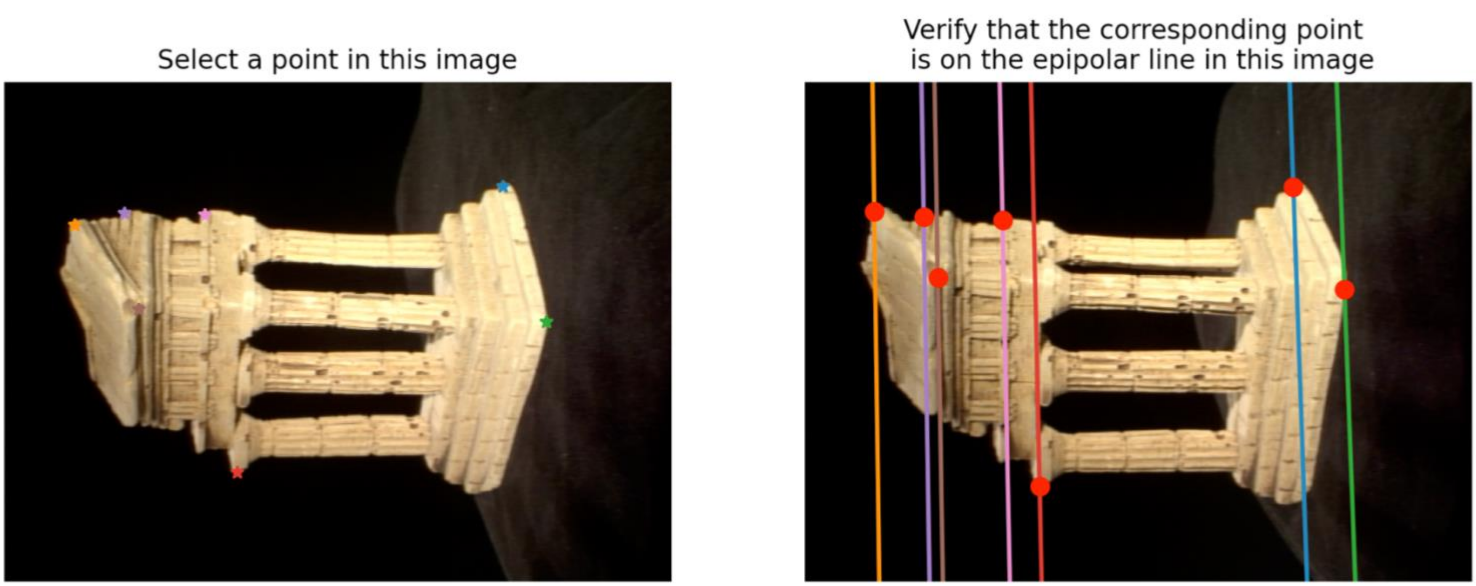
```
def triangulate(C1, pts1, C2, pts2):
    n = pts1.shape[0]
    X = np.ones((n, 3))
    error = 0

    P1_1 = C1[0, :]
    P1_2 = C1[1, :]
    P1_3 = C1[2, :]
    P2_1 = C2[0, :]
    P2_2 = C2[1, :]
    P2_3 = C2[2, :]

    for i in range(n):
        # get 2D image points
        x1 = pts1[i,0]
        y1 = pts1[i,1]
        x2 = pts2[i,0]
        y2 = pts2[i,1]

        # make A matrix
        A1 = y1*P1_3-P1_2
        A2 = P1_1-x1*P1_3
        A3 = y2*P2_3-P2_2
        A4 = P2_1-x2*P2_3
        A = np.vstack((A1,A2,A3,A4))

        # SVD to get 3D point
        D, V = np.linalg.eig(np.dot(A.T, A))
        idx = np.argmin(D)
        pt3D = V[:, idx]
        pt3D = pt3D/pt3D[3]
        X[i,:] = pt3D[0:3]

        # project back to 2D to get error
        img1_pt = np.matmul(C1,pt3D.T)
        img2_pt = np.matmul(C2,pt3D.T)
        img1_pt = (img1_pt/img1_pt[2])[0:2]
        img2_pt = (img2_pt/img2_pt[2])[0:2]

        # sum error
        e1 = (np.linalg.norm(img1_pt - pts1[i, :]))**2
        e2 = (np.linalg.norm(img2_pt - pts2[i, :]))**2
        error += (e1+e2)

    return X, error
```

