Computer Vision
CS 6476 , Spring 2018

PS 3

Professor :
Cedric Pradalier

Author :
Melisande Zonta

March 1, 2018

Contents

1 Calibration **2**
- 1.1 Projection Matrix Computation . . . 2
- 1.2 Random Sets . . . 3
- 1.3 Camera Center Computation . . . 5

2 Fundamental Matrix Estimation **6**
- 2.1 Full Rank Fundamental Matrix Estimation . . . 6
- 2.2 Fundamental Matrix Computation . . . 6
- 2.3 Epipolar Lines Estimation and Drawing . . . 7
- 2.4 2D Points Normalization . . . 10
- 2.5 Accurate Fundamental Matrix Computation . . . 10

1 Calibration

The first part of this problem set consists in determining the projection matrix of a camera based on the location of known 3D points in the world and their corresponding 2D points in the image.

1.1 Projection Matrix Computation

The computation of the projection matrix is done using the svd (singular value decomposition) trick, as visible in the code below.

Indeed as the problem is presented as the product of the matrix M and the homogenous vector of 3D points :

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}$$

The above equality provides one pair of equations for each point such that :

$$\begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 & -u_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_nX_n & -u_nY_n & -u_nZ_n & -u_n \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 & -v_1 \end{bmatrix} \begin{bmatrix} m_{00} \\ m_{01} \\ m_{02} \\ m_{03} \\ m_{10} \\ m_{11} \\ m_{12} \\ m_{13} \\ m_{20} \\ m_{21} \\ m_{22} \\ m_{23} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This is a homogenous set of equations under the form $Ax = 0$. We can apply the singular value decomposition to A and then reshapedinto a 3 x 4 matrix that corresponds to our projection matrix.

```
def svd_solver(points_2D,points_3D):

    u = points_2D[:,0]
    v = points_2D[:,1]
    X = points_3D[:,0]
    Y = points_3D[:,1]
    Z = points_3D[:,2]

    A = np.zeros(((points_3D.shape[0])*2,12), dtype=np.float32)

    ones_vector = np.ones(points_2D.shape[0])
    zeros_vectors = np.zeros(((points_2D.shape[0]),4), dtype=np.float32)

    A[::2,:] = np.column_stack((X, Y, Z, ones_vector,
```

```
                                        zeros_vectors, -u*X, -u*Y, -u*Z, -u))
    A[1::2,:] = np.column_stack((zeros_vectors, X, Y, Z,
                                         ones_vector, -v*X, -v*Y, -v*Z, -v))

    U,S,V = np.linalg.svd(A)

    M = V.T[:, -1]
    M = M.reshape((3,4))

    residuals = residuals_calculation(points_2D, points_3D, M)

    return M, residuals
```

```
def residuals_calculation(points_2D, points_3D, M):

    residuals = np.zeros(points_2D.shape[0])
    for i in range(residuals.shape[0]):
        y = M.dot(points_3D[i,:])
        y = y / y[2]
        residuals[i] = np.sqrt(np.sum(np.power(points_2D[i,:] - y, 2)))
    return residuals
```

Applying this function to the sets of normalized 2D and 3D points, we get the following matrix:

$$\begin{bmatrix} 0.45827553 & -0.29474238 & -0.01395749 & 0.00402579 \\ -0.05085589 & -0.0545847 & -0.5410599 & -0.05237592 \\ 0.10900958 & 0.1783455 & -0.04426782 & 0.5968205 \end{bmatrix}$$

Finally, projecting the last 3D point given in the file (pts3d-norm.txt), we get the following projection on the 2D image:

$$\begin{bmatrix} 0.14190586 & -0.45183985 & 1. \end{bmatrix}$$

These coordinates match the ones given in the file (pts2d-norm-pic-a.txt) with a residual (distance) of 0.0016.

1.2 Random Sets

We now compute the projection matrix for random sets of point pairs with different sizes (8, 12 and 16).

We repeat the computation 10 times for each size of set.

For each iteration, we take 4 more random pairs of points (outside the ones used for computing M) and compute the average residual.

We also output the projection matrix corresponding to the lowest residual for each size of set.

