16-720 A Computer Vision

Assignment 4

Sourojit Saha

sourojis

Collaborated with Sohan Kulkarni(sohank) and Arjun Sengupta(arjunsen)

1. Theory

1.1

We know that:

$$\begin{bmatrix} x_1x_1^{'} & x_1y_1^{'} & x_1 & y_1x_1^{'} & y_1y_1^{'} & y_1 & x_1^{'} & y_1^1 & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \tag{1}$$

Given: $(x_1, y_1) = (0, 0)$ and $(x_1^{'}, y_1^{'}) = (0, 0)$. Substituting in the above equation, we get:

$$0 + (1)(F_{33}) = 0 \implies F_{33} = 0 \tag{2}$$

1.2

Given that both the cameras have the same intrinsics, i.e. $K_1 = K_2 = K(say)$, and there is no relative rotation, i.e. $R = I_{3\times3}$. Let there be a relative translation of second camera w.r.t first camera given by $T = \begin{bmatrix} T_x & T_y & T_z \end{bmatrix}^T = \begin{bmatrix} T_x & 0 & 0 \end{bmatrix}^T$ because there is pure translation along $X - axis$. The essential matrix E is then given by:

$$E = TR = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T_x \\ 0 & T_x & 0 \end{bmatrix} \tag{3}$$

Hence the equation of line L can be given by:

$$L = EX \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -T_x \\ T_xy \end{bmatrix} \tag{4}$$

This is the equation of line parallel to $X - axis$ as the coefficient of $x = 0$.

1.3

Suppose at time t_1, the rotation and translation matrices are R_1 and T_1. And at time t_2, the rotation and translation matrices are R_2 and T_2. For any 3D point $X = \begin{bmatrix} x & y & z \end{bmatrix}^T$, the point in frame 1 is given by:

$$x_1 = R_1 X + T_1 \implies X = R_1^{-1}(x_1 - T_1) \tag{5}$$

$$x_2 = R_2 X + T_2 \implies x_2 = R_2(R_1^{-1}(x_1 - T_1))X + T_2 = (R_2 R_1^{-1})X - (R_2 R_1^{-1} T_1 + T_2) \tag{6}$$

From above equations we can see that:

$$R_{rel} = R_2 R_1^{-1} \tag{7}$$

$$T_{rel} = -(R_2 R_1^{-1} T_1 + T_2) \tag{8}$$

Given that camera intrinsic K is known. We can write E and F as:

$$E = \hat{T_{rel}} \times R_{rel} = -(R_2 R_1^{-1} T_1 + T_2) \times R_2 R_1^{-1} \tag{9}$$

$$F = K^{-T} E K^{-1} = K^{-T}(-(R_2 R_1^{-1} T_1 + T_2) \times R_2 R_1^{-1}) K^{-1} \tag{10}$$

1.4

The problem comes down to proving that the fundamental matrix between the two poses is a skew-symmetric matrix. Hence, first finding the fundamental matrix. Assuming no rotation and only pure translation. Therefore, rotation matrix $R = I_{3\times 3}$. Let the translation be $T = \begin{bmatrix} T_x & T_y & T_z \end{bmatrix}^T$. Hence, the essential Matrix E is given by:

$$E = \hat{T} \times R = \hat{T} \times I \tag{11}$$

Hence,

$$E = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \tag{12}$$

We see that E is a skew symmetric matrix, as $E^T = -E$. Since the camera intrinsic will remain the same (same camera), we can find the fundamental matrix F as:

$$F = K^{-T} E K^{-1} \tag{13}$$

Where K is the intrinsic matrix of the camera.Taking transpose of F:

$$F^T = (K^{-T} E K^{-1})^T = -K^{-T} E K^{-1} = -F \tag{14}$$

We see that F is a skew-symmetric matrix.