4.1 Epipolar Correspondence

Code Snippet:

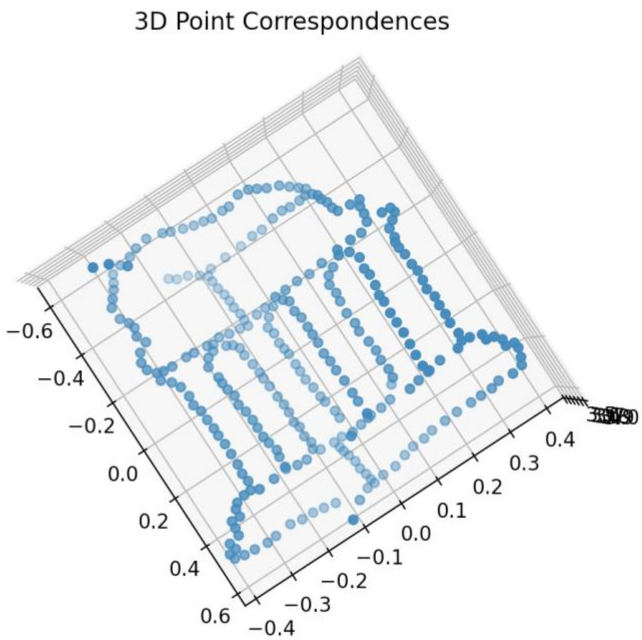
```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    min_dist = math.inf
    best_idx = 0
    h = im1.shape[0]
    w = im1.shape[1]

    # get epipolar line values
    pt = np.array([[x1], [y1], [1]])
    line = np.matmul(F, pt)
    line_y = np.arange(y1-30, y1+30)
    line_x = (-(line[1] * line_y + line[2]) / line[0])

    # gaussian filter
    im11 = ndimage.gaussian_filter(im1, sigma=1, output=np.float64)
    im22 = ndimage.gaussian_filter(im2, sigma=1, output=np.float64)

    # get image 1 patch
    window = 10
    patch1 = im1[y1 - window:y1 + window + 1, x1 - window:x1 + window + 1, :]

    # find best patch2
    for i in range(len(line_x)):
        x2 = int(line_x[i])
        y2 = int(line_y[i])
        if (x2 >= window and x2 <= w - window - 1 and y2 >= window and y2 <= h - window - 1):
            patch2 = im2[y2 - window:y2 + window + 1, x2 - window:x2 + window + 1, :]
            dist = np.linalg.norm(patch2-patch1)
            if (dist < min_dist):
                min_dist = dist
                best_idx = i

    return line_x[best_idx], line_y[best_idx]
```

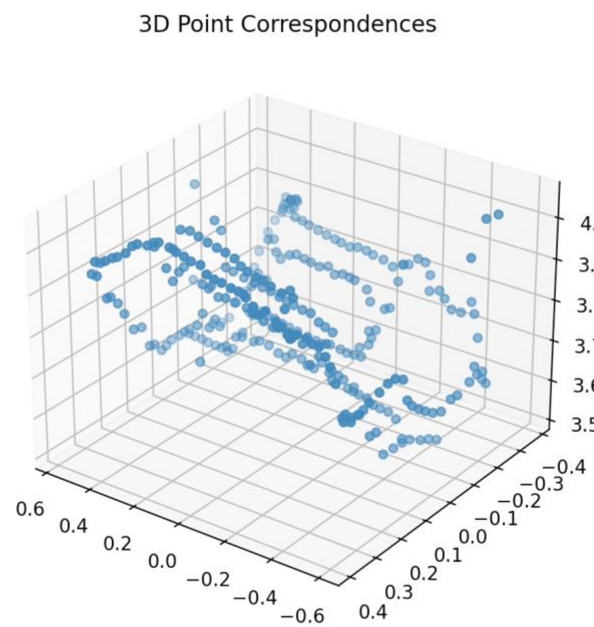
GUI:

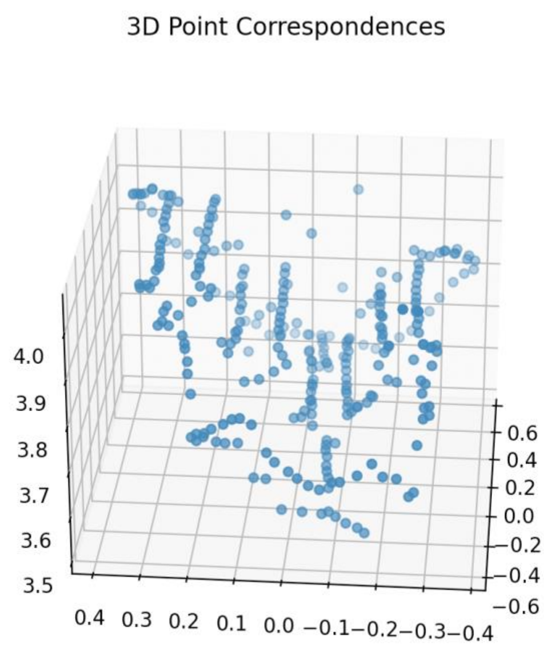


4.2 3D Visualization

Visualization:







Compute_3D_pts code snippet:

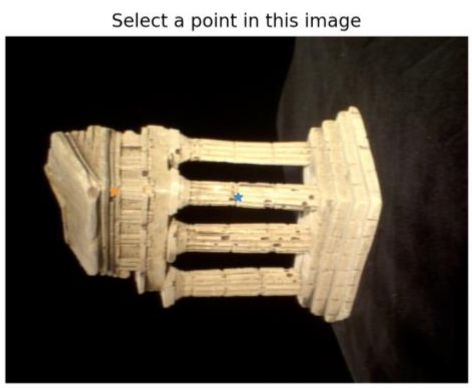
```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    n = temple_pts1.shape[0]
    temple_pts2 = np.zeros(temple_pts1.shape)
    for i in range(n):
        x2, y2 = epipolarCorrespondence(im1, im2, F, temple_pts1[i,0], temple_pts1[i,1])
        temple_pts2[i,0] = x2
        temple_pts2[i,1] = y2
    M1, M2, C1, C2, P = findM2(F, temple_pts1, temple_pts2, intrinsics)
    np.savez('q4_2', F=F, M1=M1, M2=M2, C2=C2, P=P)
    return P
```

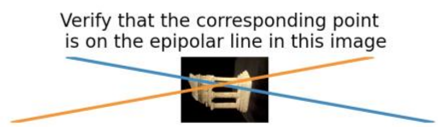
5.1 RANSAC for Fundamental Matrix Recovery

I used the calc_epi_error helper function to calculate the error of each point in my RANSAC function. Calc_epi_error calculates the sum of squared distances between the corresponding points and the estimates epipolar lines. If the error value from the helper function was lower than the threshold value, then I considered that point to be an inlier.

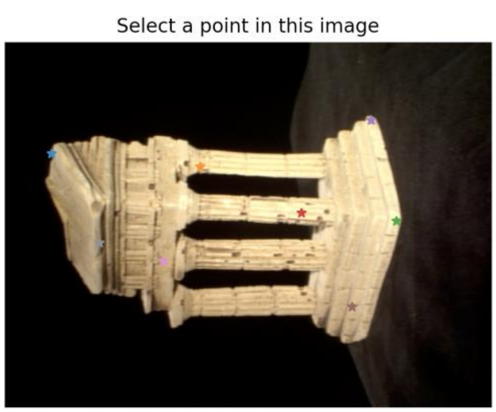
To compare the results of RANSAC and the eight point algorithm, I found F for each of them using the noisy points. Then, I used the displayEpipolarF function to see how accurate of an F matrix was computed. The F matrix provided by the eight point algorithm provided poor and inconclusive results. The results from displayEpipolarF using the F from RANSAC made sense. Clearly, the results from RANSAC are better than the results from just the eight point algorithm since we are iterating through multiple different options for F and choosing the one with the most inliers and least error. The results from each algorithm is shown below.