The following code is used:

```
k = [8, 12, 16]
number_repetitions = 10


residuals_average = []
res_average_mins = []
M_mins = []

```

```
for i in k:
    residuals_average = np.zeros(number_repetitions)
    for n in range(number_repetitions):

        # Randomly choose k points from the 2D list and their corresponding points in the 3D list

        random_points = np.random.choice(array_3D_points_homog.shape[0], i, False)

        array_2D_points_homog_random = array_2D_points_homog[random_points,:]
        array_3D_points_homog_random = array_3D_points_homog[random_points,:]

        # Compute the projection matrix M on the chosen points

        M, res = svd_solver(array_2D_points_homog_random, array_3D_points_homog_random)

        # Select all the points not in the set of k

        array_2D_points_homog_rand_four = np.delete(array_2D_points_homog.copy(), random_points, 0)
        array_3D_points_homog_rand_four = np.delete(array_3D_points_homog.copy(), random_points, 0)

        # Pick 4 points

        four_points = np.random.choice(array_3D_points_homog_rand_four.shape[0], 4, False)

        # Compute the average residual
        residuals_average[n] = np.mean(residuals_calculation(array_2D_points_homog_rand_four[four_points,:],

    array_3D_points_homog_rand_four[four_points,:], M))

        res_avg_min = residuals_average[n]
        M_min = M

    res_average_mins.append(res_avg_min)
    M_mins.append(M_min)

    print("The average residuals for k = {} are \n".format(i))
    print(residuals_average, "\n")
    print("The projection matrix for the lowest residual (R = {:.4f}) is \n\n{} \n\n==========\n".format(res_avg_min, M_min))

min_i = np.argmin(res_average_mins)
print("The best projection matrix overall (k = {}) is \n\n {}".format(k[min_i], M_mins[min_i]))
```

We get the results presented on figure 1.

We notice that for a larger set of points, the average residuals decreases. This means that our projection matrix tends to give more accurate projections when 4 more pairs of points are used in the least squares solving.

This is true in this case because there is no outliers in the input data, i.e. the pairs of points are accurately matched. Should the input data be more realistic (noise and outliers),

```
The average residuals for k = 8 are

[1.46793264 2.88430476 4.80305479 2.86286081 1.55328151 2.39777702
 2.00707875 1.00474323 1.0773313  2.33994621]

The projection matrix for the lowest residual (R = 2.3399) is

 [[ 6.7675500e-03 -3.8541546e-03 -1.2847674e-03 -8.2805336e-01]
 [ 1.4815764e-03  1.0751752e-03 -7.1890261e-03 -5.6053466e-01]
 [ 7.4139011e-06  3.7938871e-06 -1.7991388e-06 -3.3574558e-03]]

==========

The average residuals for k = 12 are

[1.15288751 1.44591127 0.83373992 1.78353658 1.6896237  1.84603161
 4.45431258 1.15335758 0.78167892 1.15108178]

The projection matrix for the lowest residual (R = 1.1511) is

 [[ 6.9312383e-03 -4.0100962e-03 -1.3596062e-03 -8.2770216e-01]
 [ 1.5478667e-03  1.0203526e-03 -7.2820755e-03 -5.6104833e-01]
 [ 7.6053402e-06  3.7109310e-06 -1.9538422e-06 -3.3856912e-03]]

==========

The average residuals for k = 16 are

[0.98436573 1.52624674 1.03109302 1.67686973 0.92116178 1.28012127
 1.1756117  1.16469811 1.53265708 1.06363408]

The projection matrix for the lowest residual (R = 1.0636) is

 [[-7.0129670e-03  4.1052629e-03  1.2766799e-03  8.2553703e-01]
 [-1.5636986e-03 -1.0218868e-03  7.3489216e-03  5.6422675e-01]
 [-7.6971528e-06 -3.7045079e-06  1.8468583e-06  3.4145606e-03]]

==========

The best projection matrix overall (k = 16) is

 [[-7.0129670e-03  4.1052629e-03  1.2766799e-03  8.2553703e-01]
 [-1.5636986e-03 -1.0218868e-03  7.3489216e-03  5.6422675e-01]
 [-7.6971528e-06 -3.7045079e-06  1.8468583e-06  3.4145606e-03]]
```

Figure 1: Results of the random sets.

the number of pairs used for the computation should be chosen more carefully.

