16-720B HW4 Writeup

Nirmay Singaram

March 2022

1 Part I

1.1 Q 1.1

Since the principle axes intersect at the point P, we have that $\mathbf{x}_2^T \mathbf{F} \mathbf{x}_1 = 0$ where $\mathbf{x}_i$ corresponds to $\mathbf{x}$ with respect to each camera after normalization. Then this means that:

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

But since these are origins we get:

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0 \tag{2}$$

And so by using basic matrix algebra we see that the left hand side evaluates to $F_{33} = 0$.

1.2 Q1.2

Since the translation is parallel to the x-axis we have $\mathbf{t} = \begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix}$. Since there is no rotation $R = I$ where R is the rotation matrix.

The cross product matrix is $t_\times = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$ which gives us that

$$E = t_\times R = t_\times = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \tag{3}$$

This means that the epipolar lines are $\mathbf{x}_2^T E^T$ and $\mathbf{x}_1^T E$ which gives us:

$$\mathbf{l}_1 = \mathbf{x}_2^T E = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & t_x & -t_x y_2 \end{bmatrix} \tag{4}$$

$$\mathbf{l}_2 = \mathbf{x}_1^T E^T = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t_x \\ 0 & -t_x & 0 \end{bmatrix} = \begin{bmatrix} 0 & t_x & -t_x y_2 \end{bmatrix} \tag{5}$$

$\mathbf{l}_1$ is the equation $t_x y - t x y_2 = 0$ which is parallel to the x-axis as y is fixed to be y_2 and x is free. Similarly $\mathbf{l}_2$ is the equation $t_x y - t_x y_1 = 0$ which is parallel to the x-axis as y is fixed to be y_1 and x is free. So both epipolar lines are parallel to x-axis.

1.3 Q1.3

If said point is $\mathbf{y}$ then with respect to both these cameras the corresponding points are:

$\mathbf{y}_1 = \mathbf{R}_1\mathbf{y} + \mathbf{t}_1$ for camera 1 and $\mathbf{y}_2 = \mathbf{R}_2\mathbf{y} + \mathbf{t}_2$

Rearrange the first equation to get: $\mathbf{y} = \mathbf{R}_1^{-1}(\mathbf{y}_1 - \mathbf{t}_1)$. Substitute this into the second equation to get:

$\mathbf{y}_2 = \mathbf{R}_2\mathbf{R}_1^{-1}(\mathbf{y}_1 - \mathbf{t}_1) + \mathbf{t}_2$

Expand this to get:

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

which tells us that: $\mathbf{R}_{rel} = \mathbf{R}_2\mathbf{R}_1^{-1}$ and $\mathbf{t}_{rel} = -\mathbf{R}_2\mathbf{R}_1^{-1}\mathbf{t}_1 + \mathbf{t}_2$

Once we know $\mathbf{t}_{rel}$ we may create the cross matrix $\mathbf{t}_{\times rel}$ from which we get:

$\mathbf{E} = \mathbf{t}_{rel\times}\mathbf{R}_{rel}$

We already know the camera intrinsics $\mathbf{K}$ and so:

$\mathbf{F} = (\mathbf{K}^{-1})^T\mathbf{E}\mathbf{K}^{-1} = (\mathbf{K}^{-1})^T\mathbf{t}_{\times rel}\mathbf{R}_{rel}\mathbf{K}^{-1}$

1.4 Q1.4

Suppose C_1 is the camera and C_2 the reflection. And say the camera intrinsics properties gives us $\lambda_1\mathbf{x}_1 = \mathbf{K}\mathbf{P}_1$ where $\mathbf{P}_1$ is the point and $\mathbf{x}_1$ is the image for C_1 and similarly $\lambda_2\mathbf{x}_2 = \mathbf{K}\mathbf{P}_2$ for C_2, where $\mathbf{K}$ is the camera intrinsics, etc.

The camera is separated from its reflection only be a translation $\mathbf{t}$ of course, which means that $\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{t}$ and so we get: $\lambda_2\mathbf{K}^{-1}\mathbf{x}_2 = \lambda_1\mathbf{K}^{-1}\mathbf{x}_1 + \mathbf{t}$

Cross product with $\mathbf{t}$ on both sides. Since $\mathbf{t} \times \mathbf{t} = 0$ we then get:

$\lambda_2\mathbf{K}^{-1}(\mathbf{t} \times \mathbf{x}_2) = \lambda_1\mathbf{K}^{-1}(\mathbf{t} \times \mathbf{x}_1)$

Take the dot product with $\mathbf{P}_2 = \lambda_2\mathbf{K}^{-1}\mathbf{x}_2$ on both sides to get:

$(\lambda_2\mathbf{K}^{-1}\mathbf{x}_2)^T(\lambda_2\mathbf{K}^{-1}\mathbf{t} \times \mathbf{x}_2) = (\lambda_2\mathbf{K}^{-1}\mathbf{x}_2)^T(\lambda_1\mathbf{K}^{-1}\mathbf{t} \times \mathbf{x}_1)$