2. Fundamental Matrix Estimation

2.1 The Eight Point Algorithm

```python
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation
    pts1 = np.hstack((pts1, np.ones((pts1.shape[0], 1))))
    pts2 = np.hstack((pts2, np.ones((pts2.shape[0], 1))))

    pts1_normalised = (T1@pts1.T).T
    pts2_normalised = (T1@pts2.T).T

    A = np.zeros((0,9))
    for i in range(pts1_normalised.shape[0]):
        x1, y1, dummy = pts1_normalised[i,:]
        x2, y2, dummy = pts2_normalised[i,:]
        a = np.asarray([[x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]])
        A = np.vstack((A, a))

    U, sigma, Vt = np.linalg.svd(A)
    F = Vt[-1,:].reshape(3,3).T
    # u, sigma, vt = np.linalg.svd(F, full_matrices=True)
    # # print(sigma.shape)
    # Sigma = np.zeros((len(sigma), len(sigma)))
    # for i in range(Sigma.shape[0]):
    #     for j in range(Sigma.shape[0]):
    #         if i == j:
    #             Sigma[i,j] = sigma[i]
    # # print(Sigma)
    # Sigma[2,2] = 0
    # F = u@Sigma@vt
    # print(np.linalg.matrix_rank(F))
    # print(pts1_normalised)
    F = _singularize(F)
    F = refineF(F, pts1_normalised[:,0:2], pts2_normalised[:,0:2])
    F = T1.T@F@T1
    F = F/F[2,2]
    return F
```

```
F:
[[-2.19299582e-07  2.95926445e-05 -2.51886343e-01]
 [ 1.28064547e-05 -6.64493709e-07  2.63771740e-03]
 [ 2.42229086e-01 -6.82585550e-03  1.00000000e+00]]
```

Figure 1: F matrix

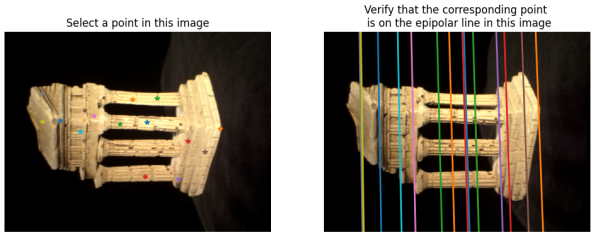

Figure 2: Result for eight-point-algorithm

3. Metric Reconstruction

3.1 Essential Matrix

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    E = K2.T@F@K1
    E = E/E[2,2]
    return E
```

3.2 Triangulate

The expression for A_i matrix is a follows:

$$\begin{bmatrix} yp_3^T - p_2^T \\ p_1^T - xp_3^T \\ y'p'_{3T} - p'_2{}^T \\ p'_{1T} - x'p'_3{}^T \end{bmatrix} X = 0 \tag{15}$$

Where:

$$A_i = \begin{bmatrix} yp_3^T - p_2^T \\ p_1^T - xp_3^T \\ y'p'_{3T} - p'_{2T} \\ p'_{1T} - x'p'_3{}^T \end{bmatrix} \tag{16}$$

Where p_i^T are the rows of the camera matrix of camera 1, and $p_i'^T$ are the rows of the camera matrix of camera 2.

```python
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation
    err = np.zeros((0,1))
    P = np.zeros((0,4))
    pts1 = np.hstack((pts1, np.ones((pts1.shape[0],1))))
    pts2 = np.hstack((pts2, np.ones((pts2.shape[0],1))))
    for i in range(pts1.shape[0]):
        x1 = pts1[i, 0]
        y1 = pts1[i, 1]
        x2 = pts2[i, 0]
        y2 = pts2[i, 1]

        A = np.array([y1*C1[2,:] - C1[1,:],
                      C1[0,:] - x1*C1[2,:],
                      y2*C2[2,:] - C2[1,:],
                      C2[0,:] - x2*C2[2,:]])

        u, sigma, vt = np.linalg.svd(A)
        X = vt[-1,:].reshape((4,1))
        # print(X)
        X = X/X[-1]
        # print(X.T)
        X1 = C1@X
        X2 = C2@X
```