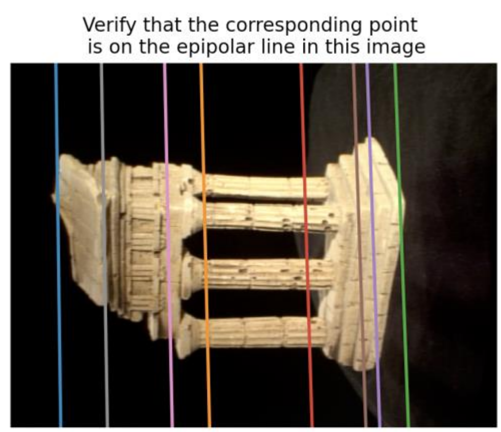
8 point algorithm results using the noisy points:





RANSAC results using the noisy points:





The table below shows the results of varying nIters and the threshold value when running RANSAC and the impact it had on the number of inlier points. From the results we can see that increasing the iterations did not have much of an impact on the results. Increasing the tolerance increased the number of inliers while decreasing it reduced the number of inliers.

Iterations	**Tolerance**	**Number of Inliers**
100	10	101
300	10	99
600	10	103
100	5	41
100	2	19
100	13	105

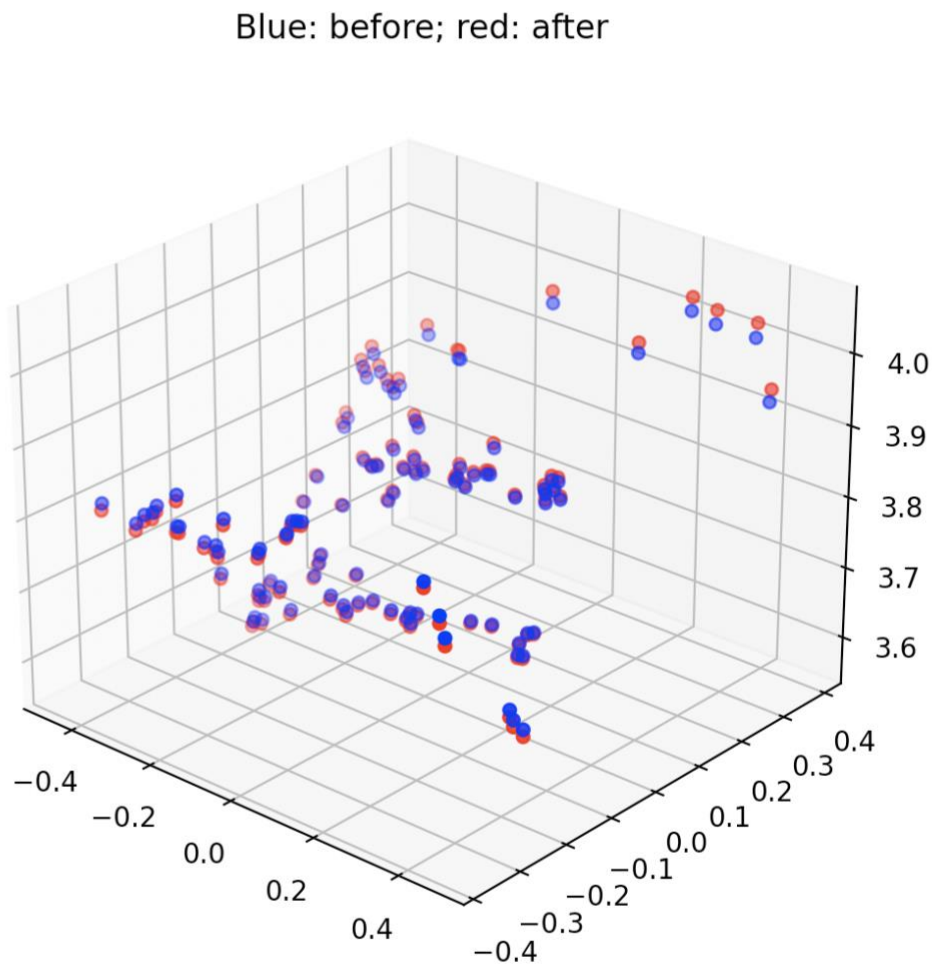
Code Snippet:

```
def ransacF(pts1, pts2, M, nIters=100, tol=10):
    n = pts1.shape[0]
    most_inliers = 0
    ones_arr = np.ones((n, 1))
    pts1 = np.hstack((pts1, ones_arr))
    pts2 = np.hstack((pts2, ones_arr))

    for i in range(nIters):
        print(i)
        # pick random indices
        points_idxs = random.sample(range(0, n), 8)
        rand_pts1 = pts1[points_idxs]
        rand_pts2 = pts2[points_idxs]

        # compute F
        F = eightpoint(rand_pts1[:,0:2], rand_pts2[:,0:2], M)

        # compute num inliers
        err = calc_epi_error(pts1, pts2, F)
        inliers = err < tol
        num_inliers = len(np.where(inliers)[0])

        # check for best F
        if num_inliers >= most_inliers:
            most_inliers = num_inliers
            idx = np.where(inliers)
            inlier_pts1 = pts1[idx]
            inlier_pts2 = pts2[idx]

    F = eightpoint(inlier_pts1[:,0:2], inlier_pts2[:,0:2], M)

    return F, inliers
```