Indeed, a larger number of matching points will reduce the impact of the Gaussian noise, but if outliers are present, the resulting estimation of M will be impacted no matter the number of pairs used.

1.3 Camera Center Computation

The next step is to compute the location of the camera center in the 3D world, based on the previously determined projection matrix.

The coordinates are computed using the following function:

```
def camera_center(M): return np.dot(-np.linalg.inv( M[:,:3]) , M[:,3])
```

The resulting estimate shown below effectively matches the coordinates given in the problem set description.

$$\begin{bmatrix} -1.51576097 & -2.35511764 & 0.28244984 \end{bmatrix}$$

When applying the M matrix found in the previous function with the set of points pts2d-pic-b.txt and pts3d.txt, we obtain the following location of the camera in the 3D world :

$$\begin{bmatrix} 303.11692085 & 307.20745856 & 30.42544351 \end{bmatrix}$$

2 Fundamental Matrix Estimation

The objective of the second part of this problem set is to determine the fundamental matrix based on locations of pairs of matching points in two images.

2.1 Full Rank Fundamental Matrix Estimation

As a first step, we solve for the full rank fundamental matrix (Ft) using the singular value decomposition method as previously, that is visible in the code below. The resulting matrix is reshaped in a 3 x 3 form.

```
def svd_fundamental(points_2D_a, points_2D_b):

    u = points_2D_a[:,0]
    v = points_2D_a[:,1]
    u_p = points_2D_b[:,0]
    v_p = points_2D_b[:,1]


    A = np.column_stack((u*u_p, v*u_p, u_p, u*v_p, v*v_p, v_p, u, v, np.ones(
    points_2D_a.shape[0])))

    U,S,V = np.linalg.svd(A, full_matrices=True)
    F = V.T[:,-1]
    F = F.reshape((3,3))

    return F
```

Applying this function to the points in our files (pts2d-pic-a.txt and pts2d-pic-b.txt) we get the following result:

$$\begin{bmatrix} -6.60698417e-07 & 7.91031621e-06 & -1.88600198e-03 \\ 8.82396296e-06 & 1.21382933e-06 & 1.72332901e-02 \\ -9.07382302e-04 & -2.64234650e-02 & 9.99500092e-01 \end{bmatrix}$$

2.2 Fundamental Matrix Computation

In order to get the actual fundamental matrix from this full rank estimate, we compute the Singular Value Decomposition (SVD) of Ft.

$$U\Sigma V^T = F_t$$

To estimate the rank 2 matrix, we first set the smallest singular value (the last one in our case, since numpy.linalg.svd sorts them in a decreasing order). Then we compute the fundamental matrix F using the following product:

$$F = U\Sigma' V^T$$

The following code implements this computation:

```
# Rank reduction using SVD
u, s, v = np.linalg.svd(Ft)
s[-1] = 0
F = u.dot(np.diag(s)).dot(v)
```

As a result, and based on the previously determined F_t matrix, we get the following fundamental matrix F:

$$\begin{bmatrix} -5.35883058e-07 & 7.89972529e-06 & -1.88480998e-03 \\ 8.83820595e-0 & 1.21802118e-06 & 1.72276843e-02 \\ -9.08539026e-04 & -2.64201801e-02 & 1.00000000e+00 \end{bmatrix}$$

2.3 Epipolar Lines Estimation and Drawing

Now we want to draw the epipolar lines on both input images.

We use the homogeneous coordinates duality between points and lines to get to the desired result.