$\implies \lambda_2^2 K^{-T}\mathbf{x}_2^T\mathbf{K}^{-1}(\mathbf{t} \times \mathbf{x}_2) = \lambda_2\lambda_1\mathbf{K}^{-T}\mathbf{x}_2^T\mathbf{K}^{-1}(\mathbf{t} \times \mathbf{x}_1)$

Since $\mathbf{t} \times \mathbf{x}_2$ is perpendicular to $\mathbf{x}_2$ we have that $(\mathbf{K}^{-1}\mathbf{x}_2)^T(\mathbf{K}^{-1}\mathbf{t} \times \mathbf{x}_2) = \mathbf{K}^{-T}\mathbf{x}_2^T\mathbf{K}^{-1}\mathbf{t} \times \mathbf{x}_2 = 0$

so the left hand side of the above equation evaluates 0, which gives us that:

$\mathbf{K}^{-T}\mathbf{x}_2^T\mathbf{K}^{-1}\mathbf{t} \times \mathbf{x}_1 = 0$

But this is just $\mathbf{x}_2^T(K^{-T}\mathbf{t}_\times K^{-1})\mathbf{x}_1 = 0$ and since the relative rotation is identity we thus get that the fundamental matrix $F = K^{-T}\mathbf{t}_\times\mathbf{K}^{-1}$

$\mathbf{t}_\times$ is Skew symmetric of course as noted in class and just by looking at the matrix, and so $\mathbf{t}_\times^T = -\mathbf{t}_\times$, and thus $\mathbf{F}^T = (\mathbf{K}^{-T}\mathbf{t}_\times\mathbf{K}^{-1})^T = \mathbf{K}^{-T}\mathbf{t}_\times^T(\mathbf{K}^{-T})^T = \mathbf{K}^{-T}(-\mathbf{t}_\times)\mathbf{K}^{-1} = -\mathbf{F}$ and so $\mathbf{F}$ is skew symmetric.

2 Practice

2.1 Q2.1

I was confused about whether we were supposed to normalize or not normalize the final $\mathbf{F}$ because of one of the assertions. I did both and will show both. Here is the code snippet for the algorithm itself:

```
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation
    pts1 = np.divide(pts1, M)
    pts2 = np.divide(pts2, M)
    t = np.diag([1/M, 1/M, 1])
    A = np.vstack([pts1[:, 0]*pts2[:, 0], pts1[:, 0]*pts2[:, 1],
        pts1[:, 0], pts1[:, 1]*pts2[:, 0], pts1[:, 1]*pts2[:, 1],
        pts1[:, 1], pts2[:, 0], pts2[:, 1], np.ones(pts1.shape[0])])
    U, S, V = np.linalg.svd(A.T)
    F = V[-1, :].reshape(3, 3)
    F = helper._singularize(F)
    F = helper.refineF(F, pts1, pts2)
    F = (t.T)@F@t
    return F
```

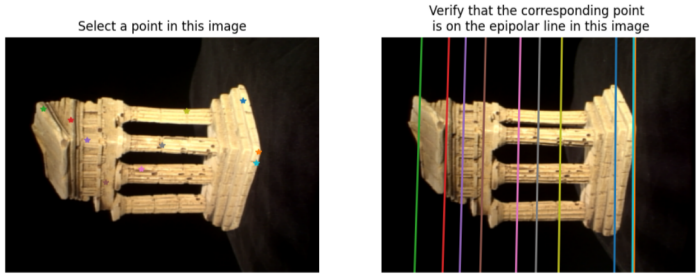
Before normalizing this is what we get:

$$\begin{bmatrix} -8.33179496e-09 & 1.29533454e-07 & -1.17187884e-03 \\ 6.51340406e-08 & 5.70605431e-09 & -4.13402143e-05 \\ 1.13078847e-03 & 1.91811381e-05 & 4.16853309e-03 \end{bmatrix}$$

After normalization, i.e, so that the assert statement is met:

$$\begin{bmatrix} -1.99873547e-06 & 3.10741096e-05 & -2.81124994e-01 \\ 1.56251706e-05 & 1.36883987e-06 & -9.91720908e-03 \\ 2.71267721e-01 & 4.60141197e-03 & 1.00000000e+00 \end{bmatrix}$$

I saved the latter since that's what the assert statement is asking for. Here's the picture:



2.2 Q2.2

Again I just normalized it so that the assertions are passed. Here's the code:

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

    Farray = []
    # ----- TODO -----
    # YOUR CODE HERE
    pts1 = pts1/M
    pts2 = pts2/M
    t = np.diag([1/M, 1/M, 1])
    A = np.vstack([pts2[:, 0]*pts1[:, 0], pts2[:, 0]*pts1[:, 1],
            pts2[:, 0], pts2[:, 1]*pts1[:, 0], pts2[:, 1]*pts1[:, 1],
            pts2[:, 1], pts1[:, 0], pts1[:, 1], np.ones(pts1.shape[0])])
    U, S, V = np.linalg.svd(A.T)
    F_1 = V[-1,:].reshape(3, 3)
    F_2 = V[-2,:].reshape(3, 3)
    funct = lambda a: np.linalg.det(a*F_1 + (1-a)*F_2)
    a_0 = funct(0)
```

