16720-B COMPUTER-VISION HOMEWORK

Sagar Sachdev (sagarsac)

Q1.1

$$x_2 F x_1 = 0$$

For $x_1 = x_2 = [0,0,1]^T$

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

$$=> f_{33} = 0$$

Therefore, proved.

Q1.2

Since the translation is in the x-axis, it can be reflected as:

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

Cross-product matrix can be written as:

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

$$\begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

$$R = I$$

$$E = TR = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} * I$$

Given that:

$$[x_1 \quad y_1 \quad 1] * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = [0 \quad t_x \quad -y_1 t_x].T$$

and:

$$[x_2 \quad y_2 \quad 1] * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = [0 \quad t_x \quad -y_2 t_x].T$$

We can compute the cross product, of the epipolar lines:

$$\begin{bmatrix} 0 \\ t_x \\ -y_1 t_x \end{bmatrix} \times \begin{bmatrix} 0 \\ t_x \\ -y_2 t_x \end{bmatrix} = 0$$

Hence the two epipolar lines are parallel.

1.3

$$\omega_1 = K(R_1\omega + t_1)$$

Making omega the subject of the formula and treating this as a linear algebra problem, we get:

$$R_{rel} = KR_2{R_1}^{-1}K^{-1}$$

$$t_{rel} = -KR_2{R_1}^{-1}t_1 + Kt_2$$

$$E = t_{rel} \times R_{rel}$$

$$F = (K^{-1})^T E K^{-1}$$

$$F = (K^{-1})^T (t_{rel} \times R_{rel}) K^{-1}$$

1.4

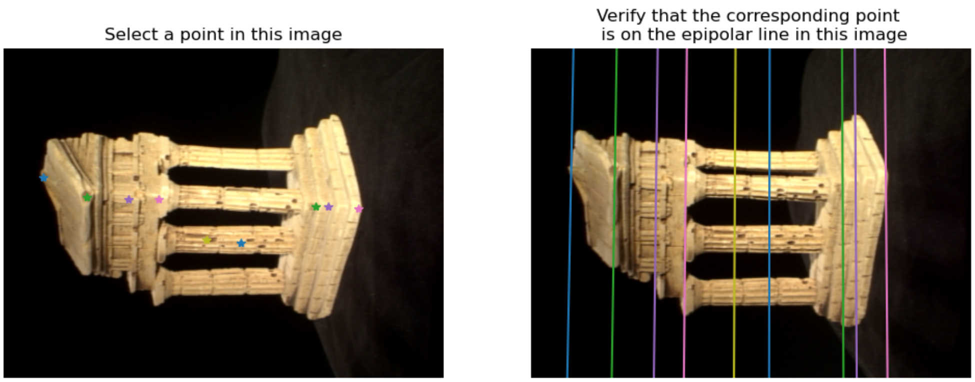
$$R_{rel} = I$$

$$t_{rel} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

$$F = (K^{-1})^T \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} K^{-1}$$

Therefore the fundamental matrix is a skew-symmetric matrix

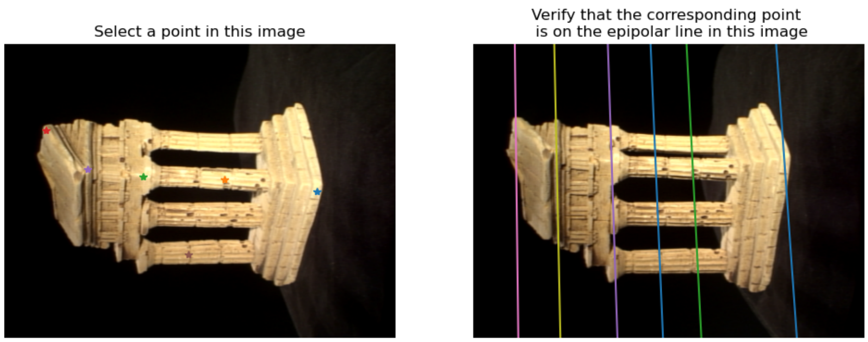
Q




```python
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation
    A_init = np.zeros((pts1.shape[0],9))
    # Normalization step:
    pts1_x = pts1[:,0]/M
    pts1_y = pts1[:,1]/M
    pts2_x = pts2[:,0]/M
    pts2_y = pts2[:,1]/M
    #Applying 8-point algorithm
    A = np.asarray([pts1_x*pts2_x, pts1_x*pts2_y, pts1_x,pts1_y*pts2_x,pts1_y*pts2_y,pts1_y,pts2_x,pts2_y,np.ones(pts1.shape[0])]).T
    U,S,Vh = np.linalg.svd(A)
    # print('U',U)
    # print('Vh',Vh)

    F = refineF(np.reshape(Vh[-1],(3,3)), pts1/M, pts2/M)
    T = np.asarray([[1/M, 0, 0],
                    [0, 1/M, 0],
                    [0, 0, 1]])
    F_unnormalized = T.T @ F @ T
    return F_unnormalized
```