We first need to compute the coordinates of the left and right edges of the image. This is achieved by computing the cross product of respectively the top and bottom left corner coordinates and top and bottom right corner coordinates, as follows (where m and n are the width and height of the image):

$$L_l = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ n \\ 1 \end{bmatrix}$$

$$L_r = \begin{bmatrix} m \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} m \\ n \\ 1 \end{bmatrix}$$

Next, for each 2D point, we compute the epipolar line using the dot product of F (respectively F^T) and the point homogeneous coordinates.

$$L_i = F. \begin{bmatrix} x_j \\ y_j \\ 1 \end{bmatrix}$$

$$L_j = F^T. \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Then we compute the homogeneous coordinates of the intersection points of L_i with L_l and L_r using a cross product:

$$P_{il} = L_i \times L_l$$

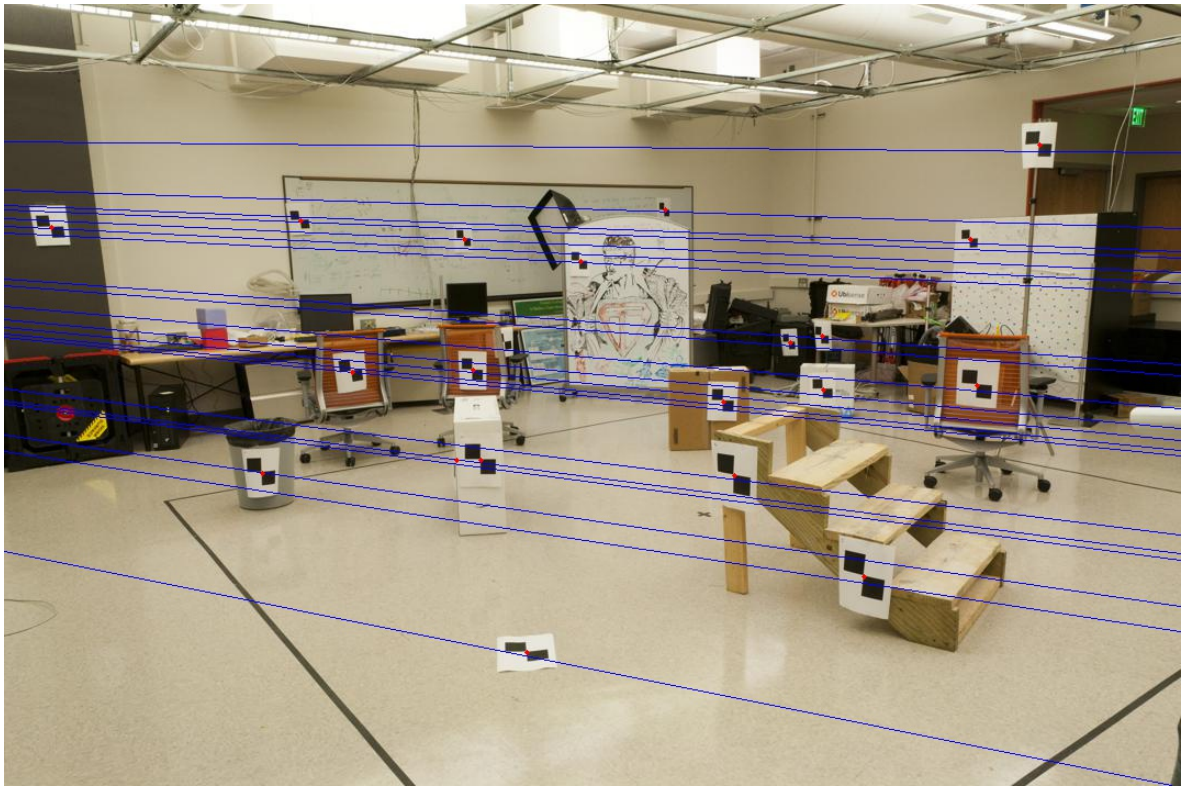
$$P_{ir} = L_i \times L_r$$

We finally draw the line on the image based on the cartesian coordinates of those two points.

