Q1.1

$$x^T F x' = 0$$

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

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

(xF)*x' = $[F_{31}\ F_{32}\ F_{33}] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \boxed{= F_{33} = 0}$

Q1.2

$$x_2^T E x_1$$

R = $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$; R is the identity matrix since it is pure translation

$$T = \begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix}$$

$$E = (TxT)*R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

$$\text{Epipolar\_line\_1} = [x_1\ y_1\ 1] * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = [0 \quad t_x \quad y_1 t_x]$$

$$\text{Epipolar\_line\_2} = [x_2\ y_2\ 1] * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = [0 \quad t_x \quad y_2 t_x]$$

As such lines are parallel to x-axis.

Q1.3

At time 1: w_t1 = $R_1 w + t_1$, at time 2: w_t2 = $R_2 w + t_2$

Solve for w in equation 1 for time 1:

$$w = R_1{}^{-1}(w_{t1} - t_1)$$

$$w_{t2} = R_2 R_1{}^{-1}(w_{t1} - t_1) + t_2$$

Simplify to get R_{rel} and t_{rel}

$$w_{t2} = R_2 R_1{}^{-1}(w_{t1} - t_1) + t_2$$

$$w_{t2} = R_2 R_1{}^{-1} w_{t1} - R_2 R_1{}^{-1} t_1 + t_2$$

$$R_{rel} = R_2 R_1{}^{-1}$$

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

Plug into equation: $E = t_{rel} R_{rel}$ and F equation

$$F = KE = K * t_{rel} R_{rel}$$

Q1.4

Assume a 3D point p on an object which are all equidistance to the mirror, there is **no rotation** (R is the identity matrix) and **only translation** between the point p and its reflection in the mirror (p').

$$M_2 = TM_1; T^T T = I$$

$$x_1 = KM_1 p$$

$$x_2 = KTM_1 p$$

$${x_2}^T F x_1 + {x_1}^T F x_2 = 0$$

Substitute the equations for $x_1$and x_2 into transposed equation of ${x_2}^T F x_1$, respectively.