```
    a_1 = (2/3)*(funct(1) - funct(-1)) - (1/12)*(funct(2) - funct(-2))
    a_2 = (1/2)*funct(1) + (1/2)*funct(-1) - funct(0)
    a_3 = ((1/12)*(funct(2) - funct(-2))) - ((1/6)*(funct(1) - funct(-1)))
    roots = np.roots([a_3, a_2, a_1, a_0])
    roots = np.real(roots[np.isreal(roots)])
    Farray = []
    for r in roots:
        F = r*F_1 + (1-r)*F_2
        F = _singularize(F)
        F = refineF(F, pts1, pts2)
        F = (t.T@F)@t
        Farray.append(F/F[2,2])


    return Farray
```

And here is the code that actually chooses the points and generates the image:

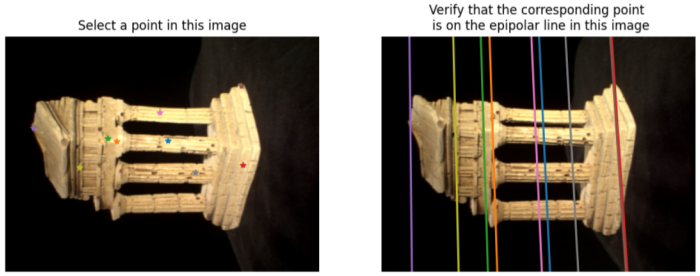
```
if __name__ == "__main__":