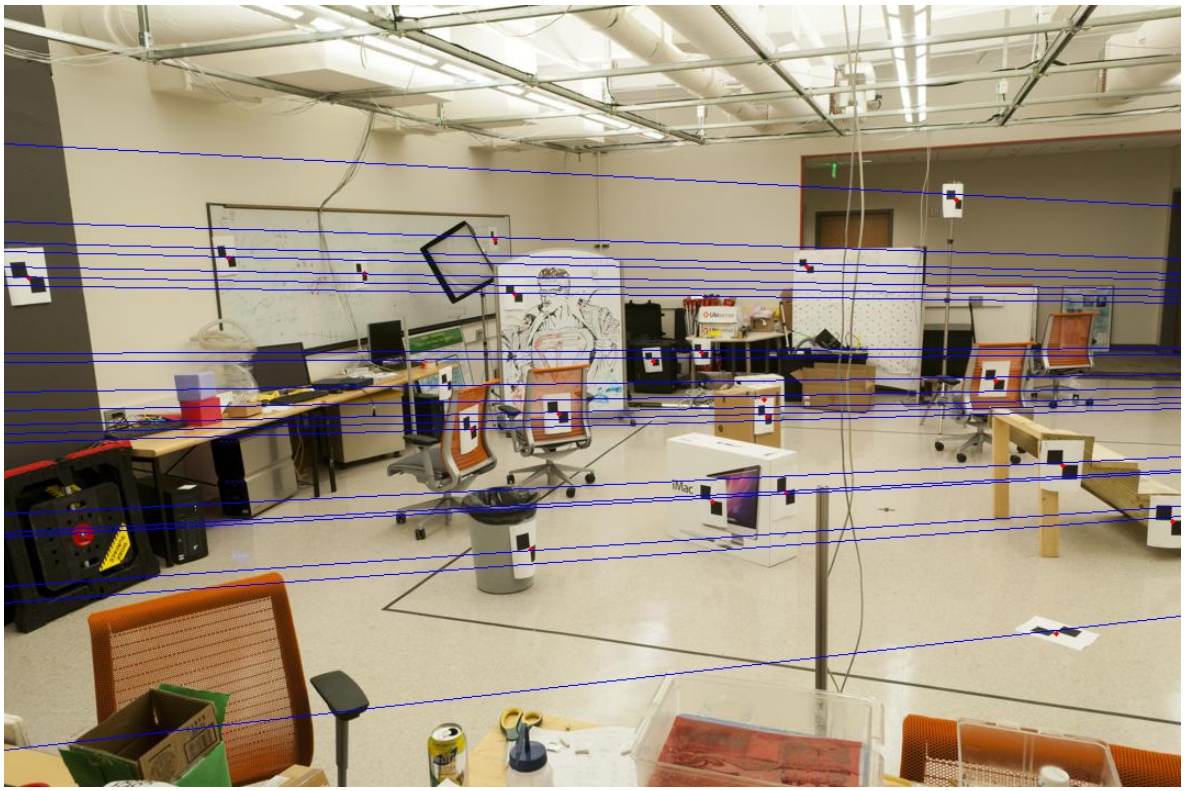
The previous equations were implemented in this code :

```
def epipolar_lines(image_a,image_b, array_2D_points, F):

    (rows,cols,_) = image_a.shape

    # Ll is defined as the line corresponding to the left hand side of the image
    # Lr is defined as the line corresponding to the right hand side of the image

    Ll = np.cross([0,0,1],[0,rows,1])
    Lr = np.cross([cols,0,1],[cols,rows,1])

    for i in range(array_2D_points.shape[0]):

        Li = F.dot(array_2D_points[i,:])
        Pil = np.cross(Li, Ll)
        Pil = Pil/Pil[-1]
        Pil = tuple(np.round(Pil[:-1]).astype(int))

        Pir = np.cross(Li, Lr)
        Pir = Pir/Pir[-1]
        Pir = tuple(np.round(Pir[:-1]).astype(int))


        cv2.line(image_a, Pil, Pir, (255,0,0), 1)

        markers = tuple(np.asarray(array_2D_points[i,:-1], dtype="int"))
        cv2.circle(image_b, markers, 2, (0,0,255), -1)
```

Computing the epipolar lines for the image A (using the image B 2D points coordinates) results in Figure 2a. Similarly, computing for the image B (using the image A 2D points coordinates) gives us the Figure 2b .



a



b

Figure 2: *a.* ps3-2-3-a. *b.* ps3-2-3-b.

2.4 2D Points Normalization

In order to correct the innacuracies of our initial estimate, we need to normalize the 2D points using a scale and offset transformation.