```
        X1 = X1/X1[-1]
        X2 = X2/X2[-1]

        err1 = pts1[i,:]-X1.T
        err2 = pts2[i,:]-X2.T
        # print(err1, err2)
        total_err = err1@err1.T + err2@err2.T
        err = np.vstack((err, total_err))

        P = np.vstack((P, X.T))
        err = np.sum(err)

    return P, err
```

3.3 Find M2

```
def findM2(F, pts1, pts2, intrinsics, filename = 'q3_3.npz'):
    # intrinsics = np.load('../data/intrinsics.npz') # Loading the
    intrinscis of the camera
    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 = K1@M1
    best_err = float('inf')
    best_M2 = None
    best_C2 = None
    best_P = None
    for i in range(M2s.shape[2]):
        M2 = M2s[:,:,i]
        C2 = K2@M2
        P, err = triangulate(C1, pts1, C2, pts2)
        print(i, err)
        if err < best_err and np.all(P[:,2]>=0):
            print(i)
            best_err = err
            best_M2 = M2
            best_C2 = C2
            best_P = P

    return best_M2, best_C2, best_P
```

4. 3D Visualization

4.1 Epipolar Correspondence

```
def epipolarCorrespondence(im1, im2, F, x1, y1):
    im1 = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
    im2 = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)
    line = F@(np.array([x1,y1,1]).T)
    window_size = 20

    im1x = np.arange(0, im1.shape[0])
    im1y = np.arange(0, im1.shape[1])
    spline_im1 = RectBivariateSpline(im1x, im1y, im1)

    imx2 = np.arange(0, im2.shape[0])
    imy2 = np.arange(0, im2.shape[1])
    spline_im2 = RectBivariateSpline(imx2, imy2, im2)

    im1x1 =  x1-(window_size//2)
    im1y1 =  y1-(window_size//2)
    im1x2 =  x1+(window_size//2)
    im1y2 =  y1+(window_size//2)

    x11 = np.arange(im1x1,im1x2)
    y11 = np.arange(im1y1,im1y2)
    X,Y = np.meshgrid(x11,y11)
    im1_template = spline_im1.ev(Y, X)

    coordinates = np.zeros((0,2))
    error = np.zeros((0,1))
    for i in range(window_size//2, im2.shape[0]-window_size//2):
        y = i + (window_size*0.5)
        x = -(line[1] * y + line[2])/line[0]
        coord = np.array([[x,y]])
        im2x1 =  x-(window_size//2)
        im2y1 =  y-(window_size//2)
        im2x2 =  x+(window_size//2)
        im2y2 =  y+(window_size//2)

        x22 = np.arange(im2x1,im2x2)
        y22 = np.arange(im2y1,im2y2)
        X,Y = np.meshgrid(x22,y22)
```

```
        im2_template = spline_im2.ev(Y, X)

        min_row = np.min([im1_template.shape[0], im2_template.shape[0]])
        min_col = np.min([im1_template.shape[1], im2_template.shape[1]])
        im1_template_min = im1_template[0:min_row, 0:min_col]
        im2_template_min = im2_template[0:min_row, 0:min_col]

        err = np.sum((im1_template_min - im2_template_min)**2)
        coordinates = np.vstack((coordinates, coord))
        error = np.vstack((error, err))
    best_match = np.argmin(error)
    best_coord = coordinates[best_match,:]
    x2 = best_coord[0]
    y2 = best_coord[1]