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


    # ----- TODO -----
    # YOUR CODE HERE
    np.random.seed(4)
    r = [56, 26, 104, 62, 53 , 92, 22] #Choices obtained using the code below
    F = sevenpoint(pts1[r, :], pts2[r, :], np.max([*im1.shape, *im2.shape]))
    #After running it I saw that there are three Fs and the third one was the best so I sa
    np.savez('q2_2', F = F[2])
    print(F)
    displayEpipolarF(im1, im2, F[2])
```

Here are the F matrices:

$$\begin{bmatrix} -9.49866741e-06 & 5.79536949e-04 & -2.42429151e-01 \\ -5.29521938e-04 & -3.30277056e-05 & 1.66769292e-01 \\ 2.30076303e-01 & -1.59297705e-01 & 1.00000000e+00 \end{bmatrix}$$

$$\begin{bmatrix} -1.85930219e-05 & 1.08292507e-03 & -2.78951444e-01 \\ -1.01987797e-03 & -6.16338303e-05 & 3.13272161e-01 \\ 2.63505307e-01 & -2.96118460e-01 & 1.00000000e+00 \end{bmatrix}$$
$$\begin{bmatrix} 8.10457567e-07 & 8.90919506e-06 & -2.01028424e-01 \\ 2.63329748e-05 & -6.00542594e-07 & 6.97429503e-04 \\ 1.92182049e-01 & -4.20123580e-03 & 1.00000000e+00 \end{bmatrix}$$

Here's the image associated with the best matrix:



3 Metric Reconstruction

3.1 Q3.1

Once again I am confused about when and where and how many times the normalization is being asked. The formula for $\mathbf{E}$ is $\mathbf{E} = \mathbf{K}_2^T\mathbf{F}\mathbf{K}_1$ but we're required to pass an assertion of $E[2,2] = 1$ so do we divide by $E[2,2]$? I don't think normalizing F carries over to E so would that be 2 normalizations?

I really think the HW should have been clearer on this topic.

I ended up normalizing both F and E since each step seems to suggest that we should normalize. This is the $\mathbf{E}$ we get:

$$\begin{bmatrix} -3.36615965e+00 & 4.56052787e+02 & -2.47343036e+03 \\ 1.98055779e+02 & -1.02807951e+01 & 6.44171617e+01 \\ 2.48028021e+03 & 1.98174709e+01 & 1.00000000e+00 \end{bmatrix}$$

Note that the values are typically a lot smaller if we don't do the normalization.

3.2 Q3.2

Given cameras $\mathbf{C}_1$ and $\mathbf{C}_2$ and $\tilde{\mathbf{w}}_i$ are the coordinates in homogeneous form, we then have that $\mathbf{C}_1\tilde{\mathbf{w}}_i = \widehat{\mathbf{x}_{1i}}$ where $\widehat{x_1}$ is the projection of $\mathbf{w}_i$ via camera $\mathbf{C}_1$ as given in the handout. Here the subscript i indicates the ith row/element of the vector. So $\mathbf{C}_{1i}$ is the ith row of $\mathbf{C}_1$. Similarly we have $\mathbf{C_2}\tilde{\mathbf{w}}_\mathbf{2} = \widehat{\mathbf{x}_{2i}}$. Then let $\widehat{\mathbf{x}_{ji}} = [u_{ji}, v_{ji}, 1]$ where j is 1 or 2.

Expanded we get:

$$\begin{bmatrix} \mathbf{C}_{11} \\ \mathbf{C}_{12} \\ \mathbf{C}_{13} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} u_{1i} \\ v_{1i} \\ 1 \end{bmatrix} \tag{6}$$

and similarly:

$$\begin{bmatrix} \mathbf{C}_{21} \\ \mathbf{C}_{22} \\ \mathbf{C}_{23} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = \begin{bmatrix} u_{2i} \\ v_{2i} \\ 1 \end{bmatrix} \tag{7}$$

From here it's just a matter of rearranging:

The last row of the first equation gives you that $\mathbf{C}_{13} \cdot \tilde{\mathbf{w}}_i = 1$. We use this as a substitution in the other two rows.

For instance, the first row of the first equation gives you that:

$\mathbf{C}_{11} \cdot \tilde{\mathbf{w}}_i = u_{1i} = u_{1i}(\mathbf{C}_{13} \cdot \tilde{\mathbf{w}}_i)$ which gives us that $(u_{1i}\mathbf{C}_{13} - \mathbf{C}_{11}) \cdot \tilde{\mathbf{w}}_i = 0$

Using the same argument the second row gives us that:

$\mathbf{C}_{12} \cdot \tilde{\mathbf{w}}_i = v_{1i} = v_{1i}(\mathbf{C}_{13} \cdot \tilde{\mathbf{w}}_i)$ which gives us that $(v_{1i}\mathbf{C}_{13} - \mathbf{C}_{12}) \cdot \tilde{\mathbf{w}}_i = 0$

The last row of the second equation tells us that: $\mathbf{C}_{23} \cdot \tilde{\mathbf{w}}_i = \widehat{\mathbf{x}_{2i}}$

Substituting this into the first two rows of the second equation gives us that: $(u_{2i}\mathbf{C}_{23} - \mathbf{C}_{21}) \cdot \tilde{\mathbf{w}}_i = 0$ and $(v_{2i}\mathbf{C}_{23} - \mathbf{C}_{22}) \cdot \tilde{\mathbf{w}}_i = 0$

Putting all of these equations into one matrix gives us:

$$\begin{bmatrix} u_{1i}\mathbf{C}_{13} - \mathbf{C}_{11} \\ v_{1i}\mathbf{C}_{13} - \mathbf{C}_{12} \\ u_{2i}\mathbf{C}_{23} - \mathbf{C}_{21} \\ v_{2i}\mathbf{C}_{23} - \mathbf{C}_{22} \end{bmatrix} \cdot \mathbf{w}_i = 0$$

(8)

which suggests to us that:

$$\mathbf{A} = \begin{bmatrix} u_{1i}\mathbf{C}_{13} - \mathbf{C}_{11} \\ v_{1i}\mathbf{C}_{13} - \mathbf{C}_{12} \\ u_{2i}\mathbf{C}_{23} - \mathbf{C}_{21} \\ v_{2i}\mathbf{C}_{23} - \mathbf{C}_{22} \end{bmatrix} \tag{9}$$

3.3 Q3.3

Here are the code snippets:

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

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

    M1 = np.hstack((np.eye(3), np.zeros((3, 1))))
    #Store all of the M2
    M2s = np.zeros((3, 4, 4))
    # print(M2_all)
    M2s = camera2(E)

    C1 = np.dot(K1, M1)
    best_err = np.inf

    for i in range(M2s.shape[2]):
        C2 = np.dot(K2, M2s[:, :, i])
        w, err = triangulate(C1, pts1, C2, pts2)

        if err < best_err:
            best_err = err
            M2= M2s[:, :,i]
            C2_opt = C2
            P = w

    np.savez(filename, M2 = M2, C2 = C2_opt, P = P)
    return M2, C2_opt, P
```

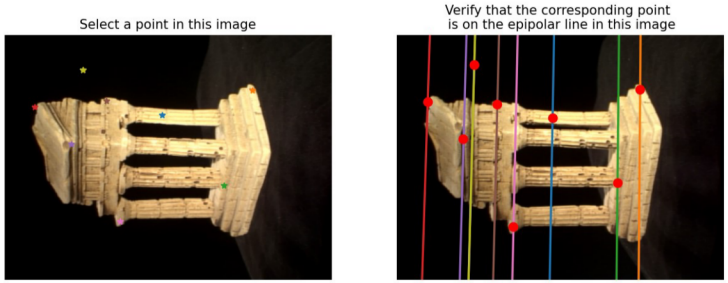
and triangulate is:

```python
def triangulate(C1, pts1, C2, pts2):
    n, _ = pts1.shape
    P = np.zeros((n, 3))
    P_homo = np.zeros((n, 4))
    for i in range(n):
```

```
            x_1 = pts1[i, 0]
            x_2 = pts2[i, 0]
            y_1 = pts1[i, 1]
            y_2 = pts2[i, 1]
            A1 = x_1*C1[2, :] - C1[0, :]
            A2 = y_1*C1[2, :] - C1[1, :]
            A3 = x_2*C2[2, :] - C2[0, :]
            A4 = y_2*C2[2, :] - C2[1, :]
            A = np.vstack((A1, A2, A3, A4))
            u, s, vh = np.linalg.svd(A)
            p = vh[-1, :]
            p = p/p[3]
            P[i, :] = p[0:3]
            P_homo[i, :] = p
        proj_p1 = np.matmul(C1, P_homo.T)
        l1 = proj_p1[-1, :]
        proj_p1 = proj_p1/l1
        proj_p2 = np.matmul(C2, P_homo.T)
        l2 = proj_p2[-1, :]
        proj_p2 = proj_p2/l2
        err1 = np.sum((proj_p1[[0, 1], :].T-pts1)**2)
        err2 = np.sum((proj_p2[[0, 1], :].T-pts2)**2)
        err = err1 + err2

        return P, err
```