$$(KTM_1 p)^T F(KM_1 p) + (KM_1 p)^T F(KTM_1 p) = 0$$

Solve for F = -F^T

$$K^T T^T {M_1}^T p^T * F * K^T {M_1}^T p^T + K^T {M_1}^T p^T * F * K^T T^T {M_1}^T p^T = 0$$

Rearrange terms to get

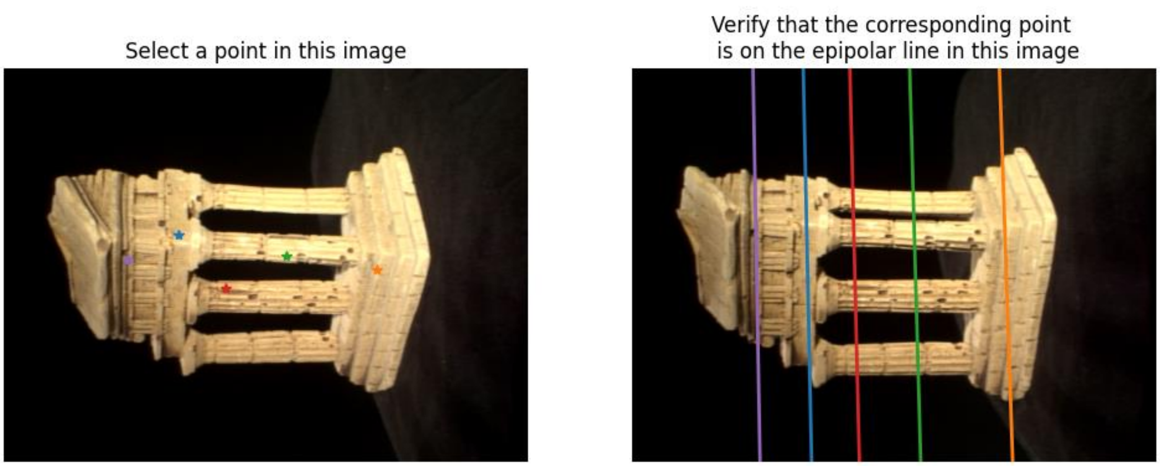
$$K^T (F + F^T) K = 0$$

$$F = -F^T$$

Q2.1

```python
def eightpoint(pts1, pts2, M):
    # Get the diagonal matrix T
    T = np.diag([1/M,1/M,1])
    #Now normalize the points using T Xnorm = Tx
    x1,y1 = pts1[:,0]*T[0,0],pts1[:,1]*T[1,1]
    x1_prime,y1_prime = pts2[:,0]*T[0,0],pts2[:,1]*T[1,1]


    A = np.zeros((pts1.shape[0],9))
    for i in range(0,pts1.shape[0]):
        A[i] = np.array([x1[i]*x1_prime[i],x1[i]*y1_prime[i],x1[i],y1[i]*x1_prime[i],y1[i]*y1_prime[i],y1[i],x1_prime[i],y1_prime[i],1])
        # print(x1[i])

    # (3) Solve for the least square solution using SVD
    # u,s,v = np.linalg.svd(np.dot(A.T,A))
    # print(A)
    u,s,v = np.linalg.svd(A)

    F = np.reshape((v[-1:,]),(3,3))

    # (4) Use the function `_singularize` (provided) to enforce the singularity condition.
    F = _singularize(F)

    # (5) Use the function `refineF` (provided) to refine the computed fundamental matrix.)
    # import pdb; pdb.set_trace()
    F = refineF(F, pts1*T[0,0], pts2*T[1,1])
```

```python
    # (6) Unscale the fundamental matrix Funnorm = T.T*F*T
    F = np.dot(np.dot(T.T,F),T)
    return F
```



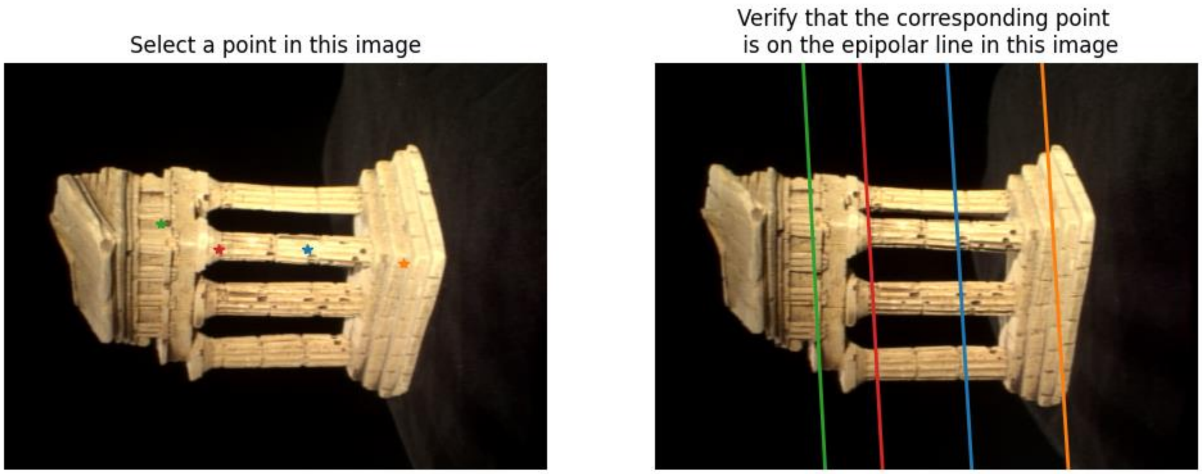

```
[[-8.33149236e-09  1.29538462e-07 -1.17187851e-03]
 [ 6.51358336e-08  5.70670059e-09 -4.13435037e-05]
 [ 1.13078765e-03  1.91823637e-05  4.16862080e-03]]
```