We first compute the transformation matrix using the inverse of the maximum coordinate value as a scale and the opposite of the means as offset, as shown in the code below:

```
def transform_matrix(array_2D_points):

    s = np.max(np.abs(array_2D_points))
    S = np.array([[1/s,0,0],[0,1/s,0],[0,0,1]])

    cu = np.mean(array_2D_points[0])
    cv = np.mean(array_2D_points[1])

    C = np.array([[1, 0, -cu], [0, 1, -cv], [0, 0, 1]])

    T = np.dot(S, C)
    return T
```

The two transformation matrices we obtain for the A and B 2D points are the following: The transformation matrix for the 2D points of image A is :

$$\begin{bmatrix} 1.06044539e-03 & 0.00000000e+00. & -5.80063627e-01 \\ 0.00000000e+00 & 1.06044539e-03 & -1.30434783e-01 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

The transformation matrix for the 2D points of image B is

$$\begin{bmatrix} 9.39849624e-04 & 0.00000000e+00 & -4.55357143e-01 \\ 0.00000000e+00 & 9.39849624e-04 & -1.26879699e-01 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

Then we compute an intermediate fundamental matrix (Fh) using the normalized 2D points as an input:

$$\begin{bmatrix} -0.01998282 & 0.28265631 & -0.0759324 \\ 0.18524791 & -0.05514497 & 0.5972918 \\ -0.00377719 & -0.72075553 & 0.01782818 \end{bmatrix}$$

2.5 Accurate Fundamental Matrix Computation

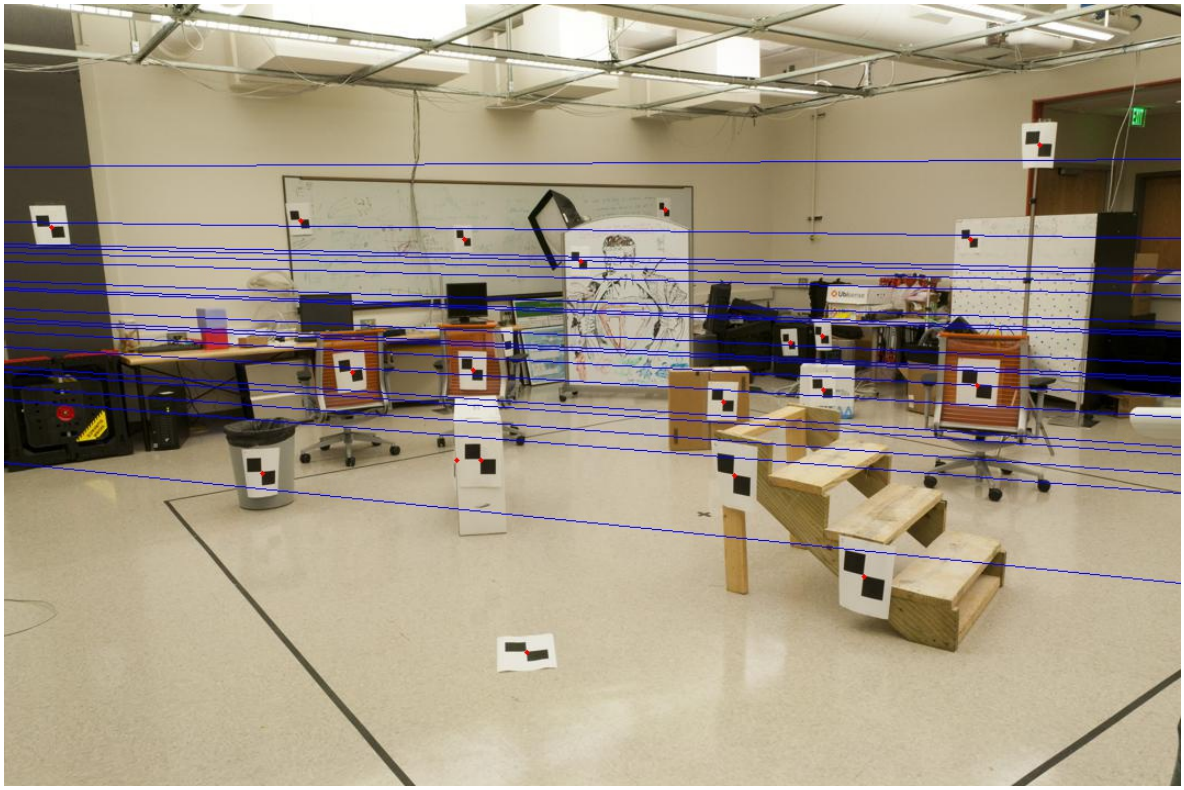
The last step is to compute the more accurate fundamental matrix F based on the previously determined Fh.