$$F = \begin{pmatrix} 1.08539884e-06 & 2.94131462e-05 & -2.89022951e-01 \\ 1.66543425e-05 & 7.92928733e-07 & -6.45376750e-03 \\ 2.77651959e-01 & 2.17130301e-03 & 1.00000000e+00 \end{pmatrix}$$

Q2.2




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

    Farray = []
    # ----- TODO -----
    # YOUR CODE HERE
    pts1 = pts1/M
    pts2 = pts2/M

    pts1_x = pts1[:,0]
    pts1_y = pts1[:,1]
    pts2_x = pts2[:,0]
    pts2_y = pts2[:,1]

    A = np.asarray([pts1_x*pts2_x,pts1_x*pts2_y,pts1_x,pts1_y*pts2_x,pts1_y*pts2_y,pts1_y,pts2_x,pts2_y,np.ones(pts1.shape[0])]).T
    U,S,Vh = np.linalg.svd(A)

    #Computing the null spaces
    F1 = np.reshape(Vh[-1,:],(3,3))
    F2 = np.reshape(Vh[-2,:],(3,3))

    #Computing the cubic polynomial
    # a = sym.Symbol('a')
    fun = lambda alpha: np.linalg.det((alpha*F1)+(1-alpha)*F2)
    a0 = fun(0)
    a1 = (2/3)*(fun(1)-fun(-1))-((1/12)*(fun(2)-fun(-2)))
    a2 = (1/2)*(fun(1)+fun(-1)) - a0
    a3 = (fun(1)-fun(-1))*(1/2) - a1
    a_root = np.polynomial.polynomial.polyroots((a0,a1,a2,a3))
    # print(a_root)
    real_root = np.isreal(a_root)
    a_root = np.real(a_root[real_root])
    T = np.asarray([[1/M, 0, 0],[0, 1/M, 0],[0, 0, 1]])
    Farray = []
    for a in a_root:
        F = a*F1 +(1-a)*F2
        # F = refineF(F,pts1,pts2)
        F_unscaled = (T.T @ F @ T).T
        Farray.append(F_unscaled/F_unscaled[2,2])
    # raise NotImplementedError()
    return Farray
```

```
F [[ 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]]
```

Q3.1

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    E = K2.T @ F @ K1
    U,S,Vh = np.linalg.svd(E)
    E = E/S[0]
    return E
```

3.2

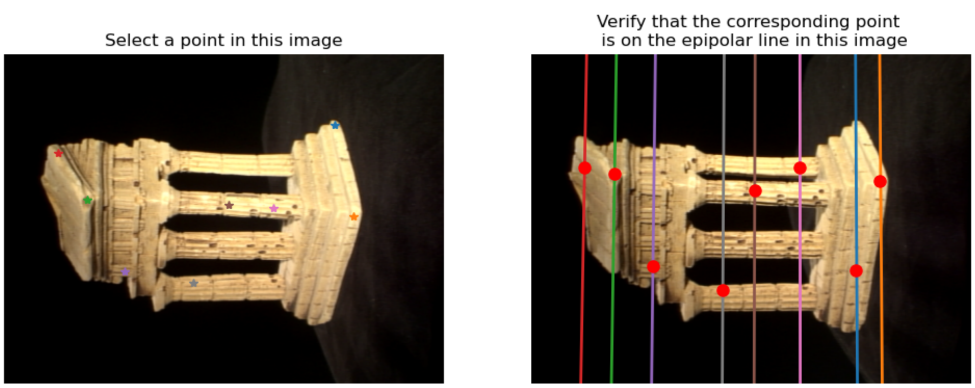
```
A
array([[-1520.4       ,     0.        ,  -145.32      ,     0.        ],
       [    0.        , -1525.9       ,   -15.87      ,     0.        ],
       [ 1518.40741129,    56.12604337,   154.99310624,   -19.74820187],
       [   66.66369177, -1523.35657744,   -67.80971217,  1522.66948847]])
```