Q2.2

```python
def sevenpoint(pts1, pts2, M):
    # (1) Normalize the input pts1 and pts2 scale paramter M.
    x1,y1 = pts1[:,0]/M,pts1[:,1]/M
    x1_prime,y1_prime = pts2[:,0]/M,pts2[:,1]/M
    # import pdb; pdb.set_trace()
    # (2) Setup the seven point algorithm's equation.
    A = np.zeros((7,9))
    T = np.diag([1/M,1/M,1])

    for i in range(0,7):
        A[i] = np.array([x1[i]*y1[i],x1[i]*y1_prime[i],x1[i],y1[i]*x1_prime[i],y1[i]*y1_prime[i],y1[i],x1_prime[i],y1_prime[i],1])
    # (3) Solve for the least square solution using SVD.
    u,s,vT = np.linalg.svd(A)

    # (4) Pick the last two column vectors of vT.T (the two null space solution f1 and f2)
    F1 = np.reshape((vT[-1,:]),(3,3))
    F2 = np.reshape((vT[-2,:]),(3,3))

    # (5) Use the singularity constraint to solve for the cubic polynomial equation of  F = a*f1 + (1-a)*f2 that leads to
        # det(F) = 0. Sovling this polynomial will give you one or three real solutions of the fundamental matrix.
        # Use np.polynomial.polynomial.polyroots to solve for the roots
    # det(F1+lambda*F2) = a3*lambda**3+a2*lambda**2+a1*lambda+a0 = 0

    #Find alpha such that Determinant(αF1 + (1 − α)F2) = 0; use symbolic variable sympy
    # a = sym.symbols('a')
    F3 = lambda a: a*F1+(1-a)*F2
    # f_det = sym.Matrix(F3)
```

```python
    poly = lambda x: np.linalg.det(x*F1+(1-x)*F2)
    coeff = extract_coefficients(poly,3)
    import pdb; pdb.set_trace()
    #After getting the coefficients need to plug them back into equation and solve for roots

    F_roots = np.polynomial.polynomial.polyroots(coeff)
    Y = np.roots(coeff)
    Farray = []


    #Now get the real parts
    for s in F_roots:
        if np.isreal(s):
            F = s*F1 + (1-s)*F2
            F = (np.dot(T.T,F)).dot(T)
            Farray.append(F)
    # import pdb; pdb.set_trace()

    # raise NotImplementedError()
    return Farray


def extract_coefficients(p, degree):
    n = degree + 1
    sample_x = [ x for x in range(n) ]
    sample_y = [ p(x) for x in sample_x ]
```

```python
    A = [ [ 0 for _ in range(n) ] for _ in range(n) ]
    for line in range(n):
        for column in range(n):
            A[line][column] = sample_x[line] ** column
    c = np.linalg.solve(A, sample_y)
    return c[::-1]
```



Farray returned:

```
[array([[-8.52983359e-07,  2.79109167e-06, -1.01587235e-03],
       [-3.19010842e-06,  4.45181394e-06, -1.69052211e-03],
       [ 1.46956549e-03, -2.51229574e-04,  1.44878089e-01]]), array([[-3.52564715e-07,  1.11006058e-06,  1.31073338e-04],
       [-1.37241753e-06,  1.89028783e-06, -7.42720419e-04],
       [ 7.78283379e-05, -7.28483570e-05,  5.98232671e-02]]), array([[ 1.51266329e-08, -1.25106396e-07,  9.73811732e-04],
       [-3.68373664e-08,  8.16173708e-09, -4.63065567e-05],
       [-9.44774867e-04,  5.82203610e-05, -2.67225007e-03]])]
```

Q3.1

```python
'''
Q3.1: Compute the essential matrix E.
    Input:  F, fundamental matrix
            K1, internal camera calibration matrix of camera 1
            K2, internal camera calibration matrix of camera 2
    Output: E, the essential matrix
'''
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    E = np.dot(np.dot(K2.T,F),K1)
    return E
```