    return x2, y2
```

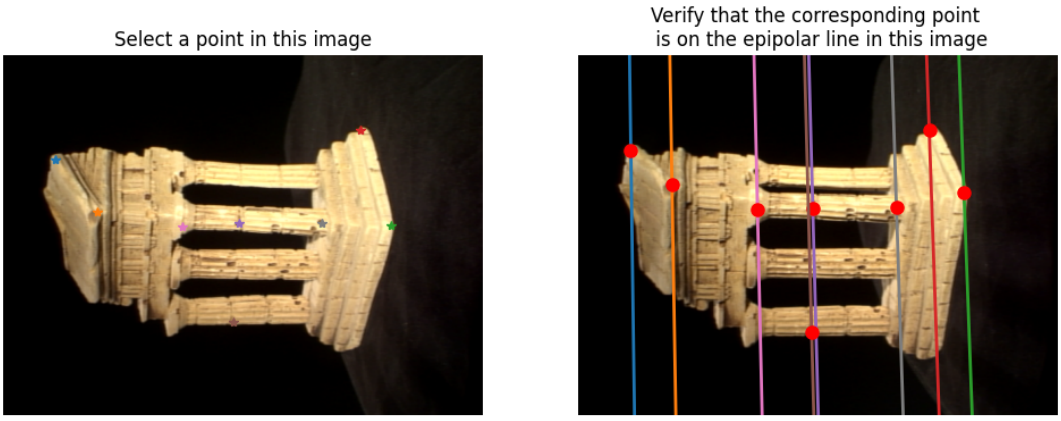

Figure 3: Epipolar Correspondence

4.2 3D Reconstruction

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

    K1, K2 = intrinsics['K1'], intrinsics['K2']
    pts2 = np.zeros((0,2))
    for i in range(temple_pts1.shape[0]):
        x2, y2 = epipolarCorrespondence(im1, im2, F, temple_pts1[i,0],
   temple_pts1[i,1])
        a = np.array([[x2,y2]])
        pts2 = np.vstack((pts2, a))
    dummy1, dummy2, P = findM2(F, temple_pts1, pts2, intrinsics)

    return P
```