$$F = T_b^T.F_h.T_a$$

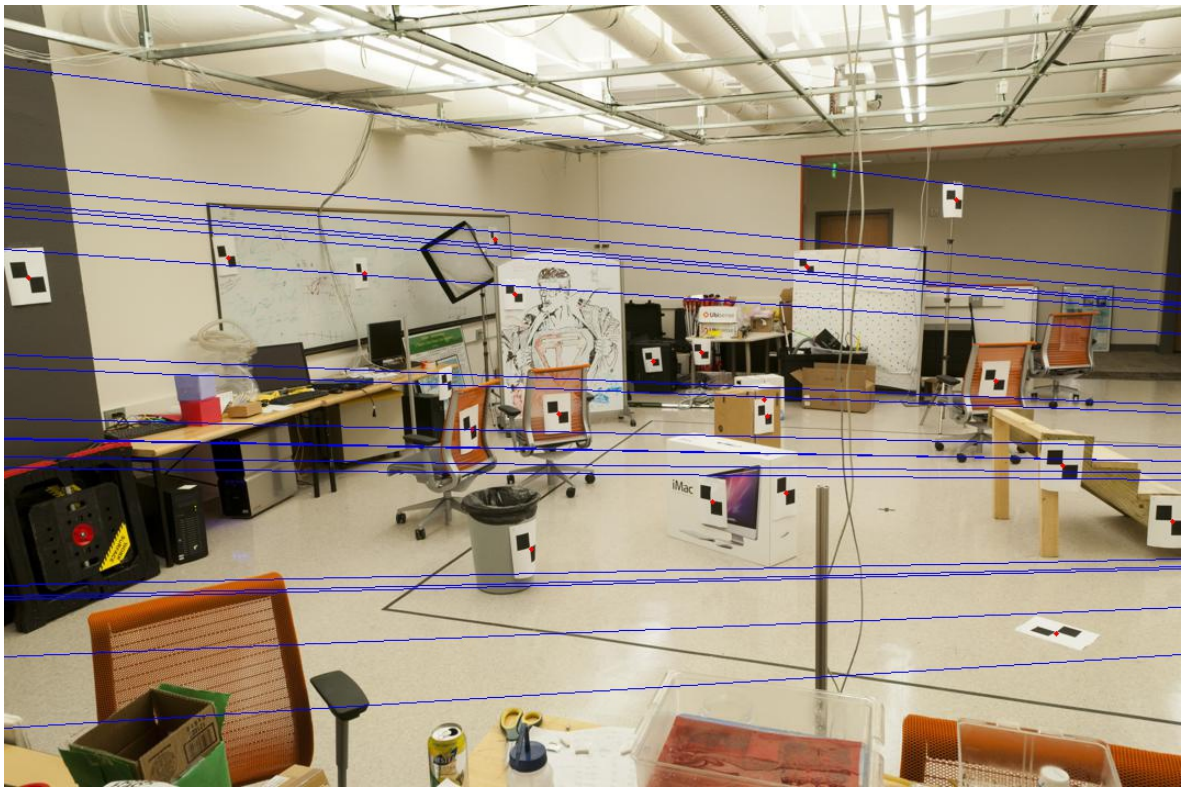
We get the following fundamental matrix:

$$\begin{bmatrix} -1.99160637e-08 & 2.81712007e-07 & -9.51215321e-05 \\ 1.84629034e-07 & -5.49607394e-08 & 4.67132566e-04 \\ -1.92810921e-05 & -8.93391641e-04 & 9.70542710e-02 \end{bmatrix}$$

Finally, we can draw again the epipolar lines on both images using the new fundamental matrix. The result, more accurate, is visible in Figure 3a. and Figure b. Obviously it didn't work, and I couldn't figure why since it seemed I wrote every formula correctly.



a



b

Figure 3: *a.* ps3-2-5-a. *b.* ps3-2-5-b.