Q3.2

```python
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation
    x1,y1 = pts1[:,0],pts1[:,1]
    x1_prime,y1_prime = pts2[:,0],pts2[:,1]

    p1 = C1[0,:].T
    p2 = C1[1,:].T
    p3 = C1[2,:].T

    p1_prime = C2[0,:].T
    p2_prime = C2[1,:].T
    p3_prime = C2[2,:].T

    #x = P*X where P is the camera matrix
    non_homo = np.zeros((pts1.shape[0],4))
    homogenous = np.zeros((pts1.shape[0],3))
    repr_error = 0

    proj_norm = np.zeros((3,pts1.shape[0]))
    for i in range(0,pts1.shape[0]):
        row1 = np.array([y1[i]*p3-p2])
        row2 = np.array([x1[i]*p3-p1])
        row3 = np.array([y1_prime[i]*p3_prime-p2_prime])
        row4 = np.array([x1_prime[i]*p3_prime-p1_prime])
        A = np.vstack((row1,row2,row3,row4)) #4x4
    # import pdb; pdb.set_trace()
    # (3) Solve for the least square solution using SVD.
```

```python
    # (3) Solve for the least square solution using SVD.
        u,s,vT = np.linalg.svd(A)
        F = vT[-1,:]
    #Non-homogenous is form (X,Y,Z,1) and Homogenous is form (X,Y,Z)
        homogenous[i] = F[0:3]/F[-1]
        non_homo[i,:] = np.array([homogenous[i][0],homogenous[i][1],homogenous[i][2],np.ones(1)],dtype=object)#Non-homogenous matrix

    # # (4) Calculate the reprojection error using the calculated 3D points and C1 & C2
    #(pts1[i]-proj1)**2 proj1 = np.dot(C1,non_homo[i,:].T)
    #proj_matrix_1 = C1*[X,Y,Z,1]

    proj_matrix_1 = np.dot(C1,non_homo.T).T
    proj_matrix_2 = np.dot(C2,non_homo.T).T

    #Reprojection error is the squared error between the true image coordinaties of a point and
    #the projected coordinations of hypothesized 3D points
    # for i in range(0,pts1.shape[0]):
    #x1 = P11(x)+P12(y)+P13(Z)+P14/Z --> x1 - P(1)/P(3); same for y1
    # repr_error = np.sum((pts1[i]- np.dot(C1,non_homo[i])**2))+np.sum((pts2[i]-np.dot(C2,non_homo[i])))

    ##Keeping track of 3D Points; Homogenize coordinates again (divide by Z)
    for i in range (0, pts1.shape[0]):
        proj_norm = proj_matrix_1[i]/proj_matrix_1[i][-1] #3XN
        proj_norm2 = proj_matrix_2[i]/proj_matrix_2[i][-1]

    #Reprojection error; Keep track of error
        repr_error += np.sum((pts1[i]- proj_norm.T[0:2].T)**2)+np.sum((pts2[i]-proj_norm2.T[0:2].T)**2)
```

$$
A_i = \begin{array}{cc}
y_1 * {C_{13}}^T & -{C_{12}}^T \\
{C_{11}}^T & -x_1 {C_{13}}^T \\
y_{2*} {C_{23}}^T & -x_2 {C_{22}}^T \\
\multicolumn{2}{l}{{C_{21}}^T - x_2 {C_{11}}^T}
\end{array}
$$

$$\begin{bmatrix} y\boldsymbol{p}_3^\top - \boldsymbol{p}_2^\top \\ \boldsymbol{p}_1^\top - x\boldsymbol{p}_3^\top \\ y'\boldsymbol{p}_3'^\top - \boldsymbol{p}_2'^\top \\ \boldsymbol{p}_1'^\top - x'\boldsymbol{p}_3'^\top \end{bmatrix}$$

The reprojection error varied between 99 – 352, as I changed my implementation of the Ai matrix, with 352 being the last value achieved. Note equation may differ from what was implemented as using the original Ai matrix resulted in a lot of errors