4 3D Visualization

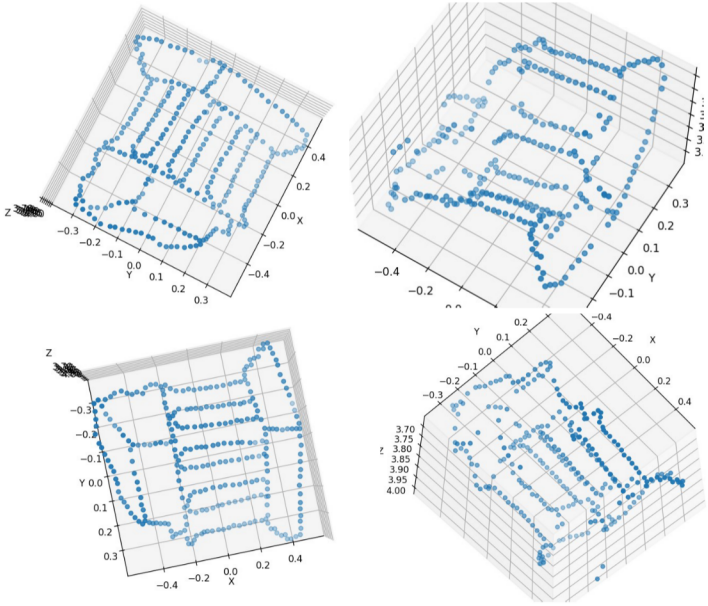
4.1 Q4.1




```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    x1 = int(x1)
    y1 = int(y1)
    # window size to adjust
    rect_size = 20
    im1sect = im1[(y1 - rect_size//2): (y1 + rect_size//2 + 1),
                  (x1 - rect_size//2): (x1 + rect_size//2 + 1), :]

    im2_h, im2_w, _ = im2.shape

    p1 = np.array([x1, y1, 1])

    epi_line = np.dot(F, p1)
    epi_l = epi_line / np.linalg.norm(epi_line)
    a, b, c = epi_l

    ep2_y = np.arange(im2_h)
    ep2_x = np.rint(-(epi_l[1]*ep2_y + epi_l[2]) / epi_l[0])

    # Do the gaussian thingy
    r_v = np.arange(-rect_size//2, rect_size//2+1, 1)
    r_x, r_y = np.meshgrid(r_v, r_v)
    d = 7
    weight = np.sum(np.dot((np.exp(-((r_x**2 + r_y**2) / (2 * (d**2))))), 1)/ np.sqrt(2*nı
    tol = 1e5
```

```
    for y2 in range((y1 - rect_size//2), (y1 + rect_size//2 + 1)):
        x2 = int((-b*y2-c)/a)
        if (x2 >= rect_size//2 and x2 + rect_size//2 < im2_w and y2 >= rect_size//2 and y2
            im2_sect = im2[y2-rect_size//2:y2+rect_size //2 +1, x2-rect_size//2:x2+rect_s
            err = np.linalg.norm((im1sect - im2_sect) * weight)
            if err < tol:
                tol = err
                x2_opt = x2
                y2_opt = y2
    return x2_opt, y2_opt
```

4.2 Q4.2



Here's the code:

```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2, C2):

    K1, K2 = intrinsics['K1'], intrinsics['K2']
    M1 = np.hstack((np.eye(3), np.zeros((3, 1))))
```

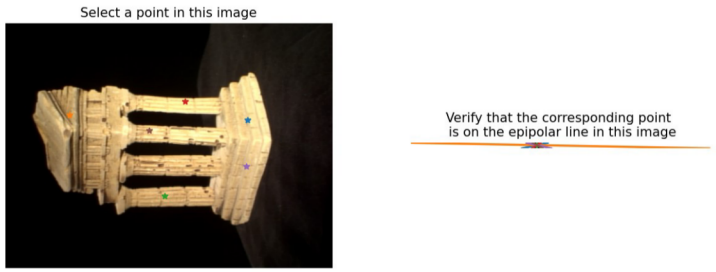
```
    E = essentialMatrix(F, K1, K2)
    x2 = np.empty((x1.shape[0], 1))
    y2 = np.empty((x1.shape[0], 1))
    for i in range(x1.shape[0]):
        correspondences = epipolarCorrespondence(im1, im2, F, x1[i], y1[i])
        x2[i] = correspondences[0]
        y2[i] = correspondences[1]
    temple_pts2 = np.hstack((x2, y2))
    M2s = camera2(E)
    C1 = np.dot(K1, M1)
    cur_err = np.inf
    for i in range(M2s.shape[2]):
        C2 = np.dot(K2, M2s[:, :, i])
        w, err = triangulate(C1, temple_pts1, C2, temple_pts2)

        if err<cur_err and np.min(w[:, 2])>=0:
            cur_err = err
            M2 = M2s[:, :, i]
            C2_opt = C2
            w_opt = w
    np.savez('q4_2', F = F, M1 = M1, M2 = M2, C1 = C1, C2 = C2)

    return w_opt
```

and the main function:

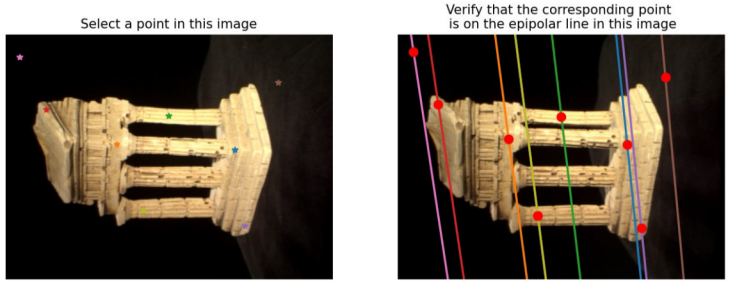
```
if __name__ == "__main__":

    temple_coords_path = np.load('data/templeCoords.npz')
    correspondence = np.load('data/some_corresp.npz') # Loading correspondences
    intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the camera
    K1, K2 = intrinsics['K1'], intrinsics['K2']
    pts1, pts2 = correspondence['pts1'], correspondence['pts2']
    im1 = plt.imread('data/im1.png')
    im2 = plt.imread('data/im2.png')
    # ----- TODO -----
    # YOUR CODE HERE
    x1 = temple_coords_path['x1']
    y1 = temple_coords_path['y1']
    temple_pts1 = np.hstack((x1, y1))


    F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
    w_opt = compute3D_pts(temple_pts1, intrinsics, F, im1, im2)
    fig = plt.figure()
```

```
20      res = Axes3D(fig)
21      res.set_xlim3d(np.min(w_opt[:,0]),np.max(w_opt[:,0]))
22      res.set_ylim3d(np.min(w_opt[:,1]),np.max(w_opt[:,1]))
23      res.set_zlim3d(np.min(w_opt[:,2]),np.max(w_opt[:,2]))
24      res.set_xlabel('X')
25      res.set_ylabel('Y')
26      res.set_zlabel('Z')
27      res.scatter(w_opt[:,0],w_opt[:,1],w_opt[:,2])
28      plt.show()
```

5 Bundle Adjustment



The above is without RANSAC. *Very* bad.
The following is with RANSAC:



As you can see it's significantly better.

The error metric for determining inliers is the basic L_1/absolute metric, i.e, $err = |\sum_i \tilde{\mathbf{x}_{2i}}^T \mathbf{F} \tilde{\mathbf{x}_{1i}} - 0|$ (since the fundamental matrix should satisfy $\tilde{\mathbf{x}_2}^T \mathbf{F} \tilde{\mathbf{x}_1} = 0$)

After playing around with the iterations and tolerance values, and discussing with some classmates (no 'official' collaborations, just offhand conversations), I decided on iterations = 1000 and tolerance = 0.8.

Code snippet:

```
def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    max_inliers = -1

    p1_hom = np.vstack((pts1.T, np.ones((1, pts1.shape[0]))))
    p2_hom = np.vstack((pts2.T, np.ones((1, pts1.shape[0]))))

    for idx in range(nIters):
        total_inliers = 0
        rand_idx = np.random.choice(pts1.shape[0], 8)
        rand1 = pts1[rand_idx, :]
        rand2 = pts2[rand_idx, :]

        F = eightpoint(rand1, rand2, M)
        pred_x2 = np.dot(F, p1_hom)
        pred_x2 = pred_x2 / np.sqrt(np.sum(pred_x2[:2, :]**2, axis=0))

        err = abs(np.sum(p2_hom*pred_x2, axis=0))
        n_inliers = err < tol
```

```
        # print(n_inliers)
        total_inliers = n_inliers[n_inliers.T].shape[0]
        if total_inliers > max_inliers:
            F_opt = F
            max_inliers = total_inliers
            inliers = n_inliers
        print(idx)

    return F_opt, inliers
```