3.3

```python
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation
    pts1_x = pts1[:,0]
    pts1_y = pts1[:,1]
    pts2_x = pts2[:,0]
    pts2_y = pts2[:,1]

    # pts1/pts2 should be N x 2

    C11 = C1[0]
    C12 = C1[1]
    C13 = C1[2]

    C21 = C2[0]
    C22 = C2[1]
    C23 = C2[2]

    # P = np.empty(())
    reproj_err = []
    # w_i = []
    P = []
    for i in range(pts1.shape[0]):
        x1,y1 = pts1[i,0],pts1[i,1]
        x2,y2 = pts2[i,0],pts2[i,1]

        A1 = C13*x1 - C11
        A2 = C13*y1 - C12
        A3 = C23*x2 - C21
        A4 = C23*y2 - C22

        A = np.vstack((A1,A2,A3,A4))
        U,S,Vh = np.linalg.svd(A)

        w = Vh[-1]
        w = w/w[-1]
        P.append(w)

        # w_i = np.ones((1,pts1.shape[0]))
        # w_i = np.asarray([[P.T],[w_i]])

        new_pts1 = (C1 @ w)
        new_pts2 = (C2 @ w)

        points1 = np.insert(pts1[i],2,1)
        points2 = np.insert(pts2[i],2,1)

        new_pts1 = (new_pts1/new_pts1[-1])
        new_pts2 = (new_pts2/new_pts2[-1])

        # points1 = (3,)
        # new_pts1 = (3,)

        # points2 = (3,)
        # new_pts2 = (3,)

        err = np.linalg.norm(points1 - new_pts1) + np.linalg.norm(points2 - new_pts2)
    err = np.sum(err)
    # reproj_err =
        # reproj_err = np.sum(reproj_err)
    P = np.array(P)
    return P, err
```

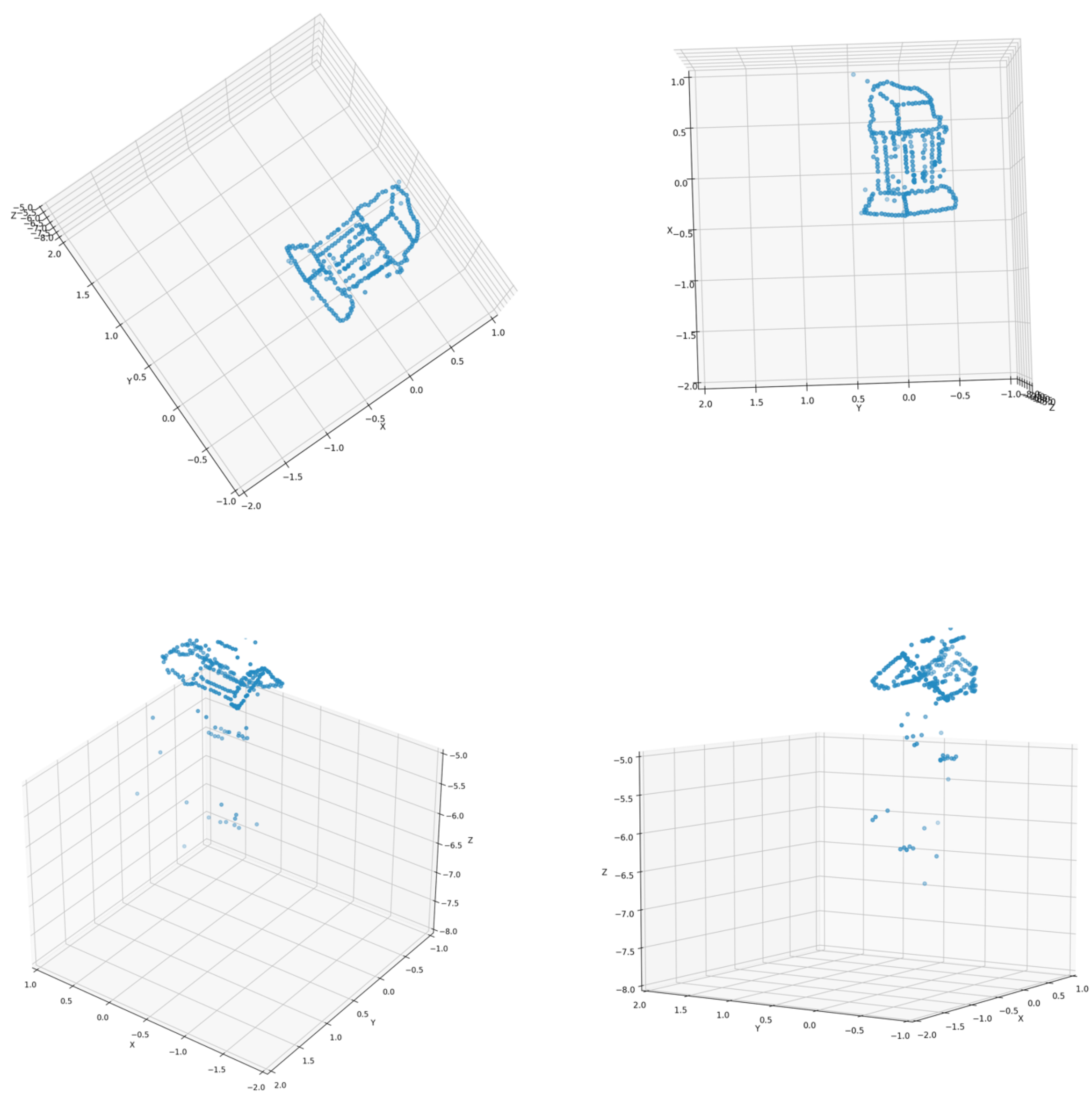
```python
def findM2(F, pts1, pts2, K1, K2, filename = 'q3_3.npz'):
    '''
    Q2.2: Function to find the camera2's projective matrix given correspondences
        Input:  F, the pre-computed fundamental matrix
                pts1, the Nx2 matrix with the 2D image coordinates per row
                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'.

    '''
    N = pts1.shape[0]
    M2 = np.zeros((3,4))
    C2 = np.zeros((3,4))
    P = np.zeros((N,3))
    M1 = np.hstack((np.eye(3,3),np.zeros((3,1))))
    # K1, K2 = intrinsics['K1'], intrinsics['K2']
    C1 = K1 @ M1
    E = essentialMatrix(F,K1,K2)
    M2_matrices = camera2(E)
    for i in range(4):
        C2 = K2 @ M2_matrices[:,:,i]
        w, err = triangulate(C1, pts1, C2, pts2)

        if min(pts1[:,0]) >= 0 and min(pts2[:,1]) >= 0:
            M2 = M2_matrices[:,:,i]
            P = w
    C2 = K2 @ M2
    return M2, C2, P