Q3.3

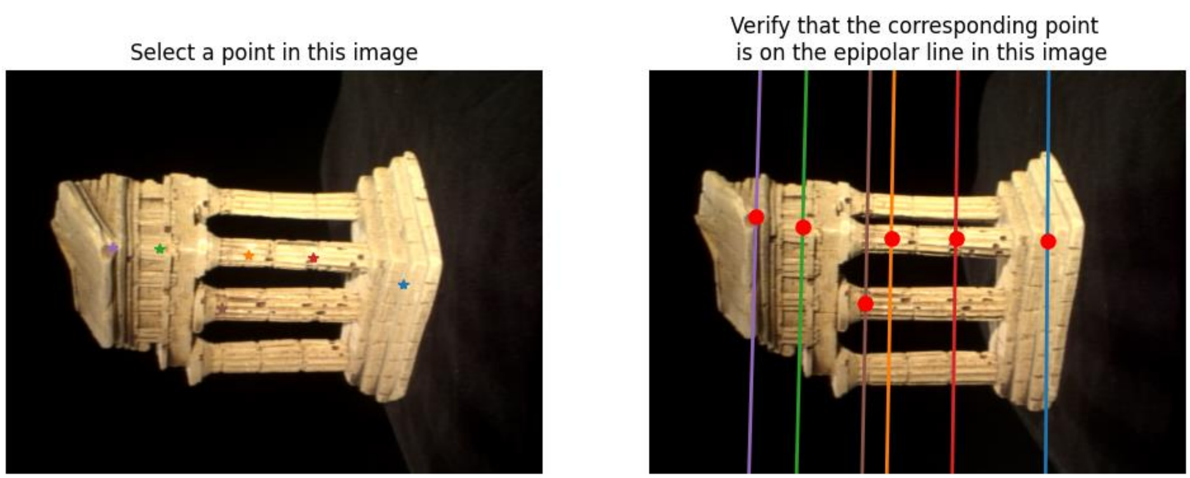
```python
def findM2(F, pts1, pts2, intrinsics, filename = 'q3_3.npz'):
    '''
    Q2.2: Function to find the camera2's projective matrix given correspondences
        Input:  F, the pre-computed fundamental matrix
                pts1, the Nx2 matrix with the 2D image coordinates per row
                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 = intrinsics['K1']
    K2 = intrinsics['K2']
    E = essentialMatrix(F, K1, K2)
    M2 = camera2(E)
    M1 = np.hstack((np.identity(3), np.zeros(3)[:,np.newaxis]))

    #Get C1: 3x4 camera matrix; M1 is just identity matrix
    C1 = np.dot(K1,M1)
    error_ = 0 #tracking best_error
    p_save = 0 #saving P
```

```python
    for i in range (0,M2.shape[2]):
        C2 = np.dot(K2,M2[:,:,i]) #Getting C2 values and check the last index of M2 since that is where the values are stored
        p,error_2 = triangulate(C1,pts1,C2,pts2)
        #The z needs to be a positive value since negayive z means point is behind the camera

        if (p[:,-1] >= 0).all():
            # Keep track of the projection error through best_error and retain the best one.
             error_ = error_2
             p_save = p
             M2_ = M2[:,:,i]
             break #Keep M2s; If there is no break then the correct M2 is not found

    np.savez('code/data/filename', M2_,C2, p_save)
    return M2_, C2, p_save
```

Q4.1

```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Replace pass by your implementation
    x = np.array([x1,y1,1])
    ep_line = np.dot(F,x) #Epipolar line on image 1
    # ep_line_x2 = np.dot(F,x2)
    #Search window
    window = 11
    window_def = im1[y1-window//2:y1+window//2+1,x1-window//2:x1+window//2+1] #This is a fixed window for image 1

    y2_search_array = np.array(range(y1-50,y1+50))
    x2_search_array = np.round(-(ep_line[1]*y2_search_array+ep_line[2])/ep_line[0]).astype(int)#Form. epipolar line

    e_dist = np.inf
    x2 = None
    y2 = None

    #Apply Gaussian weight to window 1
    blurred_window_1 = gaussian_filter(window_def, sigma = 3)

    # import pdb; pdb.set_trace()
    #Search along this line to check nearby pixel intensity
    for i in range(0,len(x2_search_array)):
        k = (x2_search_array>= window//2) & (y2_search_array >= window//2) & (x2_search_array <im2.shape[1]-window//2) & (y2_search_array <im2.shape[0]-window//2)

        if k.all():
            x2_search_array = x2_search_array[k]
            y2_search_array = y2_search_array[k]
```