5.1 Q5.2

Here are the code snippets:

```
def rodrigues(r):
    # Replace pass by your implementation
    d = r.shape[0]
    theta = np.linalg.norm(r)
    if theta == 0:
        R = np.identity(d)
    else:
        u = r/theta
        u1 = u[0]
        u2 = u[1]
        u3 = u[2]

        u_x = np.array([[0,-u3,u2],[u3,0,-u1],[-u2,u1,0]])
        R = np.identity(d)*np.cos(theta) +
        (1-np.cos(theta))*np.matmul(u,u.T) + np.sin(theta)*u_x

    return R

def invRodrigues(R):
    # Replace pass by your implementation
    A = (R - R.T)/2
    s = np.linalg.norm(np.array([A[2,1], A[0,2], A[1,0]]))
    c = (R[0,0]+R[1,1]+R[2,2]-1)/2
    r = []
    if s == 0 and c == 1:
        r = np.zeros((3,1))
    elif s == 0 and c == -1:
        Z = np.add(R, np.identity(3))
        r_1 = Z[:,0]
        r_2 = Z[:,1]
```

```
        r_3 = Z[:,2]
        if len(np.nonzero(r_1)) > 0:
            v = r_1
        elif len(np.nonzero(r_2)) > 0:
            v = r_2
        elif len(np.nonzero(r_3)) > 0:
            v = r_3
        u = v/np.linalg.norm(v)
        r_hat = u*np.pi
        r = s_half(r_hat)
    elif s != 0:
        u = np.array([A[2,1], A[0,2], A[1,0]])/s
        theta = arctan2(s,c)
        r = u*theta

    return r
```

5.2 Q5.3

The reprojection error without bundling is around 342.732 and after I got 15.823.
Code snippet:

```
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
    R2_0 = M2_init[:, 0:3]
    t2_0 = M2_init[:, 3]
    r2_0 = invRodrigues(R2_0)
    def fun(M): return (rodriguesResidual(K1, M1, p1, K2, p2, M))
    #obj_start = rodriguesResidual(K1, M1, p1, K2, p2, M2_init.flatten())
    x_0 = P_init.flatten()
    x_0 = np.append(x_0, r2_0.flatten())
    x_0 = np.append(x_0, t2_0.flatten())

    x_opt, _ = scipy.optimize.leastsq(fun, x_0)
    Pnew = x_opt[0:-6].reshape(-1, 3)
    rnew = x_opt[-6:-3].reshape(3, 1)
    tnew = x_opt[-3:].reshape(3, 1)

    R2 = rodrigues(rnew)
    M2 = np.hstack((R2, tnew))
```

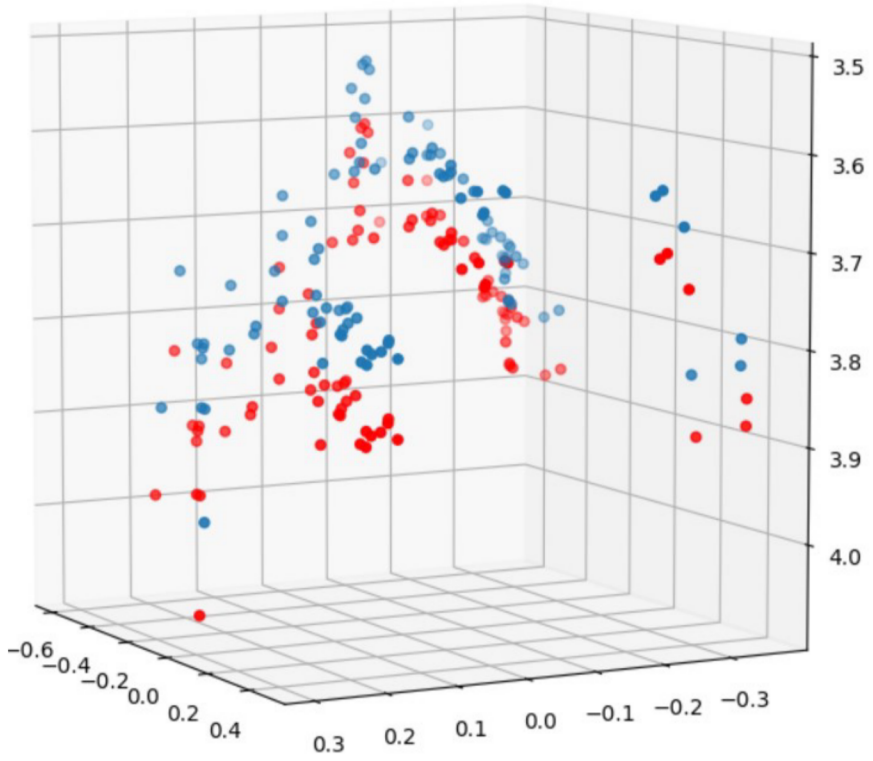
```
    #obj_end = rodriguesResidual(K1, M2, p1, K2, p2, M2.flatten())
    return M2, P2, obj_start, obj_end
```