5.2 Rodrigues and Inverse Rodrigues

```python
Q5.2: Rodrigues formula.
    Input:  r, a 3x1 vector
    Output: R, a rotation matrix
'''
def rodrigues(r):
    theta = np.linalg.norm(r)
    u = r/theta
    I = np.eye(3)
    u1 = u[0,0]
    u2 = u[1,0]
    u3 = u[2,0]
    ux = np.array([[0, -u3, u2], [u3, 0, -u1], [-u2, u1, 0]])
    uut = np.matmul(u, u.T)
    R = I*math.cos(theta) + (1-math.cos(theta))*uut + ux*math.sin(theta)
    return R

'''
Q5.2: Inverse Rodrigues formula.
    Input:  R, a rotation matrix
    Output: r, a 3x1 vector
'''
def invRodrigues(R):
    A = (R-R.T)/2
    a32 = A[2,1]
    a13 = A[0,2]
    a21 = A[1,0]
    p = np.array([a32,a13,a21]).T
    s = np.linalg.norm(p)
    r11 = R[0,0]
    r22 = R[1,1]
    r33 = R[2,2]
    c = (r11+r22+r33-1)/2
    u = p/s
    theta = np.arctan2(s,c)
    r = u*theta
    return r.T
```

5.3 Bundle Adjustment



Reprojection Error Before ~ 372.84

Reprojection Error After ~ 9.8235

Code Snippets:

```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    w = x[0:-6].reshape((-1,3))
    r2 = x[-6:-3].reshape((3,1))
    t2 = x[-3:].reshape((3,1))
    n = w.shape[0]
    w_h = np.hstack((w, np.ones((n, 1))))

    R2 = rodrigues(r2)
    M2 = np.hstack((R2, t2))
    C1 = np.matmul(K1, M1)
    C2 = np.matmul(K2, M2)

    p1_hat = np.matmul(C1, w_h.T)
    p2_hat = np.matmul(C2, w_h.T)
    p1_hat = p1_hat / p1_hat[-1, :]
    p2_hat = p2_hat / p2_hat[-1, :]
    p1_hat = p1_hat[0:2,:].T
    p2_hat = p2_hat[0:2,:].T

    residuals = np.concatenate([(p1-p1_hat).reshape([-1]), (p2-p2_hat).reshape([-1])])
    return residuals
```

```
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):

    obj_start = obj_end = 0

    R_init = M2_init[:, :3]
    t_init = M2_init[:, 3:]
    r2_init = invRodrigues(R_init)
    x = np.concatenate([P_init[:, :3].reshape((-1, 1)), r2_init.reshape((-1, 1)), t_init]).reshape((-1, 1))

    fun = lambda x: np.sum(rodriguesResidual(K1, M1, p1, K2, p2, x) ** 2)

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

    P = f[:-6].reshape((-1,3))
    r2 = f[-6:-3].reshape((3,1))
    t2 = f[-3:].reshape((3,1))
    R2 = rodrigues(r2).reshape((3, 3))
    M2 = np.hstack((R2, t2))

    obj_end = t.fun

    return M2, P, obj_start, obj_end

```