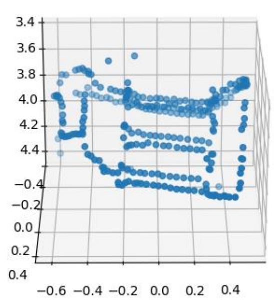
```python
        #Second window, this is changing
        shift = y2_search_array[i]
        shift_x = x2_search_array[i]

        window_def_2 = im2[shift-int(window//2):shift+int(window//2)+1,shift_x-window//2:shift_x+int(window//2)+1]
        blurred_window_2 = gaussian_filter(window_def_2, sigma = 3)
        # scipy.ndimage.gaussian_filter(im2, sigma)
        erro_d = np.sum(np.linalg.norm(blurred_window_1-blurred_window_2))
        if erro_d < e_dist:
            e_dist = erro_d
            x2,y2 = x2_search_array[i],y2_search_array[i]
    return x2,y2
```

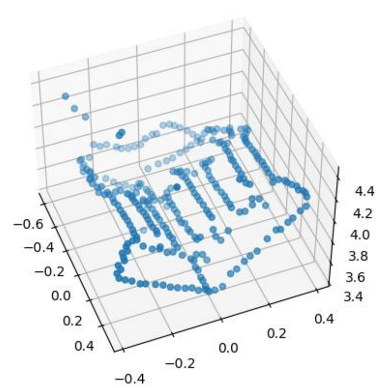


Q4.2

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

    #Get x2, y2; x1 and y1 passed in to epipolar will be at a single index
    x2 = np.zeros(shape = (1,temple_pts1['x1'].shape[0]))
    y2 = np.zeros(shape = (1,temple_pts1['y1'].shape[0]))

    for i in range(0,temple_pts1['x1'].shape[0]):
        x2_, y2_ = epipolarCorrespondence(im1, im2, F, int(temple_pts1['x1'][i]), int(temple_pts1['y1'][i]))
        x2[0][i] = x2_
        y2[0][i] = y2_

    #Compute M2 matrix and use triangulate to find 3D points
    M2,C2,P = findM2(F, pts1, pts2, intrinsics, filename = 'q4_2.npz') #Triangulate is called in the M2

    # homograpy, error_ = triangulate(C1, temple_pts1[0], C2, temple_pts1[1])

    return P
```

```
Q4.2:
    1. Integrating everything together.
    2. Loads necessary files from ../data/ and visualizes 3D reconstruction using scatter
'''
if __name__ == "__main__":

    temple_coords_path = np.load('code/data/templeCoords.npz')
    correspondence = np.load('code/data/some_corresp.npz') # Loading correspondences
    intrinsics = np.load('code/data/intrinsics.npz') # Loading the intrinscis of the camera
    K1, K2 = intrinsics['K1'], intrinsics['K2']
    pts1, pts2 = correspondence['pts1'], correspondence['pts2']
    im1 = plt.imread('code/data/im1.png')
    im2 = plt.imread('code/data/im2.png')


    F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
    P = compute3D_pts(temple_coords_path, intrinsics, F, im1, im2)

    xs = P[:,0]
    ys = P[:,1]
    zs = P[:,2]

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.scatter(xs,ys,zs)

    plt.show()
```





Q5.1

```python
def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    # Replace pass by your implementation
    #First compute the fundamental matrix F

    #Keep track of inliers
    current_inliers = 0
    inliers = []
    numb_inliers = 0
    ct_inliers = []

    p_prime = np.zeros((pts1.shape[0],3))
    for i in range(nIters):
        print("Here:         ",i)
        index = np.random.choice(pts1.shape[0],8,False)

        x1 = pts1[index]
        x2 = pts2[index]

        F_ = eightpoint(x1,x2,M) #My sevenpoint is not that accurate..