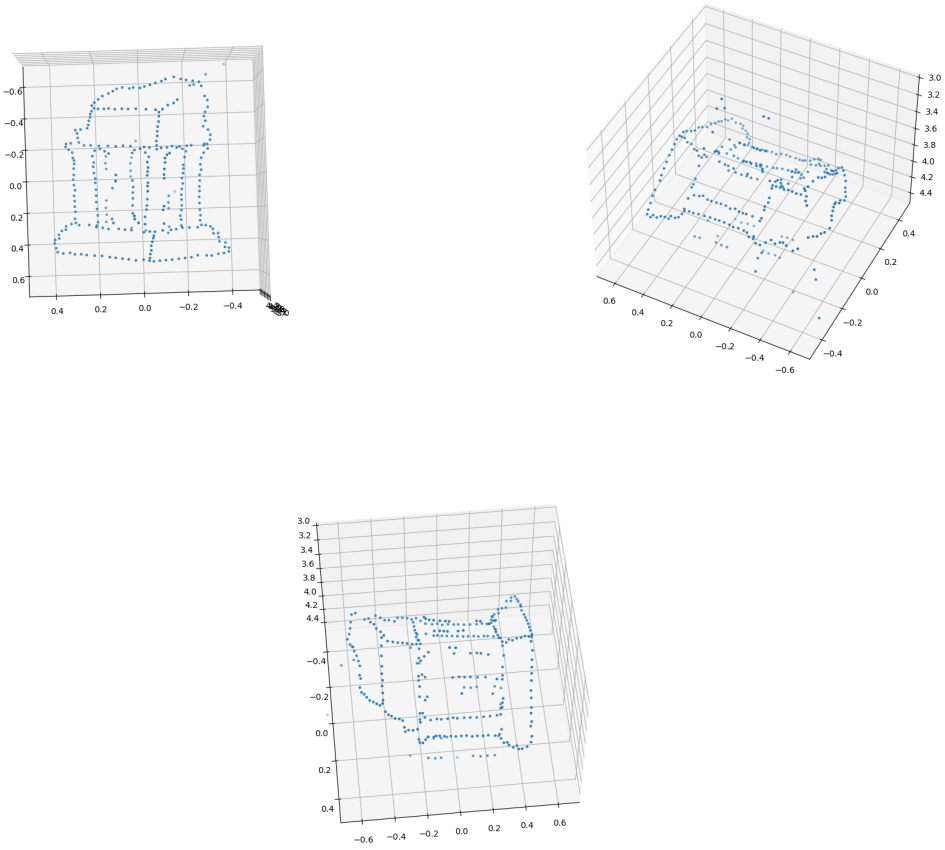

Figure 4: Different views of 3D reconstruction

5. Bundle Adjustment (Extra Credit)

5.1 RANSAC for Fundamental Matrix Recovery

```
def ransacF(pts1, pts2, M, nIters=200, tol=50):
    N = pts1.shape[0] #pts1 and pts2 are same shape
    pts1_homogenous = np.hstack((pts1, np.ones((N, 1))))
    pts2_homogenous = np.hstack((pts2, np.ones((N, 1))))
    prev_inliers = 0
    best_inliers = None
    best_F = None
    for i in range(nIters):
        pts_idx = np.random.choice(N, size = 5, replace = False)
        pts11 = pts1[pts_idx, :]
        pts22 = pts2[pts_idx, :]
```

```
        F = eightpoint(pts11, pts22, M)
        err = calc_epi_error(pts1_homogenous, pts2_homogenous, F)
        inliers = (err < tol)
        num_inliers = np.sum(inliers)
        if (num_inliers>prev_inliers):
            best_inliers = inliers
            best_F = F
            prev_inliers = num_inliers
    F_orig = eightpoint(pts1, pts2, M)
    err_orig = calc_epi_error(pts1_homogenous, pts2_homogenous, F_orig)
    inliers_orig = (err_orig < tol)
    num_inliers_orig = np.sum(inliers_orig)

    return best_F, best_inliers
```

The error metric used was sum-of-squares error. The difference between individual point pairs were calculated, squared and summed. Those points for which the sum-of-squared error was less than a threshold were considered as inliers and other points were considered as outliers.

Ablation study with constant iterations and varying tolerance.

Iterations	Tolerance	Inliers with RANSAC	Inliers without RANSAC
200	5	73	26
200	10	86	29
200	20	94	32
200	30	97	39

Ablation study with constant tolerance and varying iterations.

Iterations	Tolerance	Inliers with RANSAC	Inliers without RANSAC
10	10	33	29
100	10	67	29
200	10	86	29
300	10	86	29

As can be seen from the table above, when the tolerance is increased, the number of inliers increase, which is expected as more points which are far off are considered as good points.

No change in the F matrix is observed.

When the number of iterations are increased, we see an increase in the inliers. However

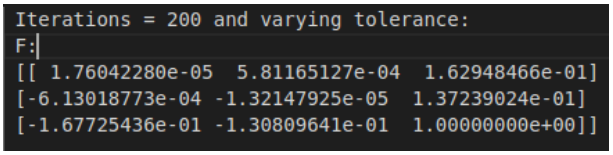

Figure 5: F matrix when obtained through RANSAC using noisy points. Iteration = 200 and tolerance = 5,10,20,30

no significant increase in inliers is observed beyond a certain value (86 in this case). This is reasonable because with increasing iterations, the chances of selecting the optimal points increases and the optimal points are reached within a certain number of iterations (200-300 in this case). The value of F also changes.

When F is calculated using only the points which are inliers, the value of F changes. The

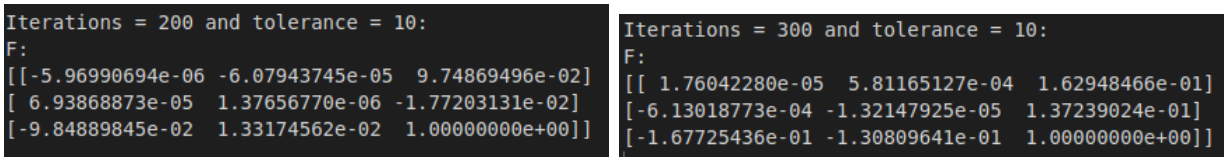

Figure 6: F matrix when obtained through RANSAC using noisy points. Iteration = 200, 300 and tolerance = 10

value of individual terms of F decrease.