```

Q4.1



```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Replace pass by your implementation
    window_size = 5
    # x1 = np.flatten(x1)
    # y1 = np.flatten(y1)


    # Initializing value of x-axis and y-axis
    # in the range -1 to 1
    x, y = np.meshgrid(np.linspace(-1,1,1+(window_size*2)), np.linspace(-1,1,1+(window_size*2)))
    dst = np.sqrt(x*x+y*y)
    # Initializing sigma and muu
    sigma = 1
    muu = 0.000
    # Calculating Gaussian array
    gauss = np.exp(-( (dst-muu)**2 / ( 2.0 * sigma**2 ) ) )
    gauss = gauss/np.sum(gauss)

    gauss3 = np.stack((gauss,gauss, gauss),axis=2)
    rows_min1 = y1 - window_size
    rows_max1 = y1 + window_size+1

    col_min1 = x1 - window_size
    col_max1 = x1 + window_size+1

    P1 = []
    w1 = im1[rows_min1:rows_max1, col_min1:col_max1]

    P1 = np.asarray([x1,y1,1])

    vector = F @ P1

    rows = np.arange(window_size,(im2.shape[0]-window_size))
    cols = (-(vector[1]/vector[0])*rows + (-vector[2]/vector[0])).astype(int)


    minimum_dist = 100000
    for i in range(len(cols)):
        rows_min2 = rows[i] - window_size
        rows_max2 = rows[i] + window_size + 1

        col_min2 = cols[i] - window_size
        col_max2 = cols[i] + window_size + 1
        w2 = im2[rows_min2:rows_max2, col_min2:col_max2]

        w_err = (w1-w2)*gauss3
        diffd = []

        diff1 = np.linalg.norm(w_err)
        p2 = np.asarray([rows[i],cols[i]])
        if diff1 <=minimum_dist:
            p2 = p2
            y2 = p2[0]
            x2 = p2[1]
            minimum_dist = diff1

    return x2,y2
```

Q4.2



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

    # ----- TODO -----
    # YOUR CODE HERE
    # raise NotImplementedError()
    x1 = temple_pts1['x1']
    y1 = temple_pts1['y1']

    x1 = x1[:,0]
    y1 = y1[:,0]
    K1, K2 = intrinsics['K1'], intrinsics['K2']

    x2 = np.zeros(x1.shape[0])
    y2 = np.zeros(x1.shape[0])
    x_2 = []
    y_2 = []
    for i in range(x1.shape[0]):
        x2,y2 = epipolarCorrespondence(im1, im2, F, x1[i], y1[i])
        x_2.append(x2)
        y_2.append(y2)


    x_2 = np.asarray(x_2)
    y_2 = np.asarray(y_2)
    pts1 = np.asarray([x1,y1]).T
    pts2 = np.asarray([x_2,y_2]).T
    # pts1 = np.asarray([x1,y1])
    # pts2 = np.asarray([x_2,y_2])