And running and finding the two (the code is a bit all over the place for this one):

```
    E = essentialMatrix(F, K1, K2)
    M1 = np.hstack((np.eye(3), np.zeros((3, 1))))
    M2_all = camera2(E)

    C1 = np.dot(K1, M1)
    err_min = np.inf

    for i in range(M2_all.shape[2]):
         M2_i = M2_all[:, :, i]
         C2 = np.dot(K2, M2_i)
         w, err = triangulate(C1, noisy_pts1, C2, noisy_pts2)

         if err < err_min:
             err_val = err
             M2 = M2_i
             C2_opt = C2
             w_best = w

    P_init, err_orig = triangulate(C1, noisy_pts1, C2_opt, noisy_pts2)
    print('Original reprojection error: ', err_orig)


    #5.3
    M2_opt, P2, _, _ = bundleAdjustment(K1, M1, noisy_pts1, K2, M2, noisy_pts2, P_init)

    C2_opt = np.dot(K2, M2_opt)
    w_hom = np.hstack((P2, np.ones([P2.shape[0], 1])))
    C2 = np.dot(K2, M2)
    err_opt = 0

    for i in range(noisy_pts1[inliers, :].shape[0]):
        pts1hat = np.dot(C1, w_hom[i, :].T)
        pts2hat = np.dot(C2_opt, w_hom[i, :].T)

        # Normalizing
        p1_hat_norm = (np.divide(pts1hat[0:2], pts1hat[2])).T
        p2_hat_norm = (np.divide(pts2hat[0:2], pts2hat[2])).T
        err1 = np.square(noisy_pts1[:, 0] - p1_hat_norm[0]) + \
            np.square(noisy_pts1[:, 1] - p1_hat_norm[0])
```

```
        err2 = np.square(noisy_pts2[:, 0] - p2_hat_norm[0]) + \
             np.square(noisy_pts2[:, 1] - p2_hat_norm[0])
        err_opt += np.sum((p1_hat_norm - noisy_pts1[i])
                          ** 2 + (p2_hat_norm - noisy_pts2[i])**2)

    print('Error with optimized 3D points: ', err_opt)

    plot_3D_dual(P_init, P2)
```

Here is what the graph looks like:



6 Extra Credit

6.1 Q6.1 and Q6.2

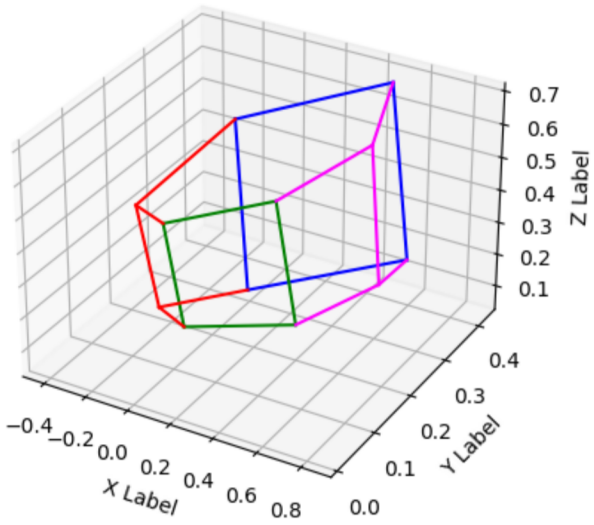
As suggested in the hint I used the triangulate function from earlier to calculate three different **w** and choose the one with the least error. I didn't have time to do much with it but here are random snippets and pictures.

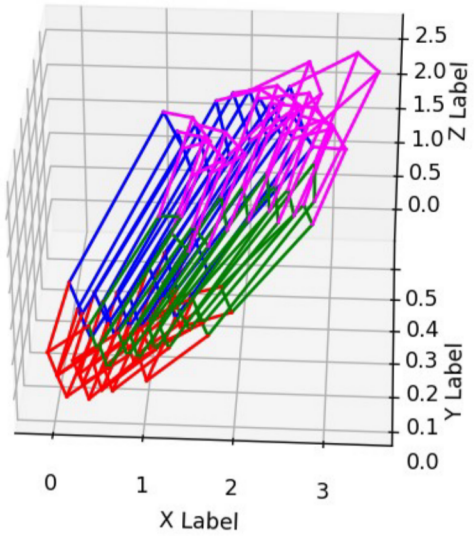
```
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres = 100):
    # Replace pass by your implementation
    P, err = triangulate(C1, pts1[:, :2], C2, pts2[:, :2])
    return P, err
```

And:

```
for i in range(10):
    time = np.load('../data/q6/time'+str(i)+'.npz')
    pts1 = time['pts1']
    pts2 = time['pts2']
    pts3 = time['pts3']
    M1_0 = time['M1']
    M2_0 = time['M2']
    M3_0 = time['M3']
    K1_0 = time['K1']
    K2_0 = time['K2']
    K3_0 = time['K3']
    C1_0 = np.dot(K1_0, M1_0)
    C2_0 = np.dot(K1_0, M2_0)
    C3_0 = np.dot(K1_0, M3_0)
    Thres = 200
    P_mv, err_mv = MultiviewReconstruction(
        C1_0, pts1, C2_0, pts2, C3_0, pts3, Thres)
    M2_opt, pts_3d = bundleAdjustment(
        K2_0, M2_0, pts2[:, :2], K3_0, M3_0, pts3[:, :2], P_mv)
    num_points = pts_3d.shape[0]
    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])
    np.set_printoptions(threshold=1e6, suppress=True)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
```

```
30  ax.set_zlabel('Z Label')
31  plt.show()
```




2.5
2.0
1.5
1.0
0.5
0.0
Z Label
0.5
0.4
0.3
0.2
0.1
0.0
Y Label
0
1
2
3
X Label