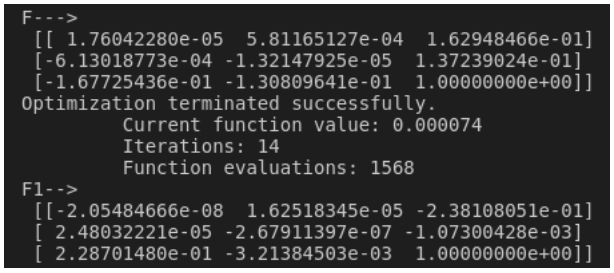

Figure 7: F matrix when obtained through RANSAC using noisy points and using only inliers.

5.2 Rodrigues and Inverse Rodrigues

```
def rodrigues(r):
    # Replace pass by your implementation
    r_norm = np.linalg.norm(r)
    theta = r_norm # radians
    if theta == 0:
        R = np.eye(3)
        return R
    u = r/r_norm
    R = np.eye(3)*np.cos(theta) + (1-np.cos(theta))*(u@u.T) + np.array([[0,-u[2],u[1]],
                                                                       [u[2], 0, -u[0]],
                                                                       [-u[1], u[0], 0]])*np.sin(theta)
    return R
```

```
def S_half(r):
    tolerance = 0.001
    r1 = r[0]
    r2 = r[1]
    r3 = r[2]
    if abs(np.linalg.norm(r)-np.pi)<0.001 and ((r1==0 and r2==0 and r3<0) or (r1==0 and r2<0) or (r1<0)):
        result = -r
        return result
    else:
        result = r
        return result

def invRodrigues(R):
    # Replace pass by your implementation
    A = (R-R.T)*0.5
    rho = np.asarray([[A[2,1], A[0,2], A[1,0]]]).T
    s = np.linalg.norm(rho)
    c = (np.sum(np.diag(R))-1)/2
    if s==0 and c==1:
        r = np.zeros((3))
    elif s==0 and c==-1:
        RI = R + np.eye(3)
        non_zero_indices = np.transpose(np.nonzero(RI))
        non_zero_col = non_zero_indices[0,1]
        v = RI[:,non_zero_col]
```

```
        u = v/np.linalg.norm(v)
        u_pi = u*np.pi
        r = S_half(u_pi)
        theta = np.arctan2(s,c)
    elif np.sin(np.arctan2(s,c))!=0:
        theta = np.arctan2(s,c)
        u = rho/s
        r = u*theta
    return r
```

5.3 Bundle Adjustment

```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation
    N = p1.shape[0]
    P = x[0:N-6].reshape((N,3))
    P = np.hstack((P, np.ones((N,1))))
    R2 = rodrigues(x[-6:-3].reshape((3,)))
    T2 = x[-3:].reshape((3,1))
    M2 = np.hstack((R2, T2))

    C1 = K1@M1
    C2 = K2@M2

    p1_projected = C1@P.T
    p2_projected = C2@P.T
    p1_projected = p1_projected/p1_projected[:,2]
    p2_projected = p2_projected/p2_projected[:,2]
    p1_projected = p1_projected[:,0:2]
    p2_projected = p2_projected[:,0:2]

    residuals = np.concatenate([(p1-p1_projected).reshape([-1]),(p2-
   p2_projected).reshape([-1])])

    return residuals
```

```
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    obj_start = obj_end = 0

    R2 = M2_init[:, 0:3]
    T2 = M2_init[:,-1]

```

```
    r2 = invRodrigues(R2).flatten()
    P_init = P_init.flatten()
    T2 = T2.flatten()

    x_init = np.concatenate((P_init, r2, T2))
    residuals = rodriguesResidual(K1, M1, p1, K2, p2, x_init)
    def blah(x):
        output = np.linalg.norm(x)
        return output
    obj_start = blah(residuals)
    x_update = scipy.optimize.minimize(blah, x_init, method = "CG")
    x_update = x_update.x
    obj_end = blah(x_update)
    N = p1.shape[0]
    P = x_update[0, N-6].reshape((N,3))
    R2 = rodrigues(x_update[-6:-3].reshape((3,)))
    T2 = x_update[-3:].reshape((3,1))
    M2 = np.hstack((R2, T2))

    return M2, P, obj_start, obj_end
```