    M2,C2,P = findM2(F,pts1,pts2,K1,K2)
    # P = np.asarray([x2,y2,1])


    return M2,C2,P
```

Q5.1

```python
def ransacF(pts1, pts2, M, nIters=100, tol=1):
    # Replace pass by your implementation
    N = pts1.shape[0]
    max_inliers = -1
    F_array = []
    temp_inliers = None
    for i in range(nIters):
        idx = np.random.choice(pts1.shape[0],8,False)
        F = eightpoint(pts1[idx],pts2[idx],M)
        F_array.append(F)

        pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)
        error = calc_epi_error(pts1_homogenous,pts2_homogenous,F)
        temp_inliers = error<tol
        if temp_inliers[temp_inliers].shape[0] > max_inliers:
            max_inliers = temp_inliers[temp_inliers].shape[0]
            F = F
            inliers = temp_inliers
    F = F/F[2,2]
    return F, inliers
```

```python
def rodrigues(r):
    # Replace pass by your implementation
    theta = np.linalg.norm(r)
    I = np.eye(3)
    if theta == 0:
        R = I
    else:
        u = r/theta
        u = u.reshape(3,1)
        u_x = np.asarray([[0,-u[2], u[1]],[u[2], 0, -u[0]],[-u[1], u[0], 0]],dtype= object)
        R = I*np.cos(theta) + (1-np.cos(theta))* (u@u.T) + u_x*np.sin(theta)
    return R
```

```python
def invRodrigues(R):
    # Replace pass by your implementation
    A = (R-R.T)/2
    I = np.eye(3)
    rho = np.asarray([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
    theta = np.arctan2(s,c)
    if s == 0 and c == 0:
        r = np.zeros((1,3))
    elif s == 0 and c == -1:
        v = R+I
        if np.linalg.norm(v[:,0]) != 0:
            v = v[:,0]
        elif np.linalg.norm(v[:,1]) != 0:
            v = v[:,1]
        else:
            v = v[:,2]
        u = v/np.linalg.norm(v,2)
        r = u*np.pi
        if np.linalg.norm(r) == np.pi and ((r[0,0] == 0 and r[1,0] == 0 and r[2,0]< 0) or (r[0,0] == 0 and r[1,0] < 0) or r[0,0]< 0):
            r = -r
        else:
            r = r
    elif np.sin(theta) != 0:
        u = rho/s
        r = u*theta
    return r
```

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation
    t = x[-3,:].reshape((3,1))
    r = x[-6:-3].reshape((3,1))
    P = x[:,-6].reshape(3,1)
    R = rodrigues(r)
    C1 = K1 @ M1
    M2 = np.hstack((R,t))
    C2 = K2 @ M2
    p1_hat1 = C1 @ P
    p1_hat = p1_hat1/p1_hat1[2,:]
    p2_hat2 = C2 @ P
    p2_hat = p2_hat2/p2_hat2[2,:]
    residuals = np.concatenate([(p1-p1_hat).reshape([-1]), (p2-p2_hat).reshape([-1])])
    return residuals
```

```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
    # raise NotImplementedError()
    R = M2_init[:,:3]
    t = M2_init[:,3]
    r = invRodrigues(R)


    vec = np.hstack((P_init.flatten(),r.flatten(),t.flatten()))
    func = lambda x: rodriguesResidual(K1, M1, p1, K2, p2, x)

    #find optimized vector
    vec_optim,_ = scipy.optimize.least_squares(func, vec)

    P = vec_optim[0:-6]
    r1 = vec_optim[-6:-3]
    t1 = vec_optim[-3:]

    R1 = rodrigues(r1)
    M2 = np.hstack((R,t1.reshape(3,1)))
    return M2, P, obj_start, obj_end
```

NOTE:
I think there is something incorrect in bundleAdjustment hence why I do not have an output.

ACKNOWLEDGEMENTS:

1) My friend from my lab Bassam Bikdash helped me (and served as office hours) for a bit of this assignment since I did not have access to the Office Hours due to personal circumstances highlighted to Prof Ramanan
2) https://www.geeksforgeeks.org/how-to-generate-2-d-gaussian-array-using-numpy/ was used for Gaussian weighting portion of the epipolar correspondences