        #Calculate distance for each correspondence
        for k in range(pts1.shape[0]):
            p_prime = np.append(pts2[k],1)
            error = np.dot(F_,p_prime)

            x2_x = error[0]
            x2_y = error[1]
```

```python
            distance_1 = np.sqrt(x2_x**2+x2_y**2)
            distance = abs(np.dot(p_prime.T,error)/distance_1)

            #Find the inliers
            # error_ = error/np.sqrt(np.sum(error[:,2:]**2,axis = 0))
            # dist = abs(np.sum(homo_2*error_,axis = 0))
            if (distance<= tol).all():
                numb_inliers+=1
                ct_inliers.append(1)
            # else:
            #     ct_inliers.append(0)
            #     numb_inliers = 0

            if numb_inliers > current_inliers:
                current_inliers = numb_inliers
                inliers = ct_inliers
                F = F_

    if F[2,2] != 1:
        F = F/F[2,2]

    return F, inliers
```

The Ransac method improved the epipolar line correspondences. I played around with raising and lowering the iterations, and lowering the number of iterations made the Ransac method less accurate versus raising the number of iterations actually yielded better results (though I defaulted to the 1000 used in the hw assignment for the write-up). Using a similar implementation to the previous homework, I implemented the Ransac method by checking for inliers where the computed error (distance) was less than the tolerance. For the error computation, I looked at the distance between the point and the error/distance_1 (this expression was meant to represent where I would predict where the points would actually fall on the epipolar line).

Q5.2

```python
Q5.2: Rodrigues formula.
    Input:  r, a 3x1 vector
    Output: R, a rotation matrix
'''
def rodrigues(r):
    # Replace pass by your implementation
    theta = np.linalg.norm(r) #theta = ||r||
    u = r/theta
    if theta <= np.pi:
        if theta == 0:
            R = np.diag([1,1,1])
        else:
            ux = np.array([[0,-u[2],u[1]],[u[2],0,-u[0]],[-u[1],u[0],0]])

            u_t = (u.reshape(3,1) @ u.reshape(3,1).T)
            R = np.diag([1,1,1])*np.cos(theta)+(1-np.cos(theta))*u_t+ux*np.sin(theta)
    else:
        R = np.eye(3)
    return R
```

```python
def invRodrigues(R):
    A = (R-R.T)//2
    rho = np.array([A[2][1],A[0][2],A[1][0]]).T
    s = np.linalg.norm(rho)
    c = (R[0][0]+R[1][1]+R[2][2]-1)//2

    # r = None
    if (s == 0 and c == 1):
        r = np.zeros((3,1))

    elif (s == 0 and c == -1):
        # index = np.where((R+np.diag[1,1,1]) != 0)
        # v = (R+np.diag[1,1,1])[:,index]
        I = R + np.diag([1,1,1])
        for i in range(3):
            if (np.count_nonzero(I[:,i]))>0 and (np.count_nonzero(I[:,i]))!=0:
                v = I[:, i]
        u = v/np.linalg.norm(v)
        r_ = u*np.pi

        if ((np.linalg.norm(r) == np.pi) and (r[0][0] == 0 and r[1][0] == 0 and r[3][0]<0)
        or (r[0][0] == 0 and r[1][0]<0) or (r[0][0]<0)):

            r = -r_ #r negative
        else:
            r = r_
```

```python
    else:
        u = rho/s
        theta = np.arctan2(s,c)
        r = u*theta
    return r
```

Q5.3

```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation
    C1 = np.dot(K1,M1)

    w = np.reshape(x[:-6],shape = (p1.shape[0],3))
    t2 = np.reshape(x[-6:-3],shape = (3,1))
    r2 = np.reshape(x[-3:],shape = (3,1))
    R2 = rodrigues(r2)

    M2 = np.htsack(R2,x[-3:].reshape(3,1))
    P_homo = np.hstack(w,np.ones(p1.shape[0],1))
    C2 = np.dot(K2,M2)

    p1_= np.dot(C1,P_homo.T).T
    p2_ = np.dot(C2,P_homo.T).T

    p1_hat = p1_[:2,:]/p1_[2,:]
    p2_hat =  p1_[:2,:]/p1_[2,:]


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

I was not able to get the correct results for running this question, I believe this is due to not computing the inliers correctly in the RansacF function but the results would output incorrectly for each iteration. I also did not have enough time to finish the implementation of this question (specifically the debugging process in the bundle adjustment portion of the question).

Note: I missed some parts of the questions where a .npz file was required to save the results. I did go back and try to capture as many as I could before